Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Data-oriented Reengineering

Dissertation submitted in partial fulfillment
of the requirements for the degree of
„Doctor of Natural Science“ (Dr. rer. nat.)

Schriftliche Arbeit
zur Erlangung des Grades
"Doktor der Naturwissenschaften" (Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Jörg P. Wadsack
Ferrariweg 34
33102 Paderborn

Paderborn, Dezember 2003

ABSTRACT

Today, information system evolution primarily consists of extending the legacy systems and migrate them to modern platforms related to the Web and mobile devices. This dissertation tackles the problem of understanding and adapting legacy web information systems based on the systems' data. In this context several methods and tools have been proposed for reengineering. Since legacy systems have grown over years and lack sufficient documentation, reengineering is a complex and hard task. Tool supported reengineering approaches and processes reduce the complexity and risks during web information system maintenance. Still, current reengineering approaches and tools often tackle only specific system parts or dedicated maintenance aspects. This thesis aims to overcome these limitations by providing a process that combines tools for reengineering the data as well as the applications. Our focus is to handle uncertain knowledge by sustaining human exploration and iteration during a tool supported reengineering process. In practice uncertain knowledge during reengineering plays a fundamental role but it is often neglected by exiting approaches due to idealistic assumptions.

The presented data-oriented reengineering process provides concepts that combine existing data and application design reengineering approaches to maintain web information systems. Since it is unrealistic to presume that reengineering can follow a strictly waterfall like process without iterations, inconsistencies occur during the reengineering process. The chosen combined approaches fulfil the requirement to deal with such inconsistencies. We based our process on models because models provide the possibility to represent (inconsistent) knowledge at different levels of abstraction. Moreover, models can be accurate enough to enable code generation. In this dissertation, we base the system's models on graphs. We employ graph transformation theory to provide mechanisms that detect, handle and resolve inconsistencies of the models automatically. The results are implemented within the REDDMOM project. We use the FUJABA TOOL SUITE for tool integration and evaluate our concepts with a case study in the Health Care domain.

AKNOWLEDGEMENTS

Many people have influenced my research during the past five years. I am especially obliged to Wilhelm Schäfer who supported me in every occasion and provided a phenomenal working environment. Special thanks go to Jens Jahnke and Albert Zündorf. Both are „responsible“ for this thesis since they convinced me that I am able accomplish it. Beside office and beer sharing, the fruitful discussions with them were the basis for most of the thesis results. Jens Jahnke welcomed my as a member of his research group at the University of Victoria and as a guest in his house. Many thanks to Anke at this point! During this period the fundamental parts of this thesis were settled. A special thank you goes to Jörg Niere for many fruitful discussions about our theses.

The achievement of this thesis would not have been possible without the contributions of many colleagues and students. I thank all persons involved in the REDDMOM project. Everyone that somehow contributed to the FUJABA environment: thanks. Further, I thank all people at the University of Victoria, especially those involved in the palliative care case study, for the great support during my stay. Finally, I thank all my colleagues of the Software Engineering Group sharing ideas, jokes, planes, beds, offices, (sparkling) wine, beer and coffee with me.

Special thanks go the „proof readers“: Holger Giese, Jens Jahnke, Ekkart Kindler, Jörg Niere, Matthias Meyer, Wilhelm Schäfer, Matthias Tichy, Robert Wagner, Lothar Wendehals and Albert Zündorf.

I thank Jürgen Maniera for the great technical support. I am obliged Jutta Haupt who helped me surviving the bureaucracy jungle and for many chats.

Without the support of my friends and my family, especially my wife Sonja and my son Simon, this would never have been possible.

I love you.

To my family

CONTENTS

List Of Figures And Tables

CHAPTER 1: INTRODUCTION

Observe constantly that all things take place by change, and accustom thyself to consider that the nature of the Universe loves nothing so much as to change the things which are, and to make new things like them.

MARCUS AURELIUS ANTONINUS
Roman emperor and philosopher (121 - 180)

1.1 Background: Web Information System Reengineering

Current trends in the field of information technology, like eHealth, eGovernment or eProcurement, lead to the emergence of web information systems. Today's (web) information systems require constantly functional extensions due to new requirements, i.e., the heterogeneous distributed information systems become increasingly heterogeneous and distributed. Further, these (web) information systems interface more and more with the clients through the Web and include a growing number of mobile devices.

These web information systems are legacy systems, i.e., they are inherited systems which have evolved over years and still evolve. These systems are mission-critical to the companies. Further, these systems are generally poorly documented and only partially understand. New development or even replacement of the legacy systems is often not practicable. Web information system's reengineering is the only solution to keep the mission-critical systems running and to manage their inherent evolution. „Reengineering (...) is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.“ -- [CC90]

The evolution of web information systems is intrinsic to their existence. The enduring, highly complex and rapid evolution of web information systems requires continuous reengineering processes. Iterations and incremental changes occur during reengineering processes. Reengineering is typically highly explorative, i.e., human-centered, because of incomplete and uncertain knowledge. To manage the reengineering process' complexity tool support is needed [JW00b].

Maintenance covers over two thirds of the expenses spent in information technology [Bas90, ZSG79]. Reengineering, including re-documentation, activities accounts for the largest part of these budgets. Nevertheless, the legacy web information systems have to be kept running and thus the business logic and the data have to be reengineered. The business logic processes the data. Consequently, understanding a web information system

consists of understanding the data as well as the business logic. Finally, the maintenance of the regained knowledge is crucial for the reengineering process and necessary to avoid cost explosion.

1.2 Problem Definition

The persistent data structure is the central part of a legacy web information system [Aik96]. Thus, the basis for web information system reengineering is to understand the data organisation of the system. The importance of the system's data and its organisation is further reflected in the increasing interest dedicated to data security, data mining, etc.

The volume of data, the data heterogeneity and the data distribution in web information systems determine the system's complexity. Since volume, heterogeneity and distribution of the data increase, the complexity of the system increases. Moreover data consistency (management) increases the system's complexity. Data consistency is complicated by the volume, the heterogeneity and the distribution of the data. This complexity of web information systems makes data reengineering hard.

The fact that business logic and data (structures) are interwoven, makes data reengineering of web information systems even harder because it increases the complexity of the systems further. Unfortunately, systematic processes and techniques for data reengineering of web information systems are hardly available.

Assume a simple two tier architecture for web information systems composed of the data and the applications. Existing reengineering approaches cover either the data or the applications. Feasible data reengineering approaches tackle only single data repositories or databases, e.g., [EH99, Jah99]. Other approaches focus on application design recovery, e.g., [MTW93, GCBM96, KSRL01, Nie03]. At a higher level of abstraction, application architecture recovery, e.g., [BH99, GAK99, SK01, PG02], identifies the component interaction but rarely establishes a link to the data or application design.

Process guidelines in form of strategic decisions for legacy system reengineering are presented, e.g., in [Mil98, War99]. System evolution support is often considered only for new software, e.g., [Gro01, OLHL02], or only of small dedicated parts, e.g., [ORH02]. A process that supports system evolution, especially system extension, considering the data and the applications do not exist. Such a process must recover documentation to enable understanding and facilitate reliable extension to keep the legacy web information systems running.

1.3 Our Data-oriented Reengineering Approach

Our approach is focused on reengineering the data organisation of web information systems. We call our approach *data-oriented reengineering*. We base our data-oriented reengineering process on component-based software engineering and model-driven development (for forward engineering). Since the starting point is a legacy system, our

process deals with pre-existent (data-related) system parts, i.e., data components (e.g., a database or a data replication service). Our approach is model-based because abstraction through models enables better understanding and facilitates system extension.

We provide a semi-automatic process for data reengineering of web information systems that manages incomplete and uncertain knowledge and supports iterations. To resolve inconsistency and uncertainty emerging during the process, human interaction is needed. Iteration is needed to explore the existing system by making assumptions that can be validated or refuted. To unburden the reengineer from error-prone and recurrent tasks, tool support automating these tasks is indispensable.

We divided the process in two phases

- Understanding the system's data components and how they interoperate and
- Adapting the data components to the (new) requirements.

Figure 1.1 shows the data-oriented reengineering process. The first phase is Data-oriented Reverse Engineering. „Reverse engineering is the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at higher level of abstraction.“ -- [CC90].

We start with the reengineering of the data. The data models of the different legacy system databases are recovered based on Jahnke's data reverse engineering approach [Jah99]. These data models are recovered and mapped to a representation at higher level of abstraction. Further, these data models are refactored by the reengineer.

The application code is analysed to restore hidden data dependencies that are fundamental for the system's understanding. These data model relationships are retrieved based on Niere's (application) design recovery approach [Nie03]. Further, application models are retrieved and linked to the data models. The result of this understanding phase are Data Component Models.

The Data Component Extension is the second phase. Since in both phases the Data Component Models are transformed, for some activities we use the same tools in both phases. The adaptation of the data components aims at extending the system by transforming the retrieved conceptual data component models and design new parts by model-driven development. Therefore the retrieved and new data components are clustered and classified before they are redesigned. We propose architectural patterns to (re)structure the data components. Based on the regained knowledge an interface to the legacy databases can automatically be generated to access the data. Finally, the old retrieved and newly created models can be executed.

To support exploration and iteration during long-lasting reengineering, the Data Component Models have to be maintained. Model Maintenance means in this context that the different models are kept consistent to each other, i.e., changes in one model that affect other models are propagated. Therefore, we provide model consistency management. The web

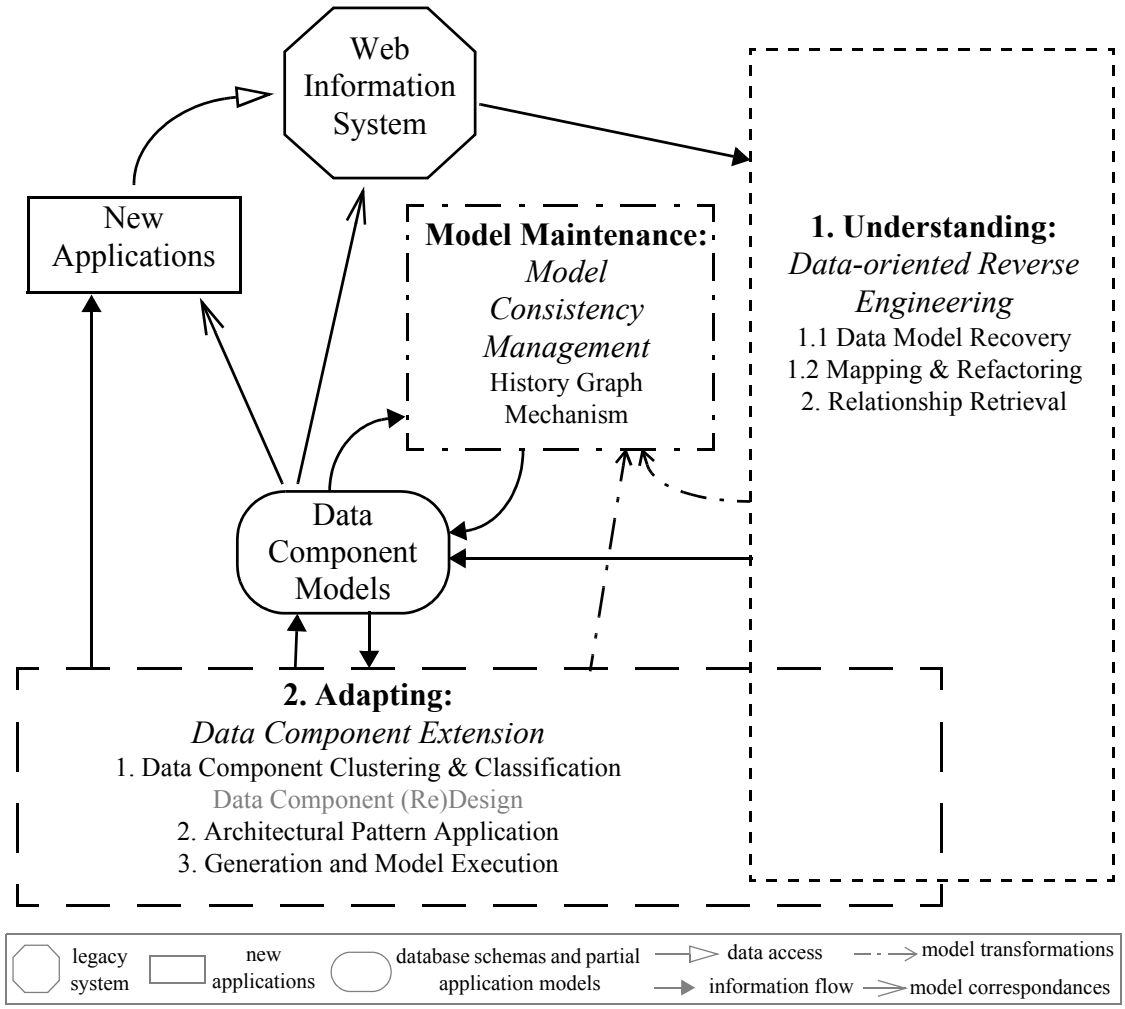


Figure 1.1: Data-oriented Reengineering process

information system Data Component Models, including their representation at different level of abstraction, are internally represented as graphs. We elaborate a mechanism, called History Graph Mechanism, that provides consistency for all model-based systems parts through explicit model transformation logging.

We consider the two phases as distinguishable states in the recurring cycle of iterations during data-oriented reengineering. Our process is flexible such that the phases as well as the activities performed during these phases are extendable and exchangeable. This implies that the participating tools can be exchanged by newer versions or by more adequate tools, and that new tools can be integrated.

The main research contributions of this dissertation have partly been published and can be summarised as follows:

- We have developed the data-oriented reengineering process to support the maintenance of web information system. The process allows iterations and the management of uncertain and inconsistent reengineering knowledge. The regained information is documented with conceptual models. Enduring reengineering after an initial reverse engineering phase is possible [SWJ04, Wad03, JSWZ02, WJ02, GW01, JW99a].
- Our data-oriented reverse engineering is a combination of a data reverse engineering approach and an application design recovery approach. To this end, we defined patterns to detect dependencies between the data component models in the application code [NWW03, WNGJ02, NSW+02, NWZ01, NWW01, JNW00, JW00a, JW99c].
- For web information system extension we provide
  - clustering to partition the web information system into different data components and a classification for these data components, and
  - architectural patterns to (re)structure the web information systems [JGW02] before the parts required for extension can be (re)design.
- Based on the Data Component Models we provide code generation for an access layer and model exection. We maintain all the (regained) knowledge about the web information system in these models at the different level of abstraction. This enables code generation for accessing the legacy databases as well as for the newly designed parts.
- Iteration and exploration during the process as well as code generation are based on the Model Maintenance. We keep the Data Component Models consistent with our History Graph Mechanism [WJ03, JWZ02, ZWR01, JW99b].
- We implement our approach in the REDDMOM project as part of the FUJABA TOOL SUITE (FUJABATS), partially by integrating existent tools, and use a case study in the Health Care domain as a proof of concept [BGN+03, NNWZ00].

1.4 Dissertation Outline

This dissertation is organised as follows:

Chapter 2 describes a case study in the growing field of Health Care. We describe the overall case study and point out our involvement. We use this case study to exemplify our concepts and techniques, i.e., it serves as running example for this dissertation.

Chapter 3 introduces our data-oriented reverse engineering process that enables legacy web information system understanding. We use a flexible combined data and design reverse engineering approach based on graph transformations. The recovered database logical schemas are mapped into a conceptual representation. This rough object-oriented representation is improved by refactoring operations and enriched with inter-schema de-

pendencies and relationships to application models. Inconsistency and uncertainty occurring during the process is managed using fuzzy logic.

Chapter 4 covers the model-driven extension of the web information systems. To classify the data components we cluster them following the adopted integration strategy. We propose architectural patterns for data mediation. Based on the retrieved web information system and newly designed models, code is generated.

Chapter 5 outlines the consistency management of the compiled data component models. Our History Graph Mechanism is based on composed graphs of the models where selective undo operations can be performed. The consistency between model and code is resolved by code generation.

All three technical chapters (3, 4 and 5) are closed with the corresponding tool support, related work and a short discussion. Chapter 6 concludes the dissertation. The transferability of results is presented and open questions and future directions of this work are discussed.

CHAPTER 2: DATA-ORIENTED REENGINEERING: A CASE STUDY

> *Pain (any pain--emotional, physical, mental) has a message. The information it has about our life can be remarkably specific, but it usually falls into one of two categories: "We would be more alive if we did more of this," and, "Life would be more lovely if we did less of that." Once we get the pain's message, and follow its advice, the pain goes away.*
>
> PETER MCWILLIAMS, *IN LIFE 101*
> United States Author (1950 - 2000)

The aim of this chapter is to corroborate the need of a data-oriented reengineering with a case study. Current approaches provide only little support for data-oriented reengineering of web information systems. Most of them do not consider the evolutionary and exploratory nature of the (data-oriented) reengineering process [HEH+95]. Tools are indispensable to unburden the reengineer from manually error-prone and/or repeating assured tasks. A semi-automatic data-oriented reengineering process able to handle inconsistencies and uncertainty is required. This process must allow human interaction and iteration to sustain exploration and evolution.

Based on the definitions of reengineering [CC90, TS95] and data reverse engineering [HA00, HHH+00] we define:

- Data-oriented reengineering is the systematic transformation of existing system parts concerning the persistent data into a new form to realise improvements. This includes some data-oriented reverse engineering followed by some data-oriented forward engineering activities.
- Data-oriented reverse engineering is the process of recovering a conceptional (high level of abstraction) representation of the persistent data to facilitate the understanding of the data structure and interrelationships of an existent system.
- Data-oriented forward engineering is the process of modifying an existing system's conceptional, implementation-independent data structure representation followed by the physical implementation.

The case study of the Health Care web information system shows that our tool-supported data-oriented reengineering approach is able to handle uncertainty and inconsistencies during the data-oriented reverse engineering process. Further, the case study demonstrates that our approach supports iterations. In this chapter we give a case study overview, detailled examples are given within the next chapters.

2.1 An Health Care Web Information System

In Health Care data is fundamental. Indeed, without patient data, drug data, health research data, etc., prevention or treatment is not possible. In Health Care, data has long only been collected, but over the last few years it has been realised that valuable knowledge may be contained within the data. The existing web information systems have to be extended for achieving data knowledge extraction beside data collection and data storage [RT02, Wil03].

By using data analysis tools such as data mining tools, knowledge can be extracted from the data so that it may be used to improve the delivery of Health Care. Knowledge discovery within Health Care has been described as an enumeration of symbols and the arrangement of those symbols into meaningful structures [Cim00]. Such a process is made easier if the data, representing the symbols, share common terminology and definitions. However, with different organisations having different data models and data definitions, data that is gathered for analysis is often heterogeneous. By using mediation technology it is possible to gather data from multiple health organizations and converge them, e.g., into a common data warehouse with a common data definition [Wei02].

To build a Health Care web information system, which contains information from a multitude of sites and can be widely accessed, data management facilities have to be put in place. Only collecting data, storing data and extracting knowledge from data is not enough. Beside the management of data on the local sites, a global data management is required. Further various mediation strategies are needed to fulfil the requirements of today's various static and mobile clients.

A central point of (Health Care) web information systems is that they base on distributed legacy systems containing the data. These legacy systems have to be analysed before they can be integrated together.

2.1.1 The Legacy (Web) Information System

The case study involves a *Palliative Care Network System* in Canada. The goal is to establish a network between different canadian Palliative Care Centers. The current situation is that several databases exist in each center. The different centers are not connected and have different data models. Further a *Palliative Care Data Warhouse*, where all data are collected and processed, is planned.

Data from a single Health Care institution is limited in its knowledge discovery potential. The ideal situation is to incorporate data from multiple sites to increase the knowledge discovery potential. However, that presents the problem of different formats and definitions of data from different sites. *Palliative Care Center Victoria*, e.g., has different data definitions and data collection requirements than either Palliative Care Center X or Y[1]. A

1. For political reasons, we are not able to make public the names of the involved centers.

further complication is that the data at different institutions may actually be stored in different database applications such as Microsoft Access, Microsoft SQL Server or Oracle. Another problem, which is experienced, is that there may be different information needs within the same center. Palliative Care Centers X or Y may gather large volumes of patient data, whereas individual sub-organisations such as pharmacy, lab and finance may only need certain subsets of that data.

Another problem is the actual collection of data. As traditionally done in health institutions, much of the data collection such as patient charting is done in a paper-based format that is later transcribed into an information system. Because that method requires a duplication of efforts, there is a potential for errors to be made during the process. Collecting patient data with mobile devices such as PDAs, electronic tablets and cellular phones is a way to avoid this error prone effort. Many Health Care institutions are experimenting in this direction [MRF+03]. However, because of differences in data collections from the different mobile devices it can still give a heterogeneous collection of data.

Currently, the University of Victoria is building this Palliative Care Network System. The tasks are (1) the reengineering and of the existing systems; and (2) the construction of an Health Information Grid for sharing the information [BBD+03].

The first task, i.e., data-oriented reengineering, is subject of the thesis. Our involvement in the project is restricted to a part of this Palliative Care Network System. This has two main reasons. Firstly, the period of our involvement was limited. Secondly, for political reasons not all the needed system insights were given to us or can be presented in this thesis. Therefore we concentrate on one Health Care web information system: the Palliative Care Center Victoria.

The second task, the integration of different Health Care systems in a configurable network of interconnected organisations, is achieved by wrapping the existing systems [OBJ03]. These wrappers use web services and industry standards to provide a uniform and adaptable interface with functionalities for interoperability as well as secured and controlled access among the individual systems. This is done within a *Grid Federation Envelope* described in [Ona03].

2.1.2 The Considered Target Web Information System

Figure 2.1 depicts the target systems architecture. The Palliative Care Center Victoria is composed of three Microsoft Access databases (DB), an Application and two portals. After reverse engineering and wrapping the three databases, the data mediation portal (DMP) and the WebPortal are constructed. The WebPortal enables new access from inside the center with browsers. This WebPortal runs in parallel with the existing applications. The data mediation portal (DMP) provides access through Data Mediation Services to the other two subsystems.

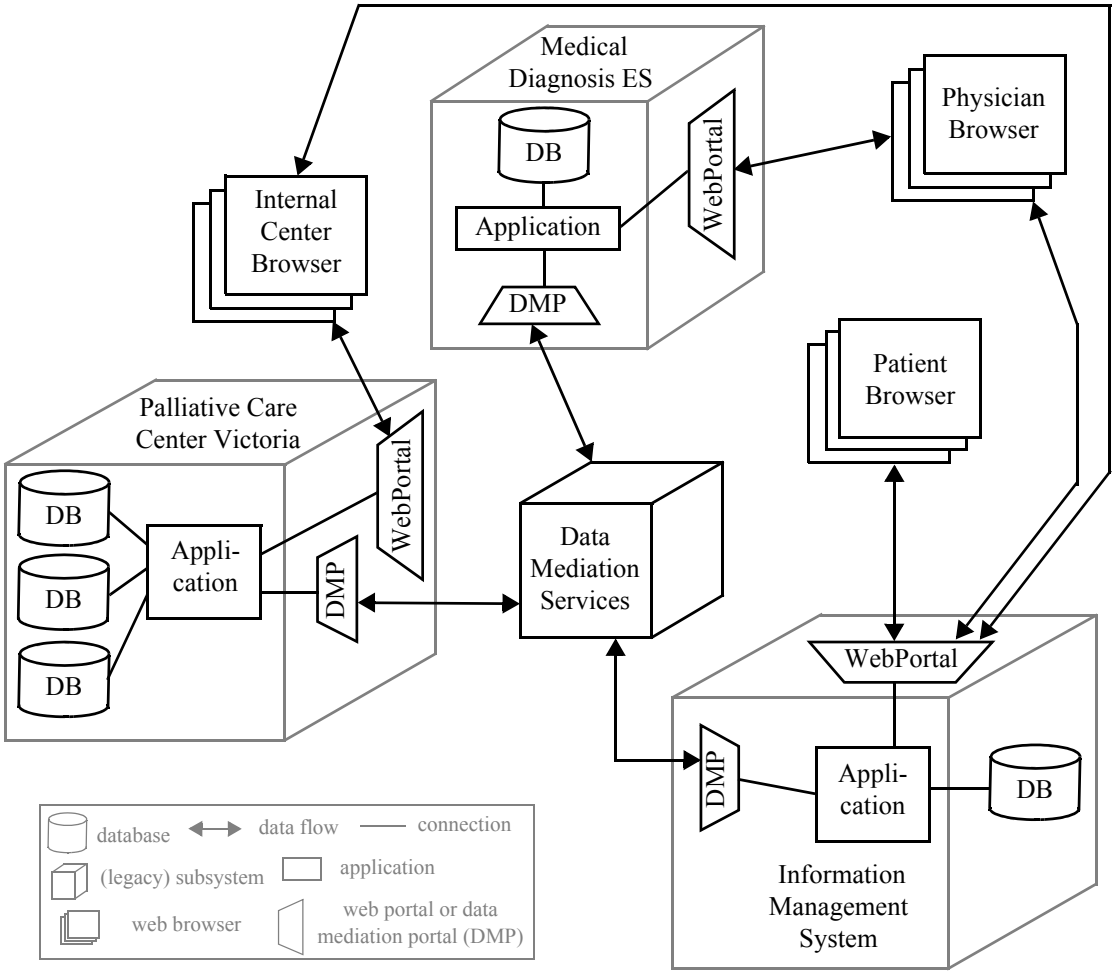


Figure 2.1 Health Care Web Information System: Considered Target System

The system incorporates three kinds of static clients in form of browsers. Except the data flow from the WebPortals to the browsers, all other data is exchanged via Data Mediation Services.

The Data Mediation Services play a central role in this architecture by managing the data flow between the other subsystems. The benefit of such services is their flexibility, loose coupling and technology independence. Each service passes the data to a mediator that puts the data into a common format before storing it in a database. The data, regardless of its initial format or the application in which it was created, is now in a format which is much more valuable for data analysis and knowledge discovery.

The Information Management System is composed of a database (DB), an Application and portals. The database stores all system users' information like roles, access rights, document versions, etc. The Application incorporates the management tasks regarding access

rights and documentation. The WebPortal allows access from all browsers with respect to the users' role. The data mediation portal (DMP) is used, via the Data Mediation Services, by the other subsystems for user identification or document access. Further it is extended to handle multimedia data.

The Information Management System is based on two systems, DSD and BEETLE, that are customised for handling documents in a medical context. At the software engineering group from the University Paderborn the distributed software development (DSD) system [GGT01] was build. The core of the system is based on the concurrent version system [CVS]. The concurrent version system functionality is encapsulated in Jini services [AOS+99]. A pure Java client permits the access of the Jini services and the Sybase database. Moreover a document report system called BEETLE exists. This system is based on a MySQL database, Java and Java Server Pages. The combination of these two systems enables the versioning and processing of documents during workflows in the Health Care domain.

The Medical Diagnosis Expert System is a prototype that includes one database (DB) hosting the knowledge base. The Application includes the inference and the inference Application Programming Interface (API). This API is accessible through two portals. The WebPortal permits access from a browser used by physicians. The data mediation portal (DMP) permits the access of electronic patient records for the diagnosis and treatment.

2.2 The Data-oriented Reengineering process

2.2.1 Understanding Phase

We follow a divide and conquer approach in the understanding phase and aim for a rapid (semi-)automatic bottom-up reverse engineering step followed by a top-down explorative human-driven reverse engineering. This activity is based on Jahnke's *database reverse engineering* approach VARLET [Jah99] and Niere's *incremental pattern instance recognition* approach [Nie03].

Figure 2.2 gives an overview of the understanding phase. Conceptual schemas of the databases are retrieved. During this process logical schemas are also recovered. The relationships are retrieved from the application code. In addition to the schemas, parts of the applications that are related to the schemas are also retrieved. The results of this phase is one cluster that contains data models from the different (web) information system.

The process starts with the reverse engineering of the logical schemas. Figure 2.3 sketches the schemas of the three databases, namely hospdata, bvmtdata and outcomes_be, of the Palliative Care Center Victoria. The parsed initial logical schemas only contain the primary and foreign keys that are explicitly defined in the databases. This step is followed by the schema analysis that consists of *structural completion* and *semantical enrichment* [Jah99]. The next step is the mapping from the logical schemas to conceptual schemas. A set of triple-graph-grammar rules [SL96] creates this conceptual representation in the

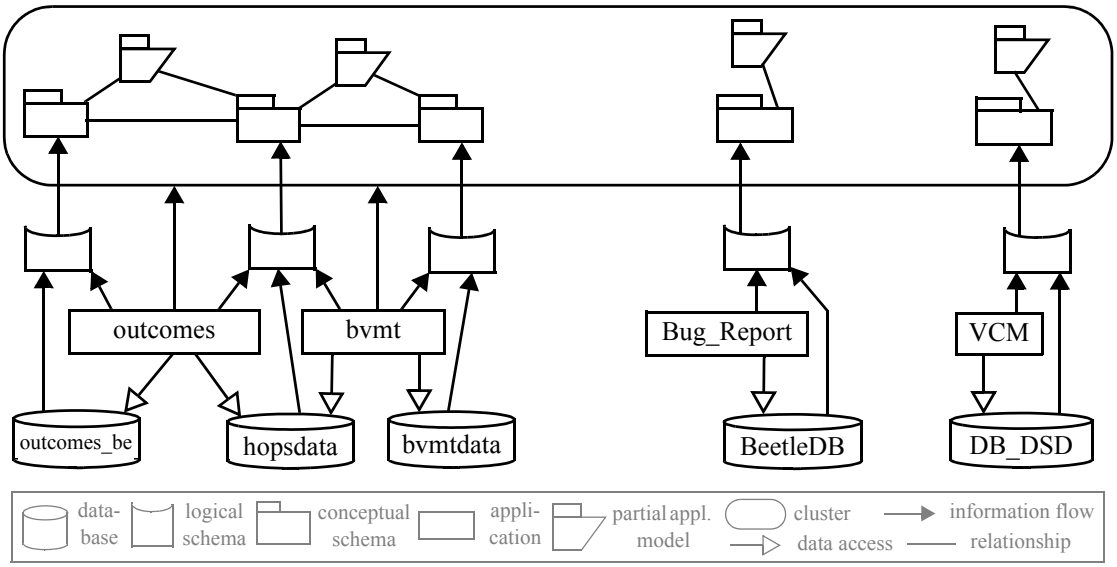


Figure 2.2: Case study understanding phase

Unified Modeling Language (UML) [BRJ99]. The resulting conceptual schemas are then refactored by the reengineer to improve the understandability. Since these steps stem from the VARLET approach, we skip the details here, and refer to Section 3.2 and [Jah99].

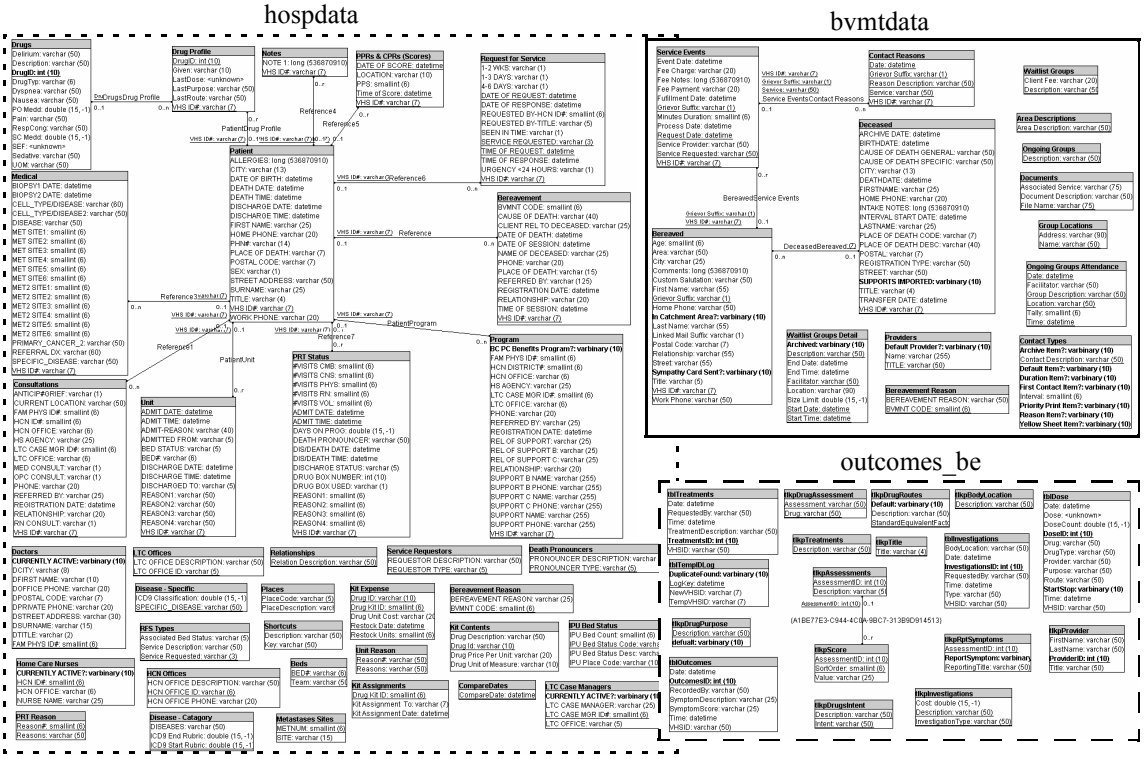


Figure 2.3: Parsed data models overview

The results of these first reverse engineering steps are independent refactored conceptual schemas. Since the VARLET approach supports an iterative explorative process that manages inconsistencies, in this phase the reengineer has two possibilities. The reengineer can either rapidly produce uncertain abstract results or analyse sequentially the (independent) schemas completely.

A crucial factor for understanding the web information system's data structure is the retrieval of the implicit inter-schema relationships. Figure 2.4 shows an example for a typical situation occurring in web information system evolution. Two tables Bereavement Reason exist in two different databases (hospdata and bvmtdata) storing the same information at the beginning of system's evolution. During maintenance activities only the table in the bvmtdata database is updated. The table in the hospdata database remains as inconsistent copy. Such relationships are valuable indicators for maintaining the system, i.e., for the reengineering responsible for the system extension and improvement. Other examples of inter-schema relationships are described in Section 3.3 and [WNGJ02].

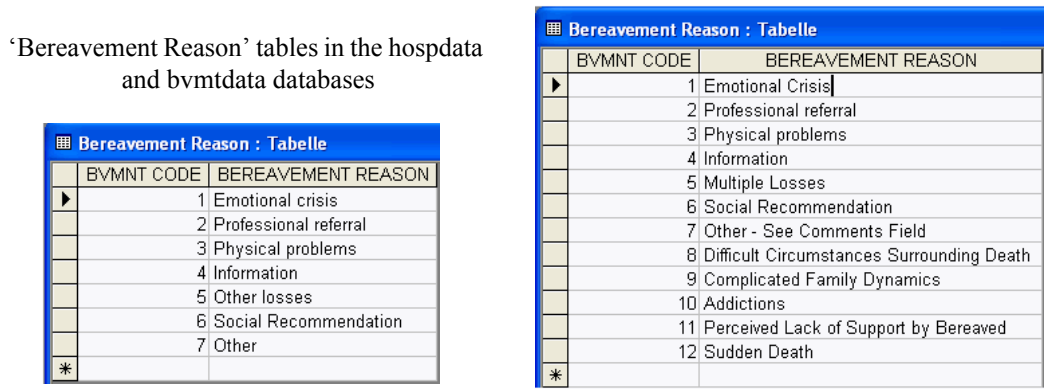


Figure 2.4: Table clone example

The detection of such inter-schema relationships is based on source code analysis with a pattern instance recognition mechanism. The first step is the identification of code fragments of interest with island grammar parsing [vDK99, Moo01] and slicing [Wei84]. These code fragments are then analysed with Niere's *incremental pattern instance recognition* approach [Nie03]. Patterns in source code that indicates (inter-schema) relationships are not certain. Niere's approach enables an iterative recovery process definition that can handle uncertainty and is based on (composed) pattern definition. Examples are given in Chapter 3. Here again the chosen approach enables an iterative explorative process that manages inconsistencies, which rapidly produces first results.

Figure 2.5 shows some inter-schema relationships of the three databases hospdata, bvmtdata and outcomes_be. The central database is the hospdata database which contains all Patient related data. The outcomes_be database stores results from examinations and finally the bvmtdata database includes information related to bereavement. Figure 2.5 shows (four relationships inside the hospdata schema and) four relationships between da-

tabase schemas. Firstly, we have the duplication between the two Bereavement Reason classes. Secondly, a duplication of the deathdate between Patient and Deceased is retrieved. Thirdly, a replication between DrugProfile and tblDose is detected, which maintains information about the last drug dose, purpose and route that was given to a patient. Fourthly, an inter-schema foreign key relates classes Patient and tblOutcomes.

Finally, the conceptual schemas can be refactored, compared to each other and views can be build from parts of the different schemas. One task from the adaptation phase can be used in the understanding phase: clustering. For further understanding clusters can be build [TWBK89] and used as views for exploring different system aspects.

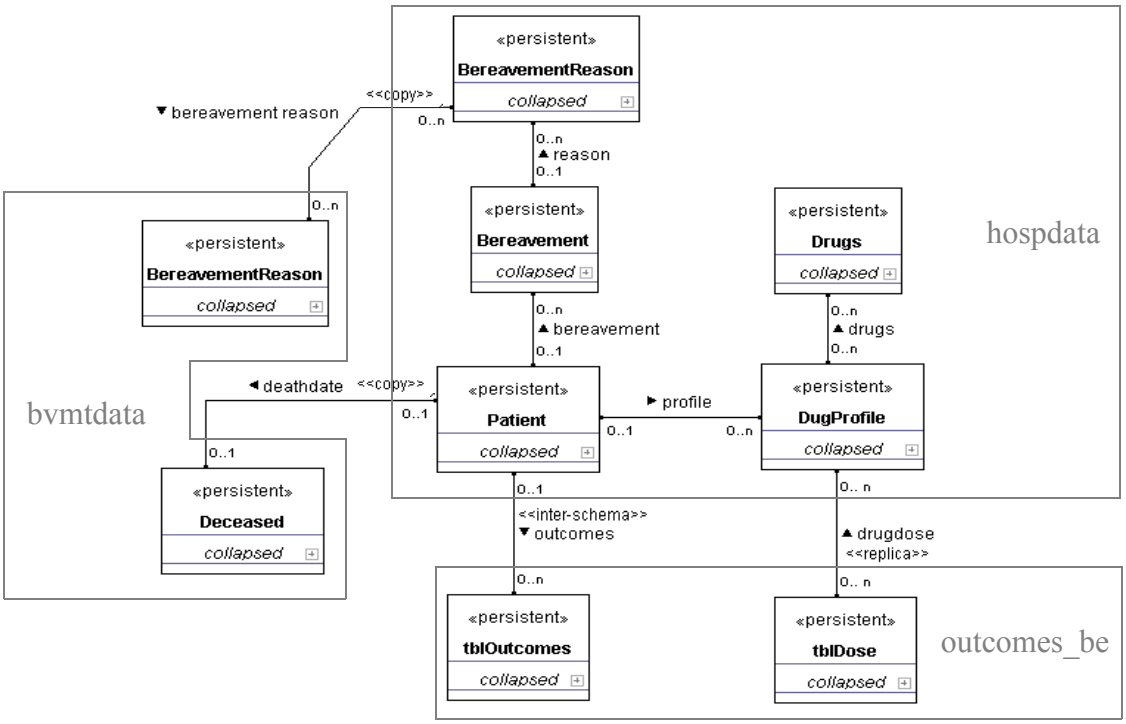


Figure 2.5: Palliative care conceptual schema excerpt in UML

2.2.2 Adapting Phase

After understanding major parts of the system, the reengineer needs support for extensions. Figure 2.6 gives an overview of the adapting phase. New system parts were constructed. The different old and new data component models are clustered, (re)structured, (re)designed and connected. Finally, new application are generated that accesses existing or new databases.

The first step in the adapting phase is clustering the web information system parts to data components. Out of the recovered conceptual schemas and their interrelationships, classes can be grouped according to their interrelationships. An example is the clustering pro-

posed by [SPPB02] that groups classes (representing entities) according to the primary keys of their corresponding entities, cf. Figure 2.7.

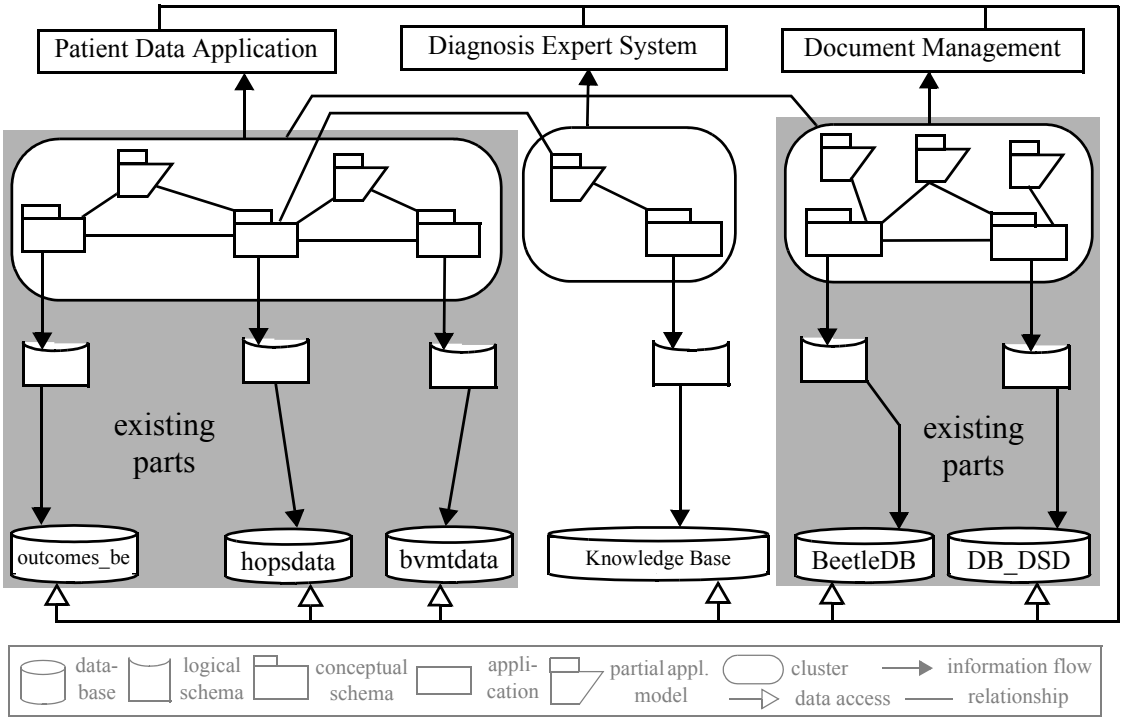


Figure 2.6: Case study adating phase

Further, in iteration with redesign, the clustering can support the reengineer in the conceptual partitioning of the data related parts based on relationships to application classes, i.e., construct external schemas. In this case study, an external schema to the Palliative Care Center Victoria databases was constructed that interfaces the Diagnosis Expert System prototype application. Moreover, clusters of classes to data components and clusters of data components are built and classified into corporate data components, data mediation components and ubiquitous data components to support the reengineer.

The next step is redesign. We propose four architectural patterns to facilitate data mediation between the different data components. The Data Portal pattern can be seen as a facade or interface to data components. The Data Fusion pattern and the Data Transducer pattern merges and transform exchanged data. The exchange itself is realised with the Data Connection pattern that cope with update, synchronisation and reachability problems of participating data components. To redesign the models themselves we provide a (design) pattern instantiation mechanism and a redesign transformation catalogue. Redesigned or newly designed data components are included in an existing data component cluster or form a new one.

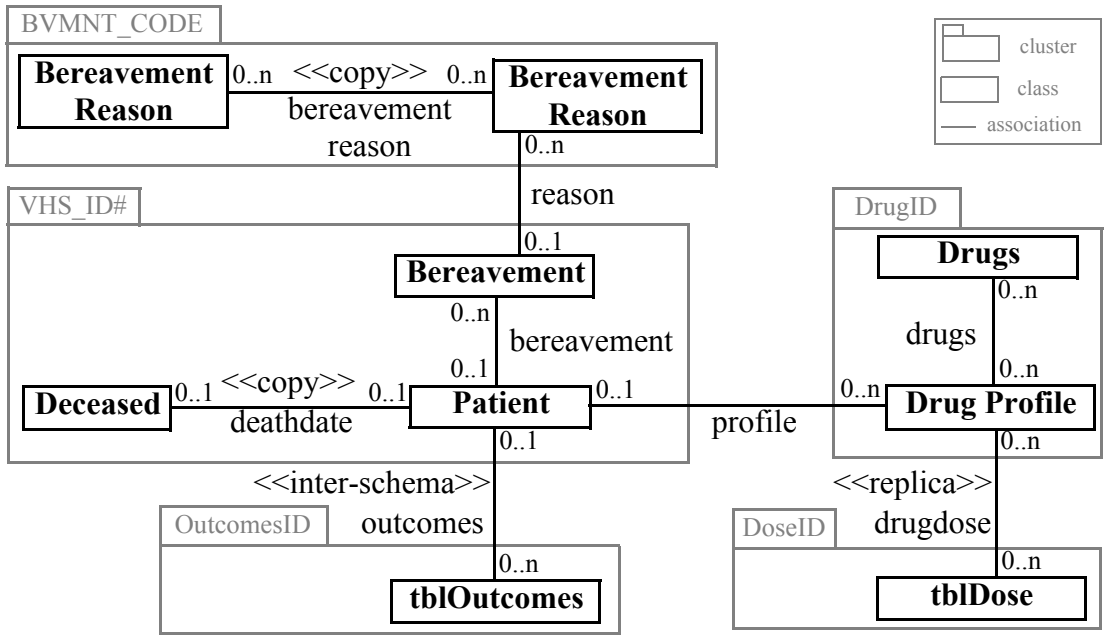


Figure 2.7: Clustering example

In this case study a prototype of a Medical Diagnosis Expert System was constructed. Currently the medical diagnosis expert system is a sample prototype. The prototype is composed of a Knowledge Base, the Expert System application and a user interface (Diagnosis GUI). These data components collaborate with the existing web information system of the Palliative Care Center Victoria. Figure 2.8 shows an overview of this extension.

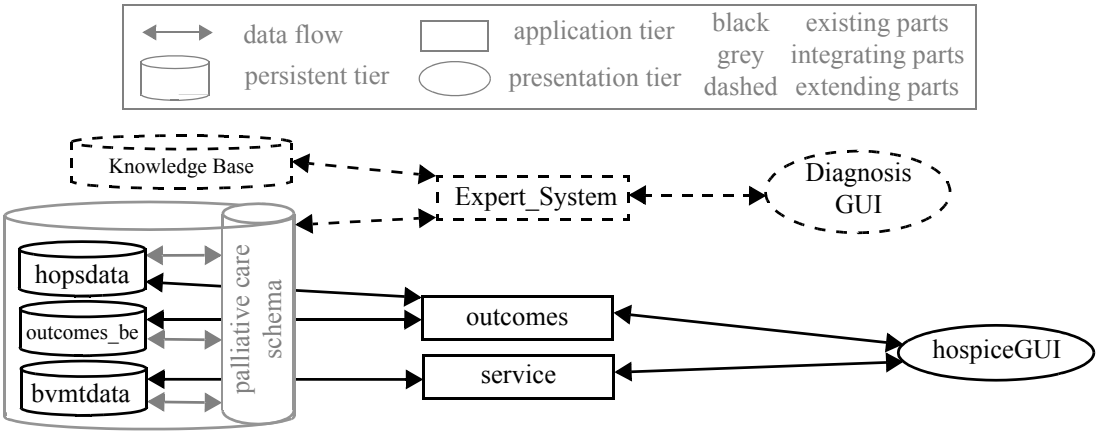


Figure 2.8: New data components integration examples

The model correspondences retrieved during the understanding phase and added during the adapting phase are used to generate an object-oriented access layer to the existing databases.

The adapting phase terminates with model execution facilities. Modelling and generating an entire application, using the chosen (middleware) technologies, is a hard task. The aim in this thesis for model execution is on the one hand to facilitate the modelling and generation of small data mediation components (web services) that exchange data in eXchangeable Markup Language (XML). On the other hand to facilitate prototyping based on the retrieved system models.

2.2.3 Model Maintenance

The data-oriented reverse engineering and the data component extension are not fully automated nor performed sequentially. On the contrary, they are part of an iterative process, i.e., reengineers explore the web information system in several iterations to understand and adapt it. Therefore the models have to be maintained. The data models and their interrelationships have to be consistent to preserve the consistency of the data. We developed the History Graph Mechanism that provides model consistency through explicit model transformation logging and selective undo for iterations, cf. Chapter 5. Figure 2.9 depicts that changes ( ) can then be traced through the models ( ).

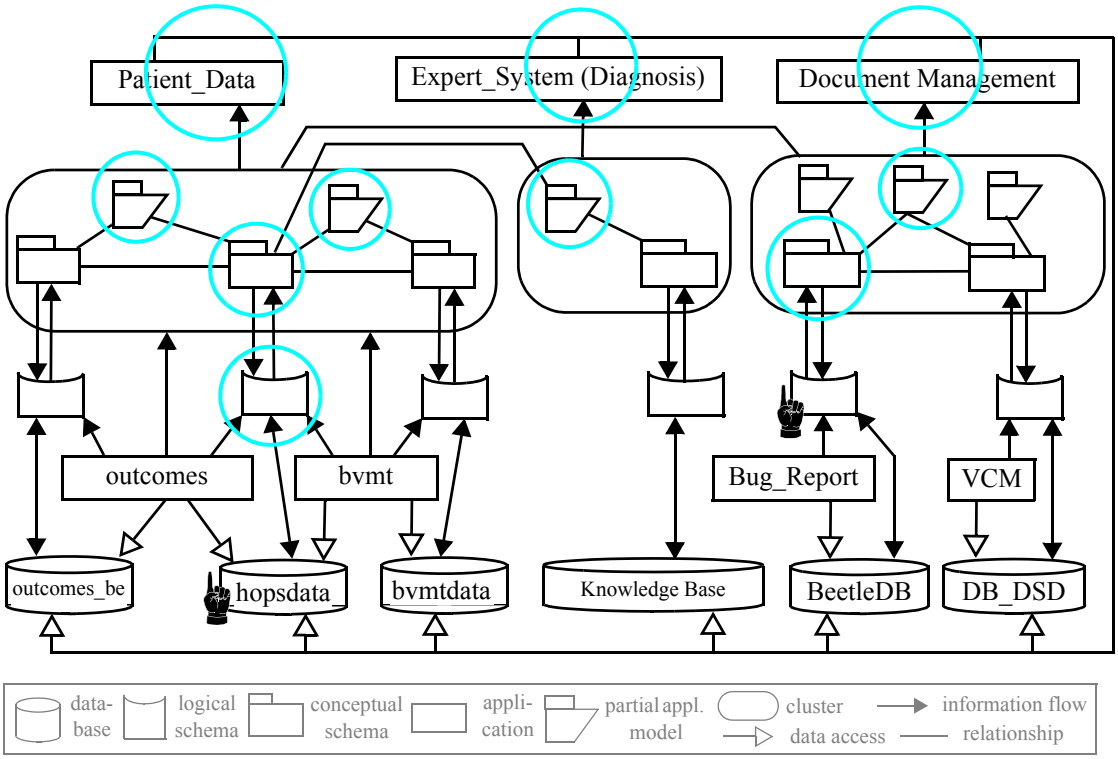


Figure 2.9: : Case study administrating phase

A sample is depicted in Figure 2.10. A duplication between class Deceased and class Patient is retrieved in a first understanding phase. The duplication is based on the attributes deathdate and death_date.

In a second understanding phase the reengineer detects variants in table 'Patient'. This has to be reflected in the conceptual model, i.e., class Patient becomes the superclass of new classes DischargedPatient and DeceasedPatient. Because attribute death_date has moved from class Patient to class DeceasedPatient, the duplication between class Deceased and class Patient is no longer valid. Instead it has to exist between class Deceased and class DeceasedPatient.

The History Graph Mechanism reestablish consistency by undoing previously applied model transformations. This means that the creation of the duplication between class Deceased and class Patient is undone. Other schema transformations, e.g., during clustering or redesign, are also affected from this change. Not all of them are necessarily undone; only those transformations that are not longer valid are undone. Finally, the reengineer can browse the list of undone schema transformations and may reapply some of them in a slightly different context like it is the case here for the duplication deathdate between class Deceased and class DeceasedPatient.

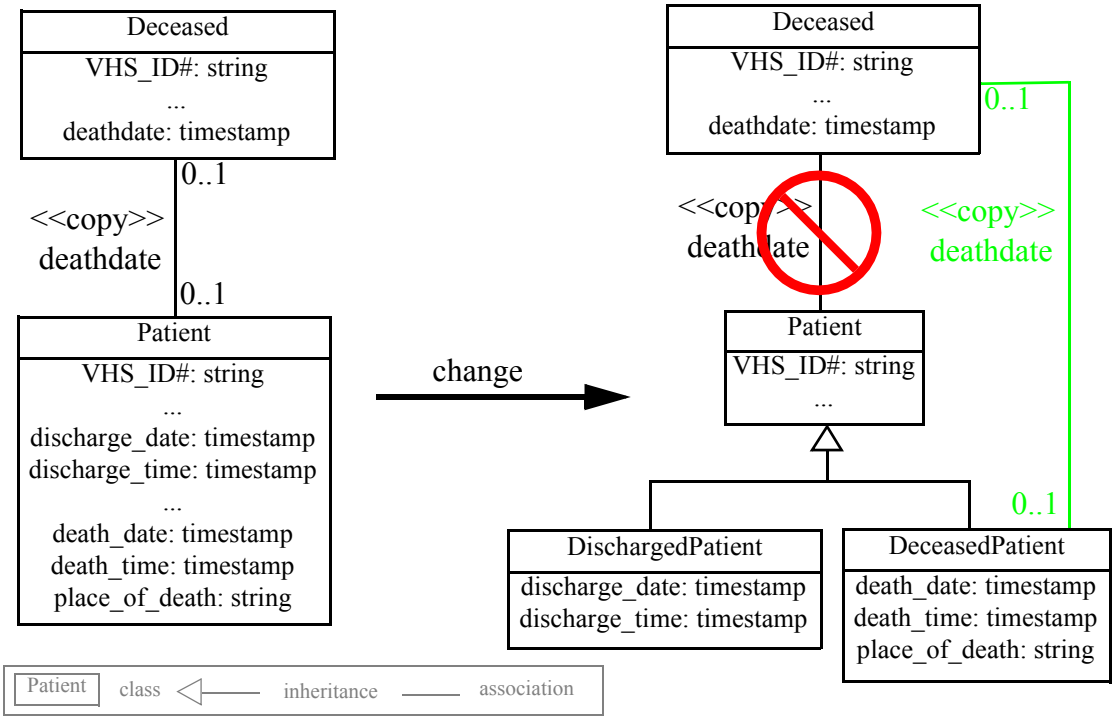


Figure 2.10: Consistency violation example

For the sake of completeness, we shortly report about the Information Management System that coordinates the management task regarding access rights and documentation. In the understanding phase the two systems DSD and BEETLE were reverse engineered. Since DSD and BEETLE are stand alone applications, the outcome of the understanding phase were two coexisting conceptual schemas that handle similar data. One problem was that they were based on two different databases. Data was hold twice without replication concept neither implementation. Further different client technologies was used and the data access was implemented with embedded Structured Query Language (SQL) [Dat89] statements.

In order to use this system as an information management system in Health Care following requirements had to be fulfilled. The document handling had to be extended to multimedia data, e.g., the parallel versionning of image files (radiology pictures) with corresponding audio files (explanations), the management of different movie files that belong together (sequences from an operation), etc. Further, the integration of the two database schemas was needed to achieve consistent document management. After schema integration, two options arise. Firstly, use the new virtual schema as external schema and stepwise migrate both applications to use it. Secondly unify the databases to one database and stepwise migrate both applications. We choose the second option because of technical reasons. The concrete resulting tasks (database integration, application migration and extension) are not further disscused in this thesis.

CHAPTER 3: DATA-ORIENTED REVERSE ENGINEERING

If data cannot be correctly understood, it cannot be combined with other information. Instead, it is just data pollution.

UNKNOWN

3.1 Reverse Engineering Data Components

Web information systems consist of large amount of heterogeneous data spread over multiple locations and platforms. These systems have to be understood for maintenance and especially for extension. As the persistent data structure is the central part of a legacy (web) information system [Aik96], our data-oriented reverse engineering approach focusses on the web information system parts related to the data, i.e., *data components*:

> A data component is a unit that encapsulates capabilities and functionalities to store and/or manage data.
>
> In this thesis a data component is mainly composed of one or more *models*, i.e., persistent data models (*schemas*) and/or application models, and relationships between the models. A data component can be composed (1) exclusively of schemas, (2) exclusively of application models or (3) of a combination of schemas and application models.
>
> Further, each data component has a defined interface to interact with other data components. Data components can be composed and exchange data via connections. A data component can be deployed independently. The basic idea of a data component is based on the component / connection philosophy presented by Szyperski [Szy99]. Note that the legacy data components may not fulfill all these characteristics entirely.

We distinguish between three kinds of data components:

- data components within corporate organisations (corporate data components),
- data components among corporate organisations (data mediation components) and
- data components between corporate information systems and mobile, embedded smart devices (ubiquitous data components).

The focus of our approach is the recovery and retrieval of the schemas from the distributed databases of legacy (web) information systems. This implies that along with the schemas the dependencies resulting from distribution have to be reverse engineered. The black

parts of Figure 3.1 illustrate this situation. By analysing the databases and the legacy applications, (1) the logical schemas are recovered and then (2) the conceptional schemas with inter-schema relationships and parts of application models are retrieved. Several approaches and tools exist for these reverse engineering subtasks. In this approach we use, combine and extend some of them. The light grey parts of Figure 3.1 can be ignored here.

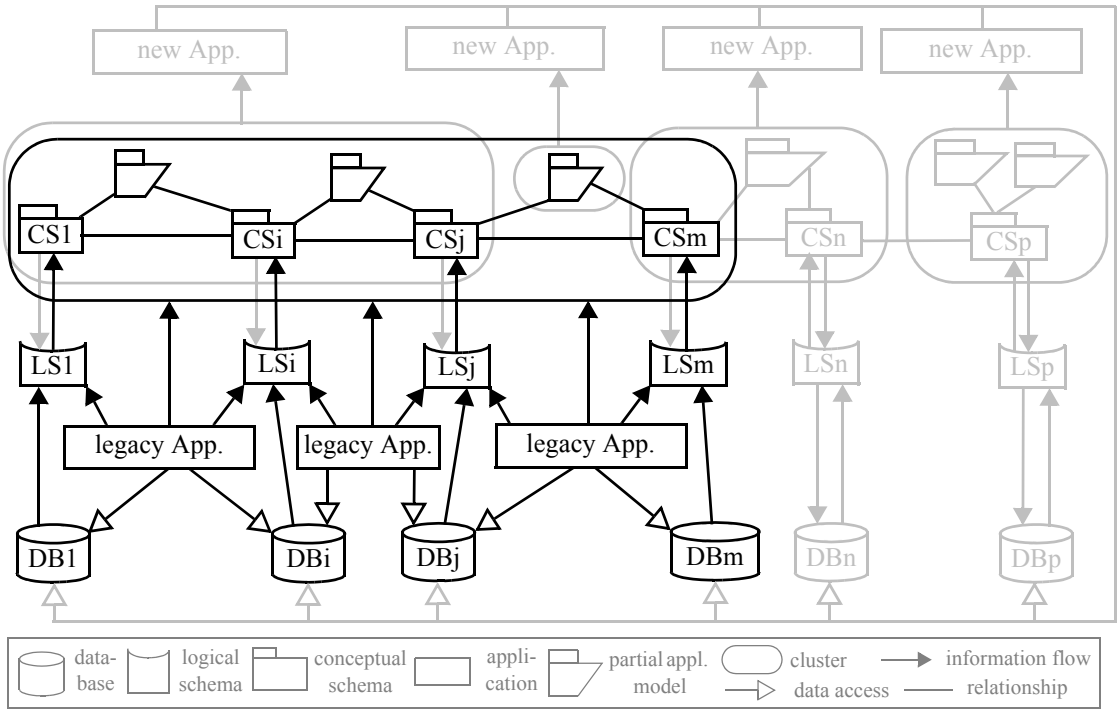


Figure 3.1: Web information system understanding

3.1.1 Reverse Engineering Steps

We divide the reverse engineering of web information systems into two steps. Firstly, the schema of each database is recovered and conceptualised. Secondly, the data dependencies between these schemas and the application model parts are retrieved.

Figure 3.2 gives an overview of these reverse engineering activities. Four data components are depicted but only two of them store data and consequently contains schemas. In the activity 1.1 Data Model Recovery[1] the physical schema is parsed and represented in an Extended Entity Relationship (EER) [Che76, BCN92] notation. This EER model represents the logical schema. It is (structurally) enriched and (sematically) completed during an iterative explorative analysis process involving reengineers and domain experts. Next, the logical schema is mapped to a conceptual schema. This initial object oriented conceptual schema in UML notation can then be refactored by model transformations (1.2 Mapping and Refactoring).

1. The Activity 1.1 Data Model Recovery is represented twice for layout reason.

Once the schemas are identified, enriched and completed, the definition of how they interoperate is needed. In the second process activity 2. Relationship Retrieval the relationships between the different data component models are investigated. This task also involves code reverse engineering and consequently uncertainty management. Again (only) a semi-automatic, iterative approach is feasible. The reengineer typically refactores the resulting conceptual schemas.

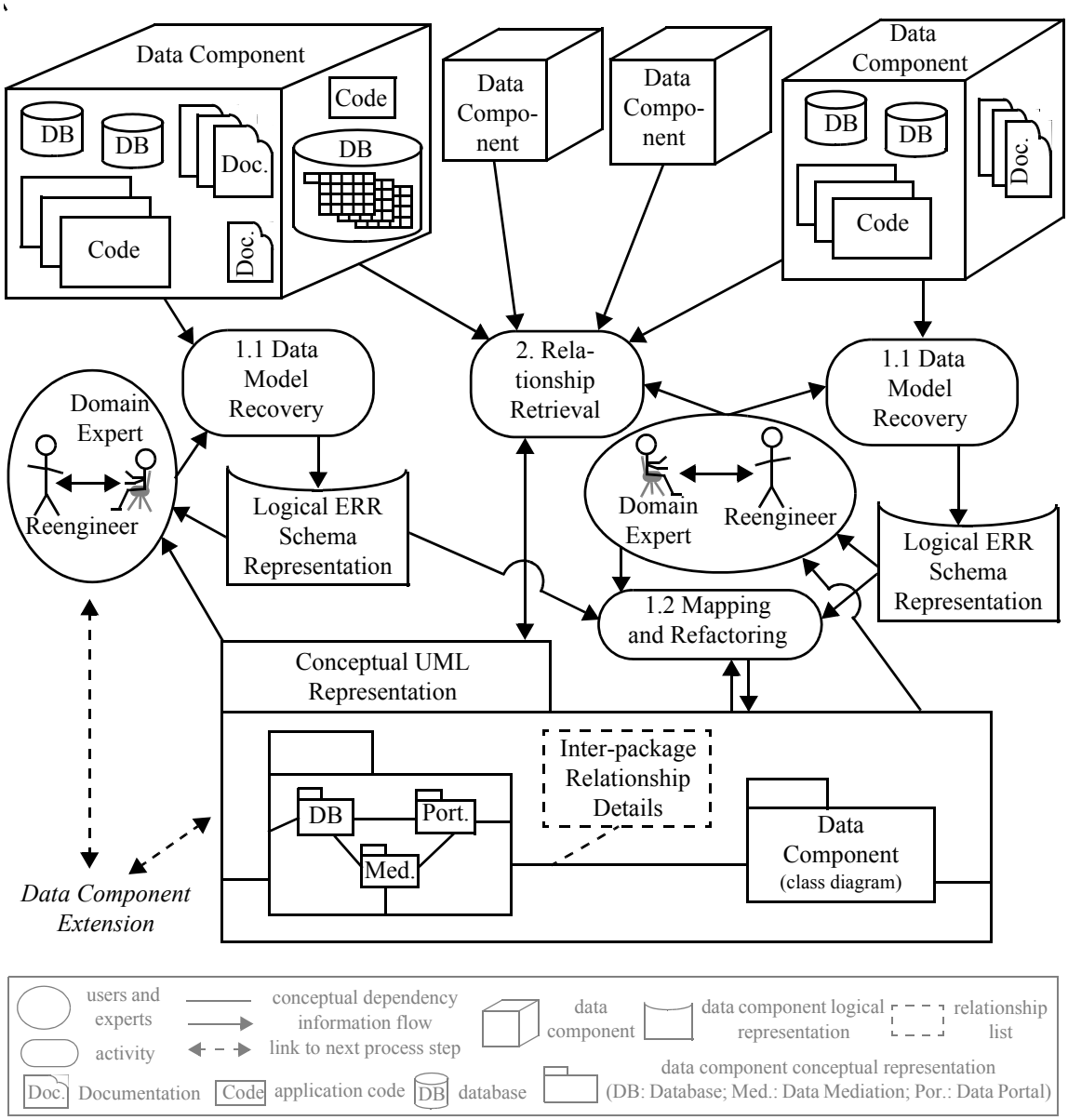


Figure 3.2: Data Component Reverse Engineering - Overview

From each activity, backward iterations to the precedent process activities are needed because of misinterpreted, incomplete or even erroneous intermediate results. Summarising, the two activities of the data-oriented reverse engineering process are done in iterations.

3.1.2 Relationships and Data Dependencies

Web information systems underlie frequent changes and integration. In contrast to non-distributed systems no overall structural representations are available and thus system understanding is rather difficult. To understand the system data components entities and relationships, the data models have to be reverse engineered. While recovering entities is relatively simple, relationship retrieval can become quite complex.

Often occurring relationships are foreign keys or attributes named like attributes of other entities ("immaterial foreign keys"). Complex functional dependencies are used when the relevant relationships are computed on-the-fly by combining the values of multiple stored attributes. While revealing the first case is relatively simple [Jah99], instances of the latter case are rather hard to detect. Moreover, assume a set of database entries which is spread over the different databases. They might depend on each other in various different ways and the intended coupling can have different semantic properties. Relevant relationships, however, will manifest themselves in the applications.

Relationships are dependencies between entities based on attribute relations. These relationships exist between attributes within an entity or between attributes of different entities. Further relationships exist between attributes of entities in different schemas. We distinguish between:

- intra-entity relationships:
  relationships between attributes within one entity.
- inter-entity relationships:
  relationships between attributes of different entities, that we divide in:
  - intra-schema relationships:
    relationships between attributes of different entities within one schema.
  - inter-schema relationships:
    relationships between attributes of different entities of different schema.

Figure 3.3 gives an overview of relationships that we consider, as a class diagram. Examples will be given in Section 3.3.2; others can be found in [WNGJ02].

A data component model is composed of Entities with Attributes and Relationships. A Relationship is a Homonym, a Primary Key, a Data Dependency, a Usage Relationship or a Join. A Data Dependency is an inclusion dependency (IND), a Redundancy dependency or a Constraint. Relationships classes are mostly based on Attribute indicators, i.e., relations between Attributes and attribute properties. Attribute properties (AttrProp) are:

- name similarity (NS)
  names of two attributes are syntactically close to each other

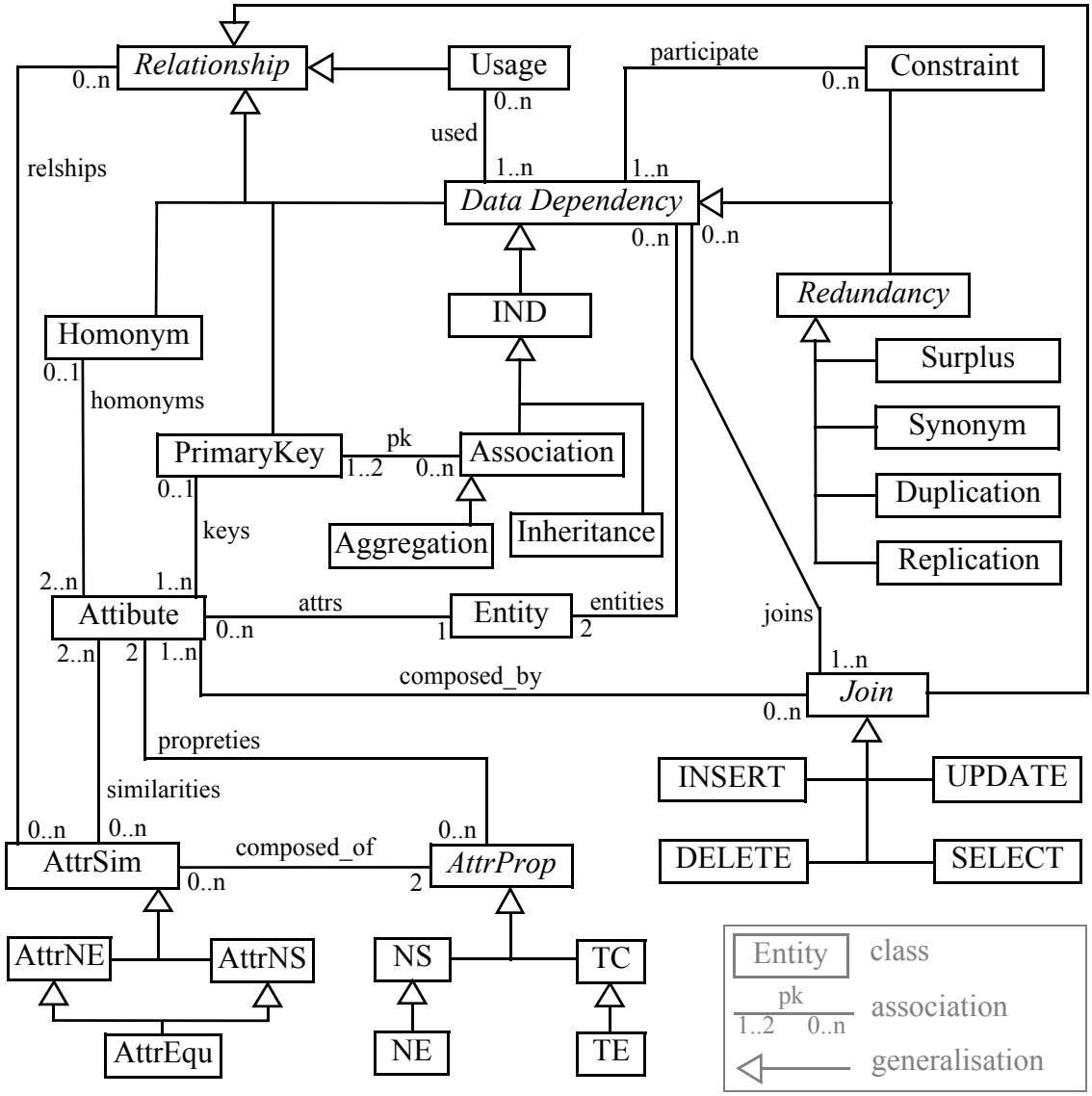


Figure 3.3: Relationships and Data Dependencies Overview

- name equivalence (NE)
  names of two attributes are equal
- type compatibility (TC)
  values of two attributes can be exchanged without converting the values, e.g., 'char' and 'varchar(18)', 'varchar(30)' and 'varchar(50)' or 'numeric' and 'double'.
- type equivalence (TE)
  types of two attributes are equal

We combine these attribute properties to four different indicators for attributes:

- attribute similarity (AttrSim=NS&TC),
- attribute name similarity (AttrNS=NS&TE),
- attribute name equivalence (AttrNE=NE&TC) and
- attribute equivalence (AttrEqu=NE&TE).

Note that the attribute indicator classes follow an inheritance hierarchy, e.g., if two attributes are equivalent they are also similar. This also holds for other classes involved in inheritance hierarchies, cf. Figure 3.3.

Primary Key

The sole pure intra-entity relationship we consider here is the PrimaryKey. A primary key is one attribute or a set of attributes that enables the unambiguous identification of an entity. This concept stems from the relational database model [Cod70].

Homonym

A relationship where two attributes of two different entities have the same name but store different information is a Homonym. These are attributes that are not related through the application code but solely by name equivalence. Homonyms help to avoid confusion about equivalent named attributes during the understanding process. Note that omitted relationship update during integration may lead to false homonyms. Indeed, attributes with the same name that store the same information but without explicit relationship in the application code can be recovered as homonyms. Checking the data can help the reengineer to detect such false homonyms.

Joins

The basis for all data dependencies are relationships between attributes, i.e., Join operations (joins). Following SQL we consider four types of joins: INSERT, DELETE, UPDATE and SELECT. Attributes in access code (not only SQL statements) that implements such a join operation will be assigned as taking part in the operation, i.e., joins are composed_by attributes.

Data Dependencies

Data dependencies are relations between entities, more precisely between their attributes. We classify them into three basic kinds:

- redundancy dependency:
  the same information is held - and maintained - (at least) twice
- inclusion dependency:
  an attribute (set of attributes) in one entity holds a part or the same information as an attribute (set of attributes) of a second entity
- constraint dependency:
  condition(s) over two or more data dependencies to assign information

Redundancy Dependency

The first inter-schema Data Dependency type is called Redundancy dependency. A redundancy dependency can be Synonym, Surplus, Duplication or Replication. In our approach redundancy dependencies must be related within the application code.

Synonym

Synonyms are attributes that hold the same information but have different names. Non name similar, type compatible attributes where information is inserted and updated together are synonym candidates. Exclusively an insert operation or exclusively an update operation is not sufficient to indicate a synonym. Therefore, synonyms imply joins (at least one insert and one update), type compatibility but no name similarity (includes name equivalence).

In some cases data is never updated because the information system uses keys that are never altered. For this case further more complex investigation has to take place which is based on attribute semantic resemblance. Semantic resemblance and in consequence semantic equivalence (synonym) are hard to determine and subject of current work.

Surplus

We have "real" redundancy if the same information is maintained but not inserted together. In addition to attribute similarity, a Surplus dependency is revealed when the attributes are updated but not inserted together (at least one update but no insert operation).

Duplication

We talk about Duplication if an explicit copy is made (at specific points in the application) but the copied information is not kept consistent. Thus, attribute similarity with at least one insert operation but no update operation indicates a duplication. Note that the insertion do not have to take place at the same time, i.e., the insert operations may be spread over the code. Such duplications are hard to detect.

Replication

Replication is an explicit copy, which is held consistent, i.e., a controlled redundancy. Thus, it is the occurrence of attribute similarity and at least one insert and one update operation. In general these operations will be performed together, but, in analogy with the duplication, the operations can be distributed in the code.

IND (Inclusion Dependency)

Inclusion dependencies (INDs) are known from (single) relational databases [EN94]. We identify an IND as a data dependency where the attributes are similar and at least one select operation exists. They build the basis for interpreting the semantics of foreign keys

(associations and inheritances). We use the classical definition of the inclusion dependency in data reengineering. We classify INDs into R-INDs, C-INDs or I-INDs according to [FV95]. An R-IND describes a relationship which is not separated by a separate entity (relation schema). An C-IND describes a cardinality constraint and an I-IND describes an is-a relationship (inheritance).

Association (R-IND, C-IND)

An Association is an inclusion dependency, i.e., an R-IND with an C-IND as option. An R-IND is an IND where the included attribute (set of attributes) is a PrimaryKey of the corresponding data (Entity). In contrast to foreign keys, an association can have a n:m cardinality. This corresponds to a reference table and a 1:n foreign key and a 1:m foreign key. In case of EER diagrams, which represent databases, we only have inclusion dependencies. The cardinality of an association is determined by the R-IND and the C-IND if it is classified as one.

Aggregation

An Aggregation is a special kind of association which represents a „has_a“ relationship. In contrast to an association, an aggregation represents a „whole / part“ relationship model [BRJ99].

Inheritance (I-IND)

Inheritance is a generalisation.

> „A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called subclass or child).“
>
> [BRJ99]

In terms of EER diagrams it is an IND classified as I-IND.

Constraint

Dependencies that relate data (attributes) in more complex manner are classified as Constraint dependencies. A constraint incorporates parts of the application model, i.e., business logic. Therefore an exact definition as well as a sub-classification of constraint dependencies are hard to give. The notion of a constraint can be refined when used in further analysis and re-design steps to identify the essential business rules and the business logic in systems that affect multiple databases.

Usage Relationship

In addition to the data dependencies it is useful for the reengineer to understand relationships between the persistent and transient parts of the web information system. We classify this kind of relationship as Usage relationship. A usage relationship is the usage of

data by transient parts of the web information system, i.e., an interface using the data dependencies for the connection to the "transient world", e.g., the application or the internet.

3.1.3 Model Representation

Data storage is mainly done in tables, records or files. Data management uses multiple data types which are assigned with values from the data storage structures. To represent these aspects we consider two models, namely the EER model and UML diagrams as object-oriented model.

Internally we use an object-oriented abstract syntax graph to represent the data component models (EER and UML) as well as the data manipulating code (application models). We refer to [Zün01] and [Nie03] for details.

We use EER diagrams to represent data model's data structures at the logical view level. A logical schema is composed of entities with attributes and relationships. The attributes have assigned data types which allows the distinction of the type of data stored. Attributes are used for identification (primary key) and search (indexes). The different relationships are represented with foreign keys or integrity constraints.

UML Class diagram

The UML enables the conceptual representation of data management aspects independently from the physical data storage kind. Instead of entities we have classes with attributes. Relationships are associations, aggregations, generalisations and constraints. Moreover, at the conceptual level, we consider data dependencies that occur with the distribution of the data component models. Of course also data types are used although they are different from the physical ones.

The reasons to introduce a conceptual representation are on the one hand the higher level of abstraction for understanding and on the other hand the representation of behaviour. Data management, i.e., data access and manipulation, is done in methods, procedures or functions which can be represented with UML.

As mentioned we represent an entity in the conceptual schema as a class. We differentiate the entities, i.e., persistent classes, from transient classes that do not represent persistent data structures. We mark a persistent class with the stereotype <<persistent>>. Attributes are represented with their respective types according to the type mapping.

We represent all relationships of Figure 3.3 except the attribute similarity (including subclasses), INDs[1] and Joins. They are schema annotations from the code that are used to recover data dependencies. Beside the fact that representing all these three kind of

1. Note that INDs are represented in the logical schema (EER diagram) but only refined INDs, i.e., R-INDs, C-INDs and I-INDs, are mapped to the conceptual schema (UML class diagram).

relationships will overload and confuse the schema representation, their information is enclosed in the represented data dependencies.

Relationships are represented with UML associations except primary key, inheritance and constraint. We conserve the primary key information, when existing, to enable mappings from classes (conceptual schema) to entities (logical schema). Attributes belonging to a primary key are underlined. Inheritance is represented by UML generalisation. A constraint is represented within an additional class with stereotype <<constraint>> containing the constraint itself. The relationships to the (attributes of the) classes involved in the constraint are represented as aggregations, cf. Figure 3.4.

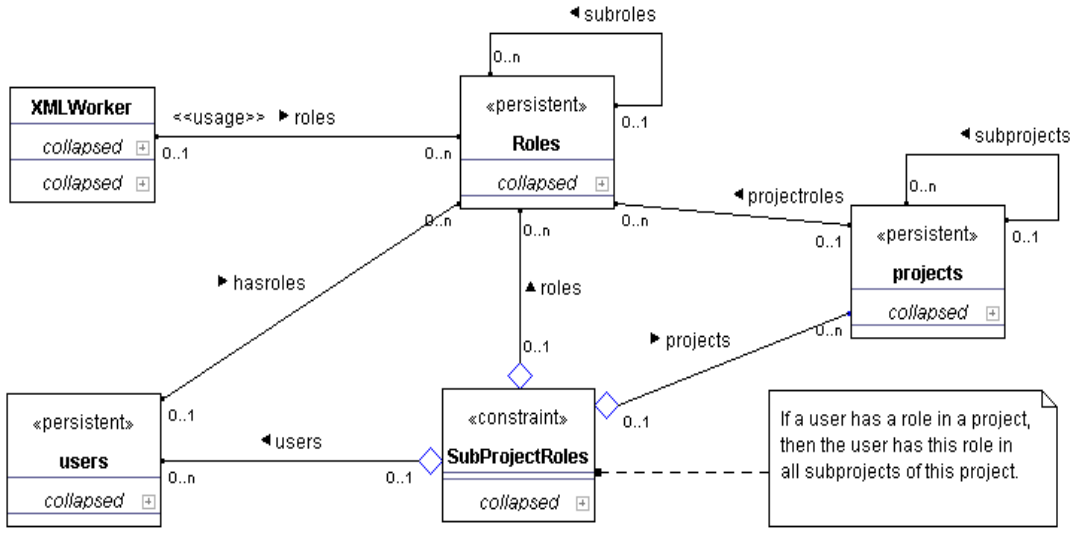


Figure 3.4: Conceptual schema view (relationships)

All relationships represented as associations and are always assigned a stereotype, except the association itself:

- Homonym <<homonym>>
- Synonym <<synonym>>
- Surplus <<surplus>>
- Duplication <<copy>>
- Replication <<replica>>
- Association none or <<inter-schema>>
- Usage Relationship <<usage>>

The abstract relationships used for the relationship retrieval are of course not assigned a stereotype because they cannot be instantiated. To further improve the understanding we distinguish between intra-schema and inter-schema associations. We add a stereotype <<inter-schema>> for inter-schema associations, cf. Figure 3.39.

UML Package diagram

We represent the logical schema with EER diagrams, the conceptual schemas with UML class diagrams, but how can we represent architectural aspects like distribution, redundancy or views? Since the architectural aspects are considered based on the conceptual schemas, using UML for representation purposes seems to be the most appropriate notation.

Schemas are represented by package diagrams, i.e., the entities, attributes and relationships are represented in class diagrams which are enclosed in a package. The relationships between the entities of different packages are grouped and represented as inter-package relationship. Further, we employ the following UML notation elements:

- stereotype, to differentiate packages (and classes and relationships)
- note, to show comments or annotations.

We further use packages to visualise packages of data manipulating classes, e.g., constraints, web interfaces or portals. Representation problems occur when a class is contained in more than one package that is to visualise. Such we restrict the representation of inter-connected packages in package diagrams to packages that do not have overlapping classes. If views are build such that a class can participate in more that one view, we recommend to use view diagrams instead of package diagrams.

Nevertheless, inter-connected packages are useful to represent certain kind of views on the system. Of course the representation of the physical aspects (how the system is physically distributed) is one of these views. This corresponds to the conceptual schemas and the manipulating data classes as they are stored and organised in the databases and files, cf. Figure 3.5. Another example of a system view divided in disjunctive connected packages are representations resulting from clustering. A clustering criterion may be to group the classes related by inheritances and redundancy relationships. For such purposes packages that contain packages, as long as the packages remain disjunctive, can be visualised.

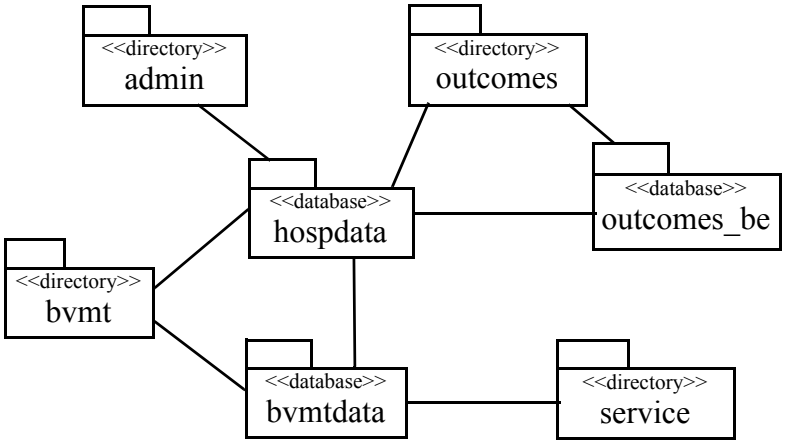


Figure 3.5: Pysical aspect view as package diagram

In case that view diagrams are judged not to be appropriate, a set of overlapping views on the system can also be organised in a package diagram. Overlapping packages can be visualised together without any connection between them. Such a diagram may be the result of manual analysis or can be the result of the automatic application of a view builder. Then the packages will be represented next to each other without any (visible) connection. Only the package name will guide the reengineer. Again, packages can be organised in subpackages if desired.

View diagram

A view diagram is a sub-diagram of a class diagram where only selected classes and relationships are visualised. The idea of views is not new and well known in the database field [EN94, Dat00]. Database (logical) views are directly mapped to conceptual schema views, cf. Figure 3.11. Note that a materialised view correspond more to a logical schema than to a database view since they are redundant subschemas and should be handled accordingly.

Further (conceptual) schema views can be defined by the reengineer. Schema views are representations of parts of schemas. That these parts are from a single schema is not mandatory. Since schema views represent aspects from the whole system, they are not fixed because changes of the underlying schemas should be reflected in the views. The reengineer can use predefined filters or define filters which dynamically calculate the schema views.

A filter is defined through the context of a class (schema entity) or a set of classes. The context of a class is the set of reachable classes from this class via relationships. The example of Figure 3.6 shows the 1-context relative to the inheritance relationships of class Patient. Operations can be either applied to schema elements in the corresponding under-

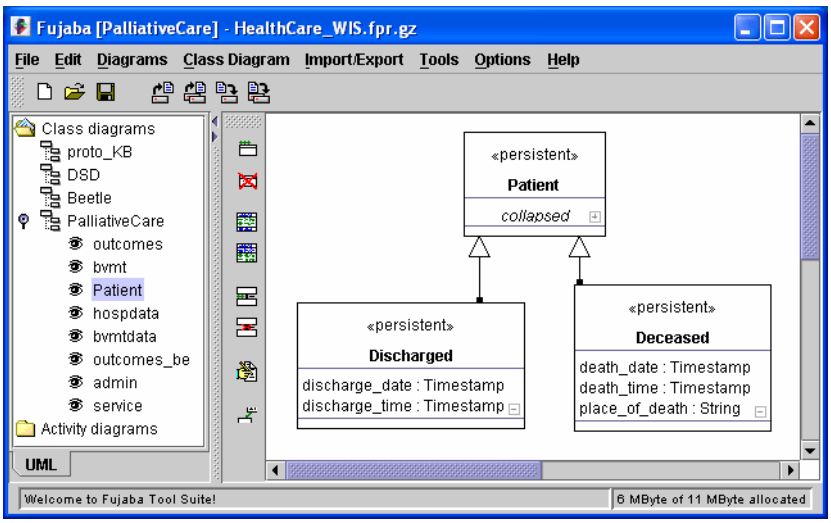


Figure 3.6: Conceptual schema view example

lying schema or in one of the schema views. Note that also data manipulating (transient) classes can be part of the views. Further details concerning view and filter definition can be found in [Rec01].

3.2 Data Model Recovery

Recovering the data structures of the web information model is the basis of the data-oriented reengineering process. We enhance the understanding of the data models by providing a conceptual knowledge-centric view on the models. We use annotations on the data models to represent the automatically and manually recovered knowledge. A fundamental aspect is that data model recovery is embedded in the iterative reverse engineering process.

The data model recovery is divided into two main activities: recovering the physical schema into a logical schema and constructing a conceptual schema. In the first activity the explicit data model elements can be parsed but hidden schema parts have to be recovered, i.e., data model elements that are implicitly enclosed in the schemas. The construction of the conceptual schema is done be an automated ad-hoc mapping followed by a manual refactoring phase.

3.2.1 Schema Recovery

The data of the data repositories participating in a web information system is organised in schemas. A data model can be represented by different schemas at different levels of abstraction. The common schema levels are the physical schema, the logical schema and the conceptual schema. We assume that the data model exists as physical schema. Such we recover a logical schema (before transforming it to a conceptual schema).

The first step is to recover the explicit data structures contained in the physical schema. We achieve this by parsing the according physical schema. A physical schema can be parsed through JDBC[1], or an XMLSchema[2] or a Document Type Definition (DTD) parser. JDBC drivers are provided for almost all data repositories. Note that we are not restricted to Java applications, we only use java as programming language to recover the schemas.

The easiest schema parts to recover are entities and attributes. They can be read (parsed) directly from the database. Depending on the database management system logical properties like primary keys can also be parsed. However, in most cases not all schema element properties are explicitly defined.

The most valuable schema elements for understanding are the dependencies between entities, namely foreign keys, inclusion dependencies, links and references. Again some of

1. The JDBC API provides Java applications with access to all major database systems and file formats, via SQL. According to Sun, JDBC is not an acronym for Java Database Connectivity.
2. http://www.w3.org/XML/Schema

them may be explicitly encoded in the database schemas, but most of them are hidden in the application code.

In case of a relational database additional information may enrich the schema knowledge, e.g., indices, stored procedures or (materialised) views. Further in federated or multi-databases the different schemas (local, component, export) may also be a valuable source of information.

3.2.2 Retrieval of Hidden Schema Parts

As mentioned above parts of the schema are hidden and cannot be parsed directly from the data repository. The most hidden parts are also the most valuable for the reengineers understanding. There are two main reasons why schema parts are hidden, i.e., not declared explicitly. Firstly, the database system does not have the language constructs to express all schema parts in an appropriate way. Secondly, like the lack of correct and complete documentation, the schemas are not attached with all the possible and desirable descriptions for many different reasons.

Why do we need this hidden schema parts? Well, basically because they enrich the data schemas with valuable reengineering knowledge. We distinguish between two ways for retrieving hidden schema parts: (1) a semi-automatic process where indicators of missing schema elements are retrieved automatically, before the reengineer validates or refutes them; (2) a manual process where the reengineer annotates the schema with knowledge that he derives from information sources like interviews, documentation, code reviews, tests or application executions.

To enrich the schemas with knowledge, operations to create schema annotations are needed, e.g., createPrimaryKey. Indeed, assume a user has gained knowledge of a certain schema for example by using a data retrieval tool. He afterwards wants to transfer this knowledge into that schema so that other reengineers can benefit from this regained knowledge.

These schema annotation operations are also used by the semi-automatic process. The annotations are endowed with a fuzzy value between 0 and 100 to express the certainty of the according annotation. Depending on his certitude the reengineer can also add uncertain knowledge to the schema. In case of the automated recovery the certainty degree depends on indicator significance that is predefined. The underlying logical model is presented in [Jah99].

Operations are needed for each schema element. Three different operations per schema element exist: create an element, remove an element and update an element. The create operation is always attached a fuzzy value. The delete operations can be bundled to a general deleteElements operation. Updates can be subdivided into several operations, e.g., updateAttribute can be subdivided into renameAttribute, changeType and changeFuzzyValue.

The basic element types for a schema are entity, attribute, primary key and foreign key. Primary keys can be tagged as candidate keys. Foreign keys can be tagged as IND and classified as R-IND, I-IND or C-IND, cf. page 43.

Special elements resulting from reverse engineering practice or database features are variant, index, view, stored procedure, optimization structure and materialised view. A variant of an entity is a sub-categorisation of the entity into different logical entities, e.g., the entity patient has attributes for death date & time and discharge date & time. We have two variants for patient here; a patient can only have a death date & time or a discharge date & time. The other elements are well known from the database field and such we refer to the corresponding literature.

Automatic schema recovery covers all elements. Entities and attributes are easy to recover through JDBC. Relationship (primary and foreign keys) retrieval will be discussed in Section 3.3. Indices, views, stored procedures and materialised views need special parsing depending on the database management system. Variants and optimization structures are more likely to be retrieved.

Figure 3.7 shows three tables and an IND as an excerpt of a database schema from the palliative care information system. This example is depicted as it was parsed with the JDBC, i.e., no analysis was performed nor annotations added. The table Patient contains three variants and table Medical contains optimisation structures. Note that the foreign key Reference3 is automatically retrieved through the JDBC based parser.

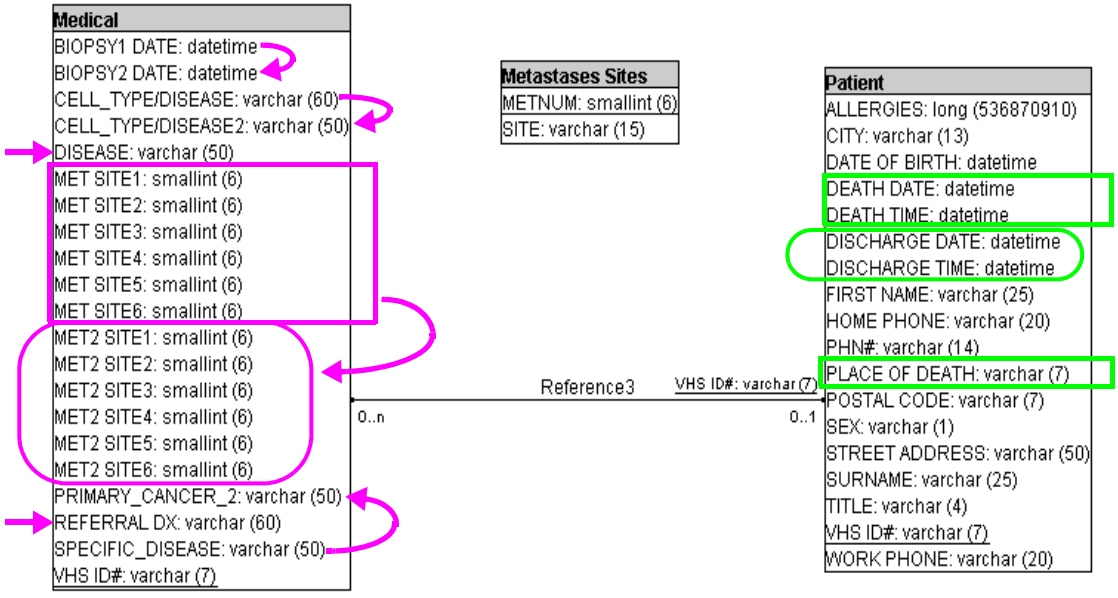


Figure 3.7: Variant and optimisation structure example

Variants can be found in code or by analysing the data. Figure 3.8 shows a simplified code fragment, sketches the three variants as three sample tuples and depicts a possible conceptual representation. Tuples belonging to variant 1 (patient is discharged) have NULL values for the three attributes related to the death of a patient. Variant 2 (patient is deceased) has the NULL values for the discharged date and time. If the patient is still in treatment (variant 3) all five attributes concerning the death or discharging of a patient have NULL values. Conceptually this results in three classes Patient, Discharged and Deceased depending the status of the patient. The code fragment outlines a status request for a patient of a given patientID. Note that this code fragment is a sample, still even more complex fragments can be detected and thus serve as indicators. For further details concerning variants we refer to [Jah99].

Several optimisation structures occur in the table Medical of Figure 3.7. Attributes REFERRAL DX and DISEASE stores the patient's referral reason and disease. The other attributes store two biopsies where BIOPSY1 DATE corresponds BIOPSY2 DATE, SPECIFIC_DISEASE corresponds PRIMARY_CANCER_2, etc. Each biopsy has six attributes to store metastases sites. Figure 3.9 shows a code sample as indicator for the optimised 1:n foreign key between Metastases Sites to Medical. Patients are searched where metastases where detected at SITE :site. Therefore all attributes of both biopsies has to be compared to the corresponding METNUM.

variant	DEATH DATE	DEATH TIME	PLACE OF DEATH	DISCHARGE DATE	DISCHARGE TIME	VHS_ID#	...
1	NULL	NULL	NULL				
2				NULL	NULL		
3	NULL	NULL	NULL	NULL	NULL		

```
...
date = „select DEATH DATE, DEATH
TIME from Patient where
VHS_ID# = :patientID";
status = „deceased";
if (NULL == date) {
date = „select DISCHARGE DATE,
DISCHARGE TIME from Patient
where VHS_ID# = :patientID";
status = „discharged"; }
if (NULL == date) {
status = „in treatment"; }
...
```

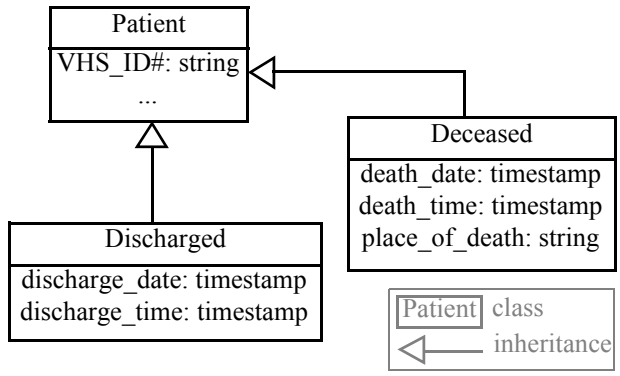


Figure 3.8: Patient variant example

The table Medical would correspond to three tables if no optimisation were introduced: a table „Disease" to store the patient's referral reason and disease; a table „Biopsy" storing the different biopsies for a patient; and a table „Biopsy_Metastases_Sites" for the detected metastases sites during a biopsy. This last table would have the 1:n foreign key to Metastases Sites. For further details concerning optimisation structures we refer to [Bew98].

```
...
M = „select METNUM from METASTASES SITES where SITE = :site";
...
patientID = „select VHS_ID from MEDICAL where
                                    (MET SITE1 = :M or MET SITE2 = :M or
                                    MET SITE3 = :M or MET SITE4 = :M or
                                    MET SITE5 = :M or MET SITE6 = :M)
                                    or
                                    (MET2 SITE1 = :M or MET2 SITE2 = :M or
                                    MET2 SITE3 = :M or MET2 SITE4 = :M or
                                    MET2 SITE5 = :M or MET2 SITE6 = :M) ";
...
```

Figure 3.9: Optimisation structure example

3.2.3 Schema Mapping

Once the logical schema is recovered, at least judged as recovered by the reengineer at this point in time, it can be mapped into a conceptual schema. The conceptual schema provides a higher level of abstraction, which makes it easier to understand the major points of interest. The conceptual representation is done with UML class diagrams. Such a mapping between logical schemas and the conceptual schemas is needed. The schema mapping is described in detail in [Jah99], we will only present the relevant parts here.

This meta-model mapping is done with triple-graph-grammars [Sch94, Lef95, SL96]. The fundamental idea of triple-graph-grammars is that two documents are integrated and such kept consistent through a third document, the integration document. Figure 3.10 depicts an example for document integration with triple-graph-grammars. A triple-graph-grammar rule relates elements of document A to elements of the integration document and those elements of the integration document to elements of document B. In Figure 3.10 this can be seen for a<---i--->b or (a1,a2)<---i1--->b1.

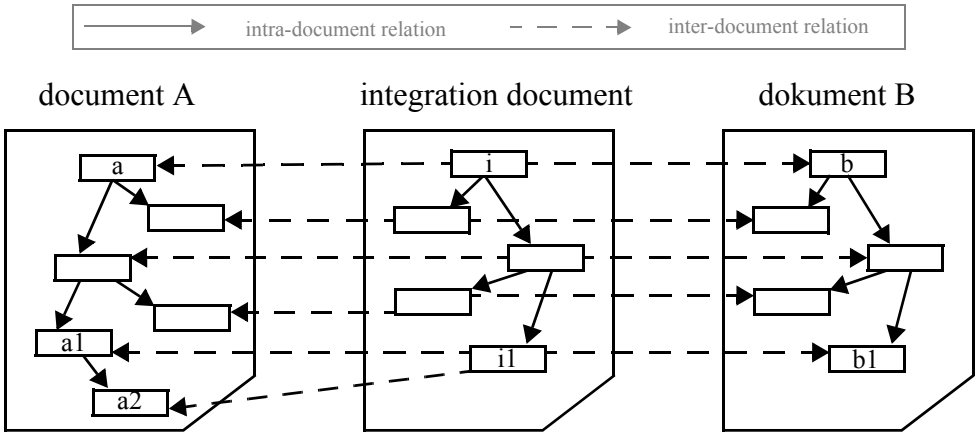


Figure 3.10: Triple-graph-grammar fundamental idea example

From a triple-graph-grammar rule, three rules are generated. These rules can be applied to the documents without any rule ordering or prioritisation. Firstly, the forward rule maps elements of the left document to elements of the right document, e.g., a of document A exist and i and b are created. Secondly, the reverse rule maps elements of the right document to elements of the left document, e.g., b of document B exist and i and a are created. Thirdly, the relating rule which relates elements of the left document and elements of the right document, e.g., a of document A and b of document B exist and i is created. If none of these three rules can be applied to an element of document A or B, this element is removed.

The gererated rules are *graph productions*. Generally, graph productions are defined as a pair of graphs, a set of application conditions, and a set of attribute-transfer-clauses. The two graphs are called the left-hand side and the right-hand side of the production, respectively. Graph elements of the left-hand side that also appear on the right-hand side are called the *interface graph*, i.e., they are neither created nor deleted. Graph productions and their application semantics have been formalised based on algebra theory, for a complete formalisation of these concepts we refer to [Roz99].

The characteristic of the triple-graph-grammar formalism is that it permits consistency management. For this purpose the integration document has a crucial importance. Assume that two documents are related like in Figure 3.10 document A and B. During editing document A we remove element a. In case that we do not have an integration document, we cannot decide whether we have to remove b or (re-)create a. In the case that we have the integration document, after removing a, i and b exist and we know that they should also be removed. This situation also could be handled with some overhead without an integration document, but consider the more complex case that we remove a1. The relation is difined as (a1,a2)<---i1--->b1, such i1 and b1 have to be removed. Without an integration document the removal of b1 would result in the removal of a2, as inverse consequence of the document consistency.

In our case this means that the logical meta-model instances are related to the conceptual meta-model instances and later kept consistent. Hence we need a mapping model, i.e., the integration document that connects the two meta-models and a set of rules. Figure 3.11 shows the mapping model. This mapping model is adapted from the migration graph model from [Jah99]. We will give a short overview of the mapping model.

The meta-model for the analysed logical schema is represented as an abstract syntax graph on the left side of Figure 3.11. The node of type LSchema represents the root of the model for a logical schema. This root contains following classes: LView, which represents views of the logical schema; Entity, which represents the entities; and LType which represents the types of the entities attributes (LAttribute). Each Entity has an attribute ename that stores the name of the represented Entity. An Entity is composed by a non-empty set of named (vname) Variants, a primary key (LKey) that is referenced by the pkey association, and a set of alternative keys which are referenced by the association akeys. Each Variant

contains a set of foreign-keys (FKey) and a set of attributes (LAttribute). An attribute has an ltype association to refer to its type. An IND is represented by one of the three types I-IND, C-IND and R-IND (cf. page 43 and [FV95]), and has two associations lkey and fkey that point to a primary key and a foreign key.

The rational for selecting the conceptual schema is driven by following observations. Many variations and extensions of the Entity-Relationship (ER) model [Che76] have been proposed to facilitate the description of data structures. The most common extensions to extended Entity-Relationship models are concepts for abstraction like *aggregation* and *inheritance* [BCN92]. In our application domain, i.e., web information systems, the distributed programming language Java is widely used for integration and extension of legacy systems. Multiple inheritance is not allowed by the type system of Java [SCC+93]. Such we have restricted our conceptual model to classes to have at most one generalisation. Note that this restriction affects only the data models, that we want to access through a Java layer, and not the application programming languages of the reengineered legacy system.

The meta-model that specifies the chosen conceptual model is displayed on the right side of Figure 3.11. The class CSchema is the root of the conceptual schema abstract syntax graph. By analogy with the logical schema, this CSchema contains views (CView), classes (Class) and attribute types (CType). A boolean attribute (abstract) is used to store the information whether a class is abstract or not, i.e., whether a class can be instantiated. The name of a class is stored in attribute clname. Inheritance relationships are represented by nodes of type Inheritance with two associations sub and sup to the subclass and the superclass, respectively. Classes are composed by a set of CAttributes and an optional key (CKey). A CKey is composed by a non-empty set of CAttributes. Each Association has a name (aname) and attributes srcname and tarname store the role names of the classes that participate as source and target of the relationship, respectively. Attributes srccard and srctotal, and tarcard and tartotal represent the information about the cardinality of the source class and target class, respectively. The value of attribute srccard (tarcard) defines the maximum cardinality for the source (target) of the relationship. If the attribute srctotal (tartotal) is true, the relationship is total with respect to its source (target), cf. [EN94].

The schema mapping model connects the abstract syntax graphs of the logical and the conceptual schema and represents their associations. The elements of the schema mapping model are depicted in Figure 3.11. Their purpose is motivated and described in more detail in [Jah99].

MapSchema is used to connect the roots of both abstract syntax graphs. To map attribute types MapType is used. Each variant in the logical schema is mapped to a concrete class in the conceptual schema. If an entity has more than one variant, it implies usually that common attributes are comprised which leads to an inheritance hierarchy with abstract classes in the conceptual schema. Consequently, an abstract class is mapped to more than one variant, namely all variants which are represented by its concrete subclasses. These

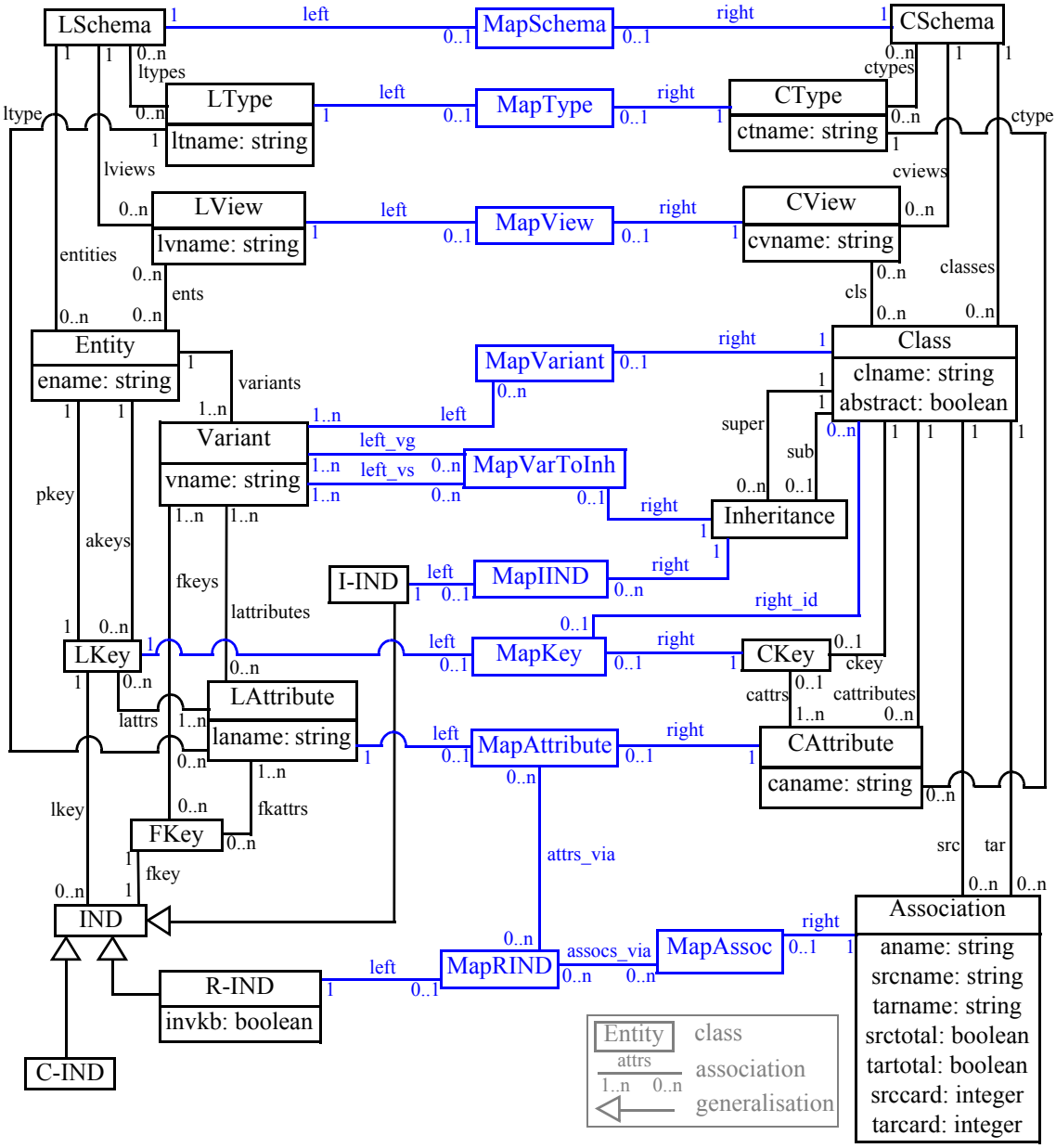


Figure 3.11: Schema mapping graph model

correspondences among classes and variants are represented by nodes of type MapVariant.

Inheritance relationships in the conceptual model can correspond in two different ways to constructs in the logical schema. Firstly, they can correspond to the inclusion of more specific variants in less specific variants that belong to the same entity. This is represented by an inheritance relationship in the conceptual model which is mapped by MapVarToInh.

Association left_vs is used to reference the variant which is more specific, while association left_vg references the variant which is more general. Secondly, the other possibility is to map INDs in the logical schema that have been classified as inheritance relationships (I-INDs) in the analysis process to inheritance relationships. In this case, the mapping is represented by a node of type MapIIND.

MapKey is used to map primary keys in the logical schema to keys in the conceptual schema. According to the ODMG 3.0 (Object Data Management Group) data model [CBB+00], our conceptual model includes the notion of unique object identifiers for instances of classes. Note that it is not required that every class contains a value-based key. Still, if we aim for object-relational data integration, unique object identifiers have to be resolved to value-based keys in the logical data model. For this purpose, every class has an association of type right_id that references a MapKey class in the schema mapping.

Attributes are mapped to attributes by nodes of type MapAttribute. To provide the flexibility to allow for different alternative schema mappings, we admit that attributes of a single class can be mapped to attributes in different entities. For such attributes, the access path from the entity that includes the value-based key associated to the class and the entity which includes such attributes has to be maintained. This is done by the association attrs_via. If a MapAttribute node does not have an attrs_via link, the mapped column belongs to the entity that contains the key referenced by the right_id association of the class that contains the mapped attribute. Otherwise, the mapped column belongs to a different entity and the attrs_via link of the corresponding MapAttribute node refers to a set of MapRIND nodes. These nodes represent the access path from the entity that contains the key referenced by the right_id association to the entity that contains the mapped column. Each MapRIND is connected to R-IND which logically represents a foreign key that has to be dereferenced to access the mapped column. In analogy with columns, MapAssoc and assocs_via associations are used to map Associations to sets of foreign keys (represented by R-INDs).

An example for a mapping rule is given in Figure 3.12. The MapEntityToClass mapping rule relates an entity including variant and primary key to a class with corresponding key. The prerequisite for this rule is that the logical root (LSchema) has a correspondent conceptual root (CSchema). This is the case if they are related by a MapSchema node. The prerequisites are represented in black whereas the elements which have to be related are in grey (green) and marked by a <<create>> stereotype. The side of each element is indicated by <<left>>, <<map>> and <<right>> corresponding to the left (logical schema), integration (mapping schema) and right (conceptual schema) document, respectively. Note that for execution reasons <<input>> and <<output>> nodes can be defined and are marked correspondingly. Further constraints and assignments can be expressed (e.g., LEFT : cl.clname:=v.vname;). The side stereotype expresses if the constraint or assignment is considered during generation.

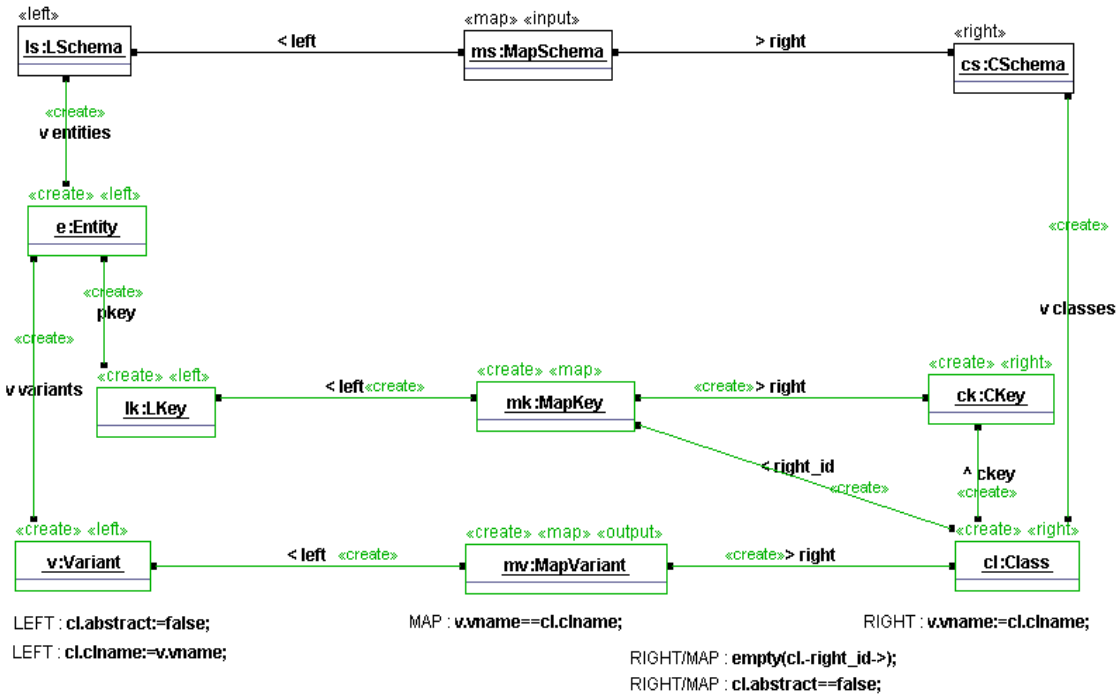


Figure 3.12: MapEntityToClass mapping rule

From this mapping rule three rules are generated: a rule that creates the conceptual elements from the logical ones (forward rule), a rule for the opposite direction (reverse rule) and a rule which relates logical and conceptual elements if they already exist (relating rule). The resulting rules are generated as story diagrams. A story diagram is an activity diagram where each activity contains either (Java) code or a story pattern. A story pattern is a graph production with an adapted collaboration diagram notation. We refer to [FNTZ98] for details.

Figure 3.13 shows the relating rule MapEntityToClassRelating as a story diagram. The MapSchema node ms is the input node, i.e., the node where the subgraph matching starts. The first activity initialises the MapVariant output node outParam. The next activity is a story pattern. Starting form the bound object ms, the black nodes are searched. For the relating rule, these are all logical and conceptual elements belonging to an entity and its corresponding class including the variant and keys. The variant v and class cl have the same name, cl is not abstract and does not have a value-based key assigned (empty(cl.-right_id->)). These constraints are all marked as MAP or RIGHT/MAP in the mapping rule. Once the subgraph has a corresponding match in the abstract syntax graph, the nodes in grey (green) will be created with the corresponding links. In this case MapKey and MapVariant nodes are created to relate the entity e, variant v and key lk to class cl and key ck. If the subgraph matching and creation of nodes is successful then the last activity assigns the

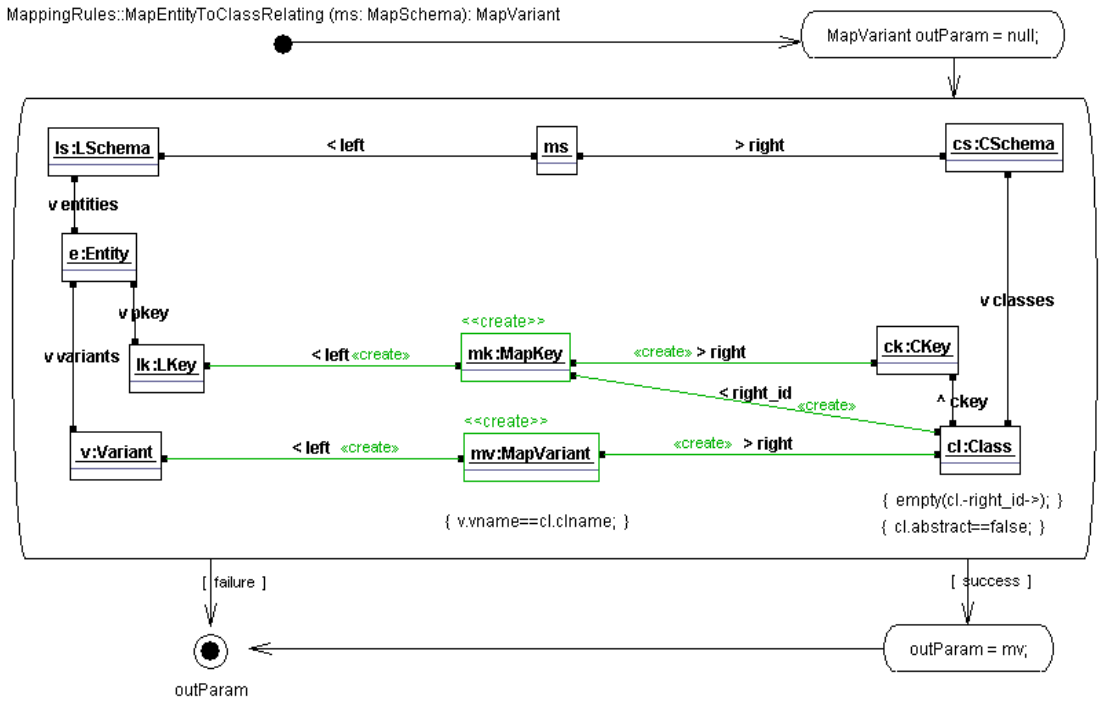


Figure 3.13: MapEntityToClass relating rule (story diagram)

MapVariant node mv to outParam. Otherwise the outParam remains null as assigned during initialisation. Finally the rule returns the outParam.

In analogy with the relating rule the forward rule (Figure 3.14) and the reverse rule (Figure 3.15) are generated. Only the story patterns are presented because the remaining of the story diagram is identical to Figure 3.13 except the rule names. The constraint and assignments are generated accordingly to the side they were defined for.

Other mapping rules are needed to establish a full correspondence between the logical and conceptual schema. The Table 3.1 gives an overview over the mapping rules needed for a bidirectional mapping and consistency preservation between the schemas.

Catalogues of (forward) mapping rules are given in [Wad98] and [Jah99]. To specify all rules special graph-grammar constructs are employed. The mapping rule MapAttrToAttr depicted in Figure 3.16 contains examples of a path construct, optional nodes and graphical constraints. A path enables the navigation through the abstract syntax graph from one node (or set of nodes) to another node (or set of nodes) even when they are not directly related by an association. A simple path from v:Variant to lk:LKey is given in Figure 3.16. This path <-variants-.-pkey-> accesses the entity primary key from the variant it starts from. Optional nodes are used in this rule to relate attributes that belong to a key. If an attribute belongs to a key the corresponding key nodes will be matched and the needed

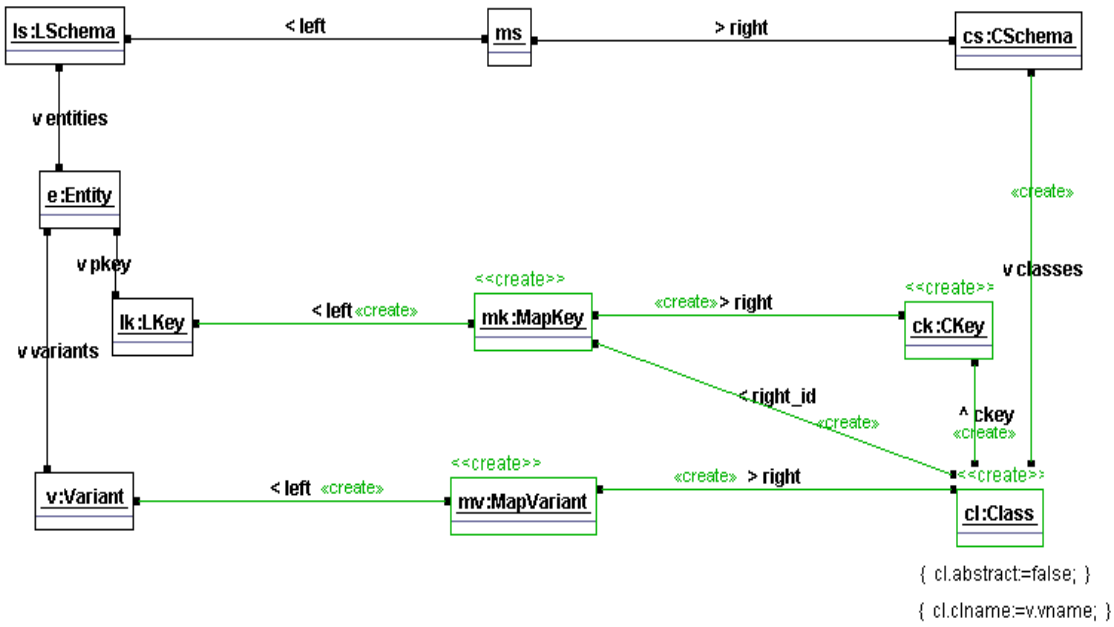


Figure 3.14: MapEntityToClass foward rule (story pattern)

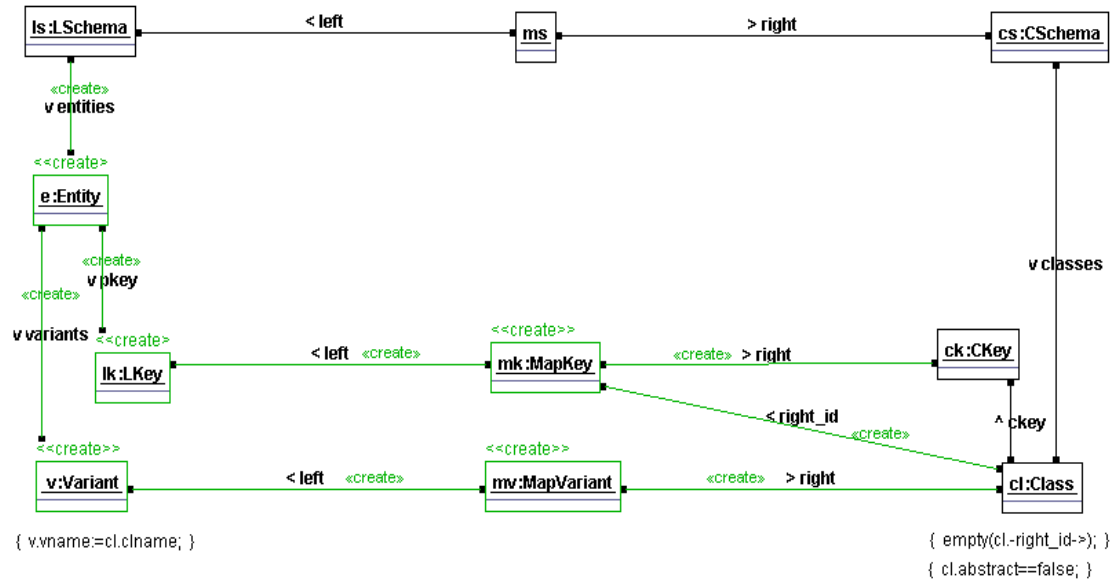


Figure 3.15: MapEntityToClass reverse rule (story pattern)

links to reflect this information will be created. If an attribute do not belong to a key the nodes containing information about a key will not be matched. The generated rules will

not fail because the nodes where declared as optional and thus are not required for the successful application of the rules. Finally, we have two graphical constraints.

mapping rule	description
MapSchema	Relates the root nodes of the logical and conceptual schema.
MapType	Relates the logical types to the conceptual types. In our case mainly database types to Java types, e.g., varchar to string.
MapView	Relates database views to UML class diagram views, cf. Section 3.1.3.
MapEntityToClass	Relates entities to concrete classes, including the primary keys.
MapVariantToConcreteClass	Relates additional entity variants to concrete classes.
MapVariantToAbstractClass	Relates variants to abstract classes in case that variant structures represent inheritance hierarchies.
MapVariantToInheritance	Relates variant hierarchal structures with the inheritance relationships of the corresponding hierarchy of concrete and abstract classes.
MapAttrToAttr	Relates the attributes of the entity variants and the related classes.
MapRINDToAssoc[1:1]	Relates an inversely key-based R-IND to a total one-to-one association.
MapRINDToAssoc[N:0,1]	Relates a non inversely key-based R-IND that has an inverse C-IND to a left-total one-to-many association.
MapRINDToAssoc[0,N:0,1]	Relates all R-INDs that are not inversely key-based and do not have an inverse C-IND to partial one-to-many associations.
MapIINDToInheritance	Relates I-INDs to inheritances.

Table 3.1: Mapping Rule Overview

Figure 3.17 shows the graphical constraint IsNotForeignKeyAttribute. A graphical constraint is expressed with a story diagram and attached to a node. The constraint is only checked if the attached node is a prerequisite of the rule. IsNotForeignKeyAttribute checks if the logical attribute is not a foreign key attribute in at least one variant. Logical attributes that contains information about foreign keys have no related attribute in the conceptual schema. This information is implicitly contained in associations. For this reason no such check is needed when a conceptual attribute is mapped to a logical attribute. The story diagram checks for the given attribute la that is contained in a variant, if a path via a foreign key from this variant to the given attribute does exist. If this is the case then the success variable becomes true and the constraint ("if (success)") will return "false", otherwise the failure variable becomes true and consquently the constraint will return "true".

The presented mapping is partial, e.g., many-to-many associations or aggregations are not considered. Mapping rules for such constructs can be defined in analogy with the presented rules. These rules lead to ambiguities, e.g., relating an R-IND to an association or an

aggregation. These ambiguities can be resolved by ordering mapping rules or adding further semantic annotations to the logical schema. Our experience shows that the number and the complexity of rules then grows considerably. We adopt a solution that enables a fully automatic mapping between the logical and conceptual schema with the limited set of twelve rules of Table 3.1. The reengineer can use conceptual schema refactoring operations to add conceptual constructs to the conceptual schema. In Section 3.2.4 we present an extensible catalogue of conceptual schema refactoring operations, which maintain automatically the correspondences to the logical schema.

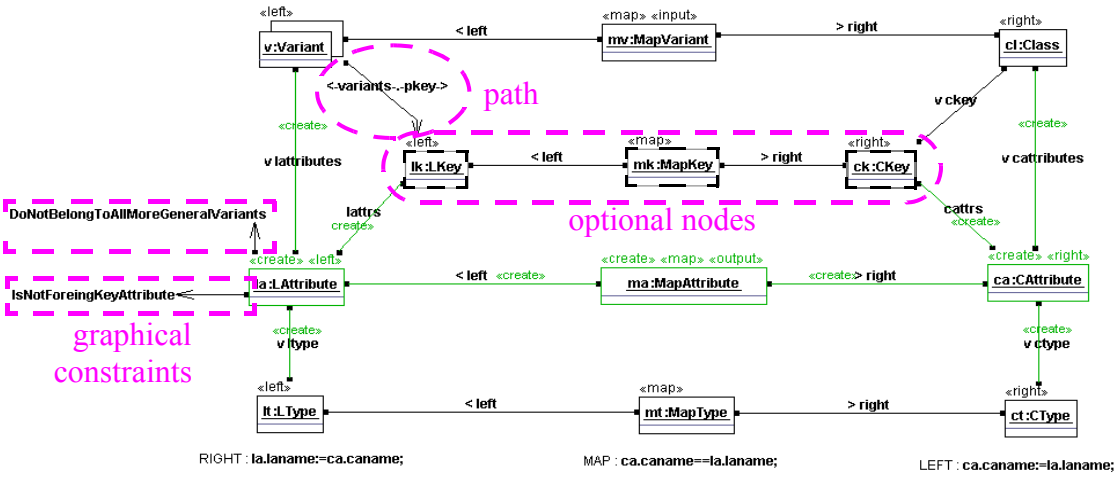


Figure 3.16: Mapping rule features in the MapAttrToAttr rule

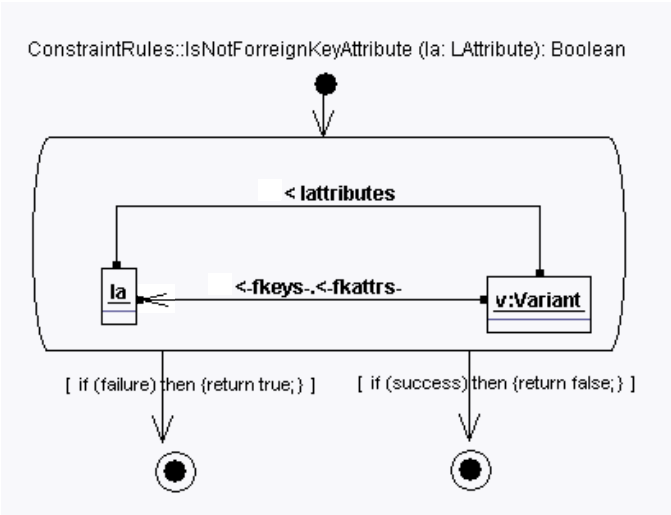


Figure 3.17: Graphical constraint with story diagram

3.2.4 Conceptual Schema Refactoring

According to [Fow99], refactoring is the modification of the internal structure of software without changing its observable behaviour, to make the software easier to understand, modify and maintain. The fundamental idea of refactoring is also reflected in conceptual schema refactoring, i.e., modifying the conceptual schema structure without changing the (observable) information capacity, to make the schema easier to understand, use and maintain.

Refactoring is a complex task regarding the impact analysis that a refactoring operation has to perform before it can be applied. This holds for software refactoring [Fow99], database (physical schema) refactoring [Amb02] and even logical schema refactoring. For conceptual schema refactoring this is different because the conceptual schema is not (yet) used by third parties. Indeed, the conceptual schema is newly created for understanding and will be used as capsule for the data access only after the reengineering. Two factors must be taken into account for conceptual schema refactoring: (1) the schema transformations must preserve the information capacity and (2) consistency between the conceptual schema and logical schema has to be preserved. While the first will be discussed subse-

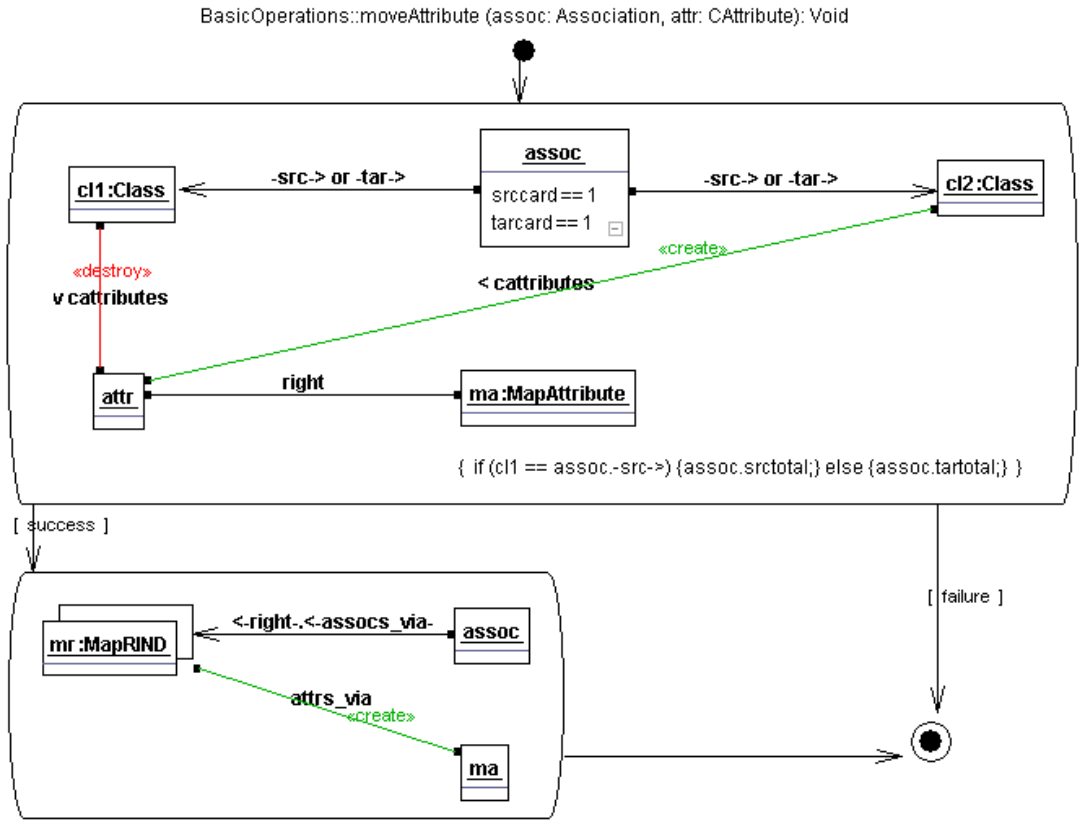


Figure 3.18: moveAttribute refactoring operation

quently, the latter is satisfied (guaranteed) by the mapping consistency mechanism with triple-graph-grammars (cf. Section 3.2.3).

Refactoring operations must preserve the information capacity [BCN92] of the schema. Crucial for an information capacity preserving operation is that the schema contains data structures, which enclose the same data before and after the operation application. An application, which runs and uses data described by the schema, must still run after schema refactoring, i.e., the data is still fully available. Formal graph transformations that preserve information capacity are presented for database integration [GO00] and database reengineering [JZ99].

An example for a basic refactoring operation is moveAttribute. Figure 3.18 shows the implementation of moveAttribute as a story diagram. Attribute attr is moved from class cl1 to class cl2 via association assoc. The prerequisite for this operation is that the move operation is done via a one-to-one association. This is done with assertions. An assertion is a constraint that is part of the matched node, e.g., srccard==1. Further the association has to be total with respect to class cl1. In other cases moving an attribute via an association would change the information capacity. Further the attribute has to have a link to a MapAttribute node. This ensures that attr has a corresponding attribute in the logical sche-

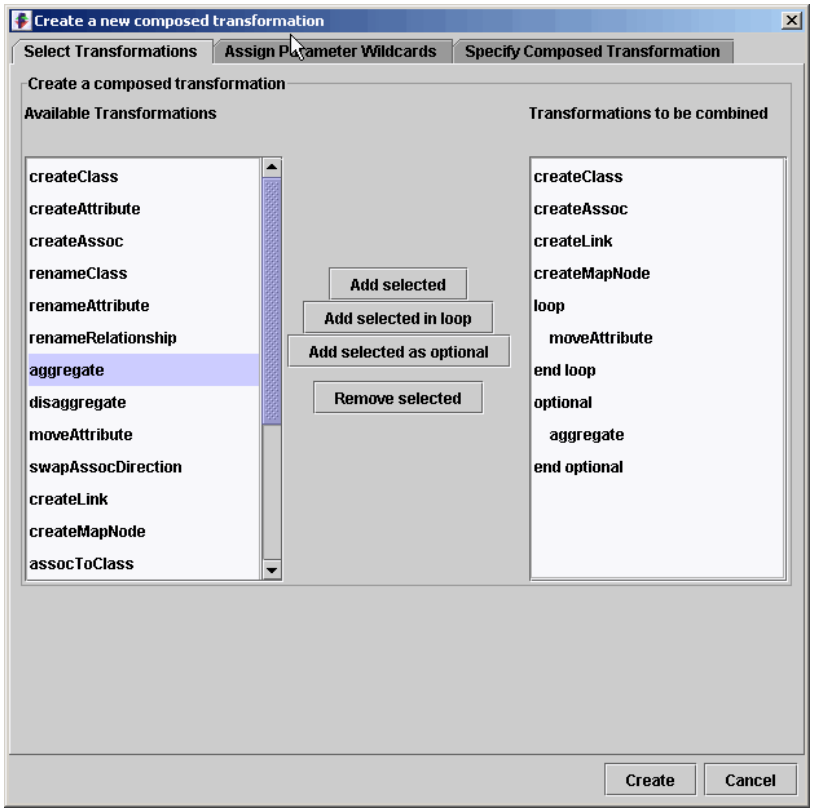


Figure 3.19: splitClass refactoring operation

ma. If the activity is applied the link between cl1 and attr is deleted and a link between cl2 and attr is created. After successful completion of this activity possible access pathes have to be updated. Note, if no MapRIND node is found the operation still terminates correctly.

Refactoring operations can be divided in basic schema transformations and composed schema transformations. Indeed much operations are based on several basic operations. Figure 3.19 shows splitClass which is an example for a composed refactoring operation. On the left hand the available transformations are listed. The new composed transformation is on the right side. The splitClass operation is composed of two basic design transformations createClass and createAssoc. Before moving the attributes from the old class to the new class, the correspondences to the logical schema are maintained with mapping transformations. createLink creates a link of type 'right_id' between the given new class and MapKey node. The new class is mapped to all variants, the old class is mapped to (createMapNode). Finally, the basic refactoring operation moveAttribute is applied in a loop to a passed set of attributes. Optionally the new class is aggregated (Aggregate).

Table 3.2 gives an overview of all refactoring operations classified into basic refactoring (BR) and composed refactoring (CR) operations. Note that operations, which do not preserve information capacity if they are applied separately (design operations), may be needed for composed refactoring operations, e.g., createClass in splitClass. Further, the correspondences to the logical schema have to be maintained, cf. MappingOperations Figure 3.19.

name	description	kind
renameClass	Updates the name of a class with a given new class name.	BR
renameAttribute	Updates the name of an attibute with a given new attribute name.	BR
renameRelationship	Updates the name of a relationship with a given new relationship name.	BR
Aggregate	Converts an association to an aggregation. Limited to associations inside packages.	BR
Disaggregate	Converts an aggregation to an association.	BR
MoveAttribute	Moves attributes from a class to an associated class via a given one-to-one relationship.	BR
SwapAssocDirection	Exchanges the source and target of a given association.	BR
renamePackage	Updates the name of a package with a given new package name.	BR
moveClass	Moves a class from a package to another package if this class is not aggregated to any other class. A constraint class cannot be moved alone.	BR
unifyAttrs	Merges two attributes to one attribute if they are related by a redundancy dependency.	BR

Table 3.2: Refactoring operation overview

name	description	kind
AssocToClass	Replaces an association between two classes by an intermediate class with two associations with calculated cardinalities to the initial two classes. Limited to relationships inside packages.	CR
ClassToAssoc	Replaces a class with two associations, both not many-to-many, to an association with calculated cardinalities.	CR
SplitClass	Divides a class into two classes associated by a one-to-one association or aggregation and moves the given attributes to the new class.	CR
MergeClasses	Merges two classes associated by a one-to-one relationship or an aggregation into a single class.	CR
moveClasses	Moves a given set of classes from a package to another package if these classes are not aggregated to any other classes outside the given set.	CR
MergePackages	Merges two packages into a single package.	CR
SplitPackages	Divides a package into two packages and moves the given classes to the new package.	CR

Table 3.2: Refactoring operation overview

3.3 Relationship Retrieval

The great data heterogeneity existent in web information systems and the various mechanisms used to integrate the subsystems make relationship retrieval quite complex. Therefore, we divide the retrieval problem into smaller problems, namely (1) the single schema recovery (cf. Section 3.2), (2) extracting code fragments of interest from the web information system and (3) the retrieval of relationships based on the extracted code fragments and the single schemas.

Retrieving relationships between schemas makes sense only if these are considered as *completely* reverse engineered. A complete schema in this context means complete after data model recovery. Nevertheless, not all schemas have to be recovered; our incremental process allows the addition of schemas later in the process such that relationships to these schemas can then be retrieved. Due to results of the relationship retrieval those schemas may have to be further analysed which can engender changes.

In contrast to the recovery of a single data model the inter-schema relationships are located in access and management data components, i.e., application code. Hence a code recovery mechanism is needed. Since we are only interested in the data related aspects of the code and parsing the entire existing code may not be scalable, we extract code fragments of interest. In some cases simple incremental parsing will be sufficient. In more complex cases we use island grammar parsing [vDK99, Moo01] and slicing [Wei84].

For the relationship retrieval, we use a pattern instances detection mechanism of patterns which interrelate data component model elements [Nie03]. The pattern instance detection is based on a (design) pattern instance recovery mechanism [NSW+02] which uses fuzzy logic to express uncertainty [NWW01, NWW03]. Validation of detected relationships which results in the adaptation of the associated fuzzy value can be done by several satellite activities. Finally, the reengineer confirms or rejects the retrieved relationships.

3.3.1 Code Fragment Extraction and Parsing

Relationships between attributes of entities in different schemas may occur everywhere in the application code and in various kinds. Such the problems we face for code fragment extraction are to determine which fragments are fragments of interest and how these fragments looks like. In case of web (thus, heterogeneous) information systems, however, the extraction problem is even more difficult because it deals with multiple different programming languages. Such we have to resolve an occurrence heterogeneity problem.

The code fragments of interest are those which manipulate the data. Such we have to extract all fragments that somehow accesses or manipulates the data through their corresponding data structures. This implies that we have to know if the data manipulating fragments are delimited by keywords and if they are multi-lingual.

In general, distributed transactions are used to ensure data integrity for the access and manipulation of multiple databases. The code used within such a distributed transaction may be spread over a set of methods. In practice besides local method calls also remote method calls are used. Within the application tier the current transaction context is propagated either in an implicit [COS98, OTS98] or explicit [AOS+99] manner. The resulting transaction boundaries can determine the relevant excerpt of code fragments for the analysis of relationships. Such the fragments are delimited which reduces the lines of code dramatically and allows us to handle also large applications.

Further, relationships are included in the application code where the database access is done by using API's provided by the database itself. For example, JDBC is a standard interface to access various kinds of databases. The interface provides data structures which can be accessed and modified in the Java programming language. Hence a complete analysis of an application is too expensive regarding the analysis time, we can use the JDBC interface declarations as starting points for extracting fragments of interest only. This also holds for specific in house interfaces.

An easy case of fragment extraction is when the data manipulation is done via query log files. Query log files contain all the access and manipulation operations normally in the database manipulation language, e.g., SQL. These files are called by the application code to perform the data manipulations, such as done with scripts. Such a query log files can be seen as a result of fragment extraction. In fact, in this case is more a fragment detection than extraction. Nevertheless, the files also have to be located.

To find the fragments of interest we apply two techniques: island grammar parsing and slicing. The island grammar parsing covers the cases where the fragments are delimited but not entirely the multi-language problem. Since we start from the data models, we know the entities and attribute names. We can use a slicer to find all the occurrences of these entities and attributes. To extract the fragments of interest, they do not need to be delimited and multi-language fragments can be handled. The drawback of slicing is the size and number of fragments extracted this way. Indeed, slicing the application code for all entities and attributes would cover huge parts of the code. Only selective slicing is manageable and scalable.

Therefore we define the following extraction process. We start with the identification of fragments by using an island grammar parser, cf. Figure 3.20. So called *island*s are extracted according to the grammar definition. Next, in case that an island is encapsulated in a method, we extract the method signature and slice the method calls backwards. Finally, the such extracted code fragments are parsed and attached to the abstract syntax graph representing the data component models.

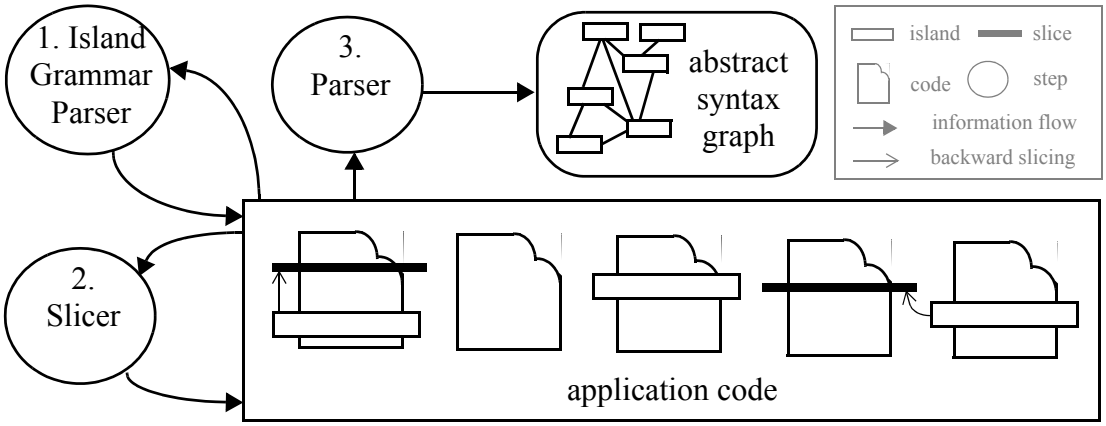


Figure 3.20: Extracting code fragments of interrest

Island grammar parsing

To automate the extraction of code fragments of interest, we make use of parser technology to import the legacy code into the abstract syntax graph that can be further processed. This is not new and has been done for decades in numerous reverse engineering tools. Today, parsers, or parser generating grammars, are available for a wide variety of languages, including database languages like SQL and programming languages like COBOL, C and Java. These parsers have been used to build extractors for reverse engineering homogeneous systems, i.e., systems implemented in a single language.

In case of web (thus, heterogeneous) information systems, however, the extraction problem is more difficult because it deals with multiple different programming languages. Us-

ing multiple different parsers solves the problem only partially, because this approach fails to capture the relationships between software artefacts written in different languages. This problem gets worse for multi-language systems where certain languages are embedded in other languages. This situation is typical for many web information systems: most database management systems provide proprietary data manipulation languages embedded in various host languages like C, COBOL, Java, etc. In addition, the code fragments may contain code pieces from multiple modules integrated in the application tier via interoperable interfaces [Cha96, Ib96, COR99].

Parsers for extracting relationships among distributed schemas have to deal with code fragments, that are amalgamations of different languages including proprietary dialects. This feature renders the reuse of existing parsers highly unlikely. In addition, experiences show that the construction of multi-lingual custom parsers might become a fairly complex task. The reduction of reverse engineering effort achieved by the resulting extractor may be lost when building and adapting a multi-lingual custom parser. However, the code fragments of interest are not arbitrary amalgamations of different languages but rather the well separated code fragments executed within distributed transactions. Therefore, we can simplify the task by looking at its specific characteristics.

Still, it is important to note that the code fragments of interest for extracting relationships among distributed schemas are typically in a fairly small subset of the multi-language grammar. Therefore, it is viable to construct more simple parsers that filter out only those "interesting" parts of the multi-language syntax and ignore everything else. A naïve way of performing this filtering is using a pre-processing step with a lexical analyser such as the Unix grep command.

Reverse engineering researchers have started to investigate more powerful approaches, one of them being island grammars [vDK99]. An island grammar can informally be defined as a set of production rules that describe the language fragments of interest (so-called islands) plus another set of production rules that catch the rest (so-called water). Obviously, the idea behind the concept of island grammars is to make the water significantly less descriptive than the islands in order to decrease the complexity of the associated parser. For a formal definition of island grammars and how to build robust island grammar parsers we refer to [Moo01].

The definition of an island grammar for our application, the extraction of relationships among distributed, heterogeneous information systems, is a task that requires an intimate understanding of the concept of island grammars and a highly explorative process. To simplify this process, [Chu04] developes a tool for interactively creating island grammars based on code examples of interest identified by the user.

This tool, called BUFFY, initially assumes that the entire input code represents water. When the user identifies instances of interesting code fragments, BUFFY suggests a set of island productions for this instance. Subsequently, the user can interactively correct and refine these productions to characterise the associated island. Then, the user can generate

```
public class Database implements DatabaseBackendInterface
...
public String getValue(int databaseConnectionID, String keyName, String table-
Name, String[] attributeName, String[] attributeValue, String[] type)
{
  ResultSet result = null;
  String sqlString = "SELECT " + keyName + " FROM " + tableName + " WHERE ";
  // loop around the array
  for(int n = 0; n<attributeName.length; n++)
  {
     if( n == (attributeName.length - 1) )
     {
        if(  type[n].toUpperCase().equals("STRING") ||
             type[n].toUpperCase().equals("DATE") )
        {
           sqlString = sqlString + attributeName[n] + " = '" +
              attributeValue[n] + "'";
        } else {
           sqlString = sqlString + attributeName[n] + " = " + attributeValue[n];
        }
     } else{
        if(  type[n].toUpperCase().equals("STRING") ||
             type[n].toUpperCase().equals("DATE") )
        {
           sqlString = sqlString + attributeName[n] + " = '" +
              attributeValue[n] + "' AND ";
        } else {
           sqlString = sqlString + attributeName[n] + " = " +
              attributeValue[n] + " AND ";
        }
     }
  }
  String retValue = "@ERROR@";
  try {
     this.lastUsedConnection.put( new Integer(databaseConnectionID),
             new Long(System.currentTimeMillis()) );
     result = select( databaseConnectionID, sqlString );
     result.next();
     retValue = result.getString( keyName );
  } catch (Exception ex) {
     retValue = "@ERROR@";
  }
  return retValue;
}
...
```

Figure 3.21: Code fragment of interest

a prototype extractor and run it against other parts of the input code in order to verify if this island recognises other instances of this pattern. Depending on the result of this verification step the user might iteratively refine the description of the island. Resulting, the

fragment extractor yields islands (code fragments) delimited by transaction boundaries or method definition including the indicators for the relationship retrieval.

Figure 3.21 shows an example of a code fragment of interest. In this case it is the whole method getValue. The island grammar production for this code fragment is relatively simple: fragments of interest contains the keywords „SELECT“, „FROM“ and „WHERE“ and methods build the fragments boundaries. Still, this fragment on its own is not a complete indicator, to identify the elements involved further recovery is needed.

Slicing

To trace the use of code fragments of interest inside methods we employ slicing [Wei84]. Various notions of slicing have been proposed. For an overview we refer to [Tip95, HG98]. To slice the programs for nested method calls interprocedural static backward slicing is appropriate. Starting from the method declaration the calls of this method are sliced backwards.

in DatabaseBackendInterface:

```
public String getValue(int databaseConnectionID, String keyName,
          String tableName, String[] attributeName, String[] attributeValue,
          String[] type) throws RemoteException;
```

in DatabaseConnection:

```
public String getValue( String keyName, String tableName, String[] attribute-
Name, String[] attributeValue, String[] type)
{
   String retValue = "";
   try {
      this.checkConnection();
   retValue = this.databaseBackend.getValue( this.databaseConnectionID,
              keyName, tableName, attributeName, attributeValue, type);
   } catch (Exception ex) {
      if(this.out)
      {
         ex.printStackTrace();
      }
      retValue = "";
   }
   return retValue;
}
```

in XMLWorker (lines 479 and 559):

```
roleid=this.databaseConnection.getValue("roleid", "roles",
            new String[]{"userid", "projectid", "rolename"},
            new String[]{userid, projectid, "programmer"},
            new String[]{"int", "int", "String"});
```

Figure 3.22: sliced code fragment of interest

Figure 3.22 sketches an example where the method containing the code fragment from Figure 3.21 is sliced backwards. Starting point is the method getValue in class Database. Class Database implements the (abstract) class DatabaseBackendInterface where getValue is declared. In the class DatabaseConnection method getValue is called inside another method getValue. Note that the method signatures are different. This second method getValue is finally called in class XMLWorker with concrete values.

The difficulty of such slices is to know the depth of nested method calls. Two solutions are considered. Firstly, from our experience a depth of 3 is largely sufficient to cover all nested methods until the call with concrete values. The drawback is that often the slices produced are larger than required because the nested method call depth is less than 3. Contrariwise all nested method calls greater than 3 are not sliced to their end. Secondly, the slicing can be done user-driven until the method call with concrete values. The drawback here is that the fragment extraction is not longer automatic and thus user intensive.

We opt for the first solution because we search for indicators only. Moreover we try to reduce the user involvement in this early activity. The example of Figure 3.21 and Figure 3.22 is of depth 1. The starting method declaration is getValue from Figure 3.21. The first nested call is in class DatabaseConnection. The call in XMLWorker is not nested; it is the call with concrete values. Note that the method calls sliced backwards of depth 2 and 3 were omitted in this example.

Parsing

Our pattern instance recovery approach is graph based. Therefore, we represent the schemas as well as the manipulating source code as an abstract syntax graph. This abstract syntax graph is an object-oriented graph, cf. [Zün01, Nie03]. The code fragments are parsed into abstract syntax graphs. Additionally, these (code fragments) abstract syntax graphs are attached to internal schema abstract syntax graphs.

The output of the island grammar parser are either code fragments of interest embedded in methods (followed by slicing) or standalone code fragments of interest. In both cases the result will be an abstract syntax graph. In case that the corresponding method calls have to be sliced the abstract syntax graph is not yet attached to schema abstract syntax graph. The standalone code fragments are directly attached to the schema abstract syntax graph.

The slicing results and query log files have to be parsed and attached to the schema abstract syntax graph. In case of the slicing the linking elements are the method declaration headers. Otherwise, the linking of the abstract syntax graphs is done via attributes and classes of the object-oriented graph.

For mid-sized unilingual web information systems a sequential incremental parsing can avoid code fragment extraction. The files containing code are parsed sequentially and the resulting abstract syntax graphs are only stored temporally. An iterative process is then

needed that combines the parsing and pattern instance recovery. A file is parsed followed by the pattern instance recovery. Only the recovered pattern annotations remain in the schema abstract syntax graph and the code abstract syntax graph is deleted. Then the next file is parsed and so on. Classes, attributes and method headers are directly parsed and represented in an abstract syntax graph. The method bodies are incrementally parsed on demand. This parsing on demand has then to be triggered from the pattern instance retrieval process.

3.3.2 Pattern Definition

Detecting instances of patterns for relationship retrieval requires that patterns be formally defined, since informally described parts of the patterns are not amenable to (semi-)automatic recovery. The most formal part of patterns is the structure part. In our approach patterns (and subpatterns) are defined with respect to the abstract syntax graph. Subpatterns define abstract syntax graph structures that are constituent parts of other patterns or subpatterns.

Using an abstract syntax graph representation has several advantages over using a textual source code representation. It is easily combinable with representation of the schemas which is generally done with graphs, in our case also with abstract syntax graphs. It avoids white space and formatting problems. It automatically normalises the code for simple syntactic variants such as 'SQL SELECT' vs. „Select". Abstract syntax graphs also provide additional information, such as identifier application and declaration links, that is useful in further analysis.

We use a simplified abstract syntax graph model for readability reasons for examples in this thesis. Figure 3.23 shows an excerpt of the type graph model underlying the pattern definition as an UML class diagram. We omit the complete graph model and just present the classes Node, ASGNode and Annotation. Each element of the abstract syntax graph is represented by a corresponding subclass of class ASGNode. Each annotation used in a pattern definition is a subclass of class Annotation. The attribute kind qualifies the pattern, e.g. SELECT and Comparison in Figure 3.23. The Annotation nodes are associated to Node nodes, i.e., ASGNode nodes or Annotation nodes, in the graph by a qualified asso-

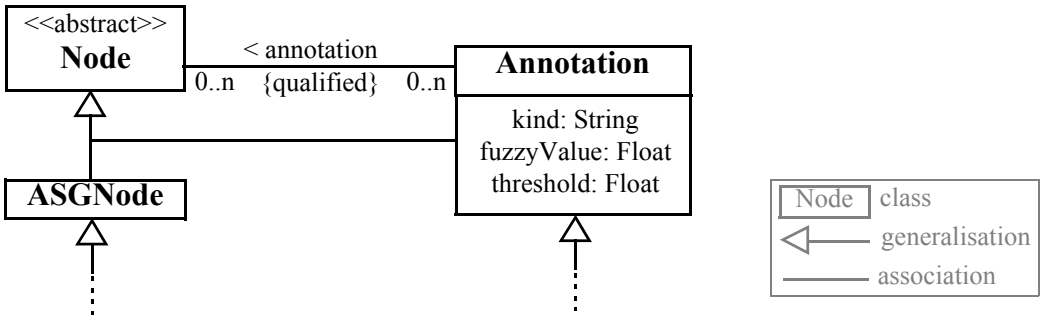


Figure 3.23: Type Graph Model

ciation. The qualifier must be unique. The other two attributes of Annotation are used for handling uncertainty in the retrieval process.

Recognising an instance of a pattern or subpattern in the abstract syntax graph under analysis results in the addition of a corresponding annotation. The oval-shaped node :SELECT shown in Figure 3.24 is an annotation which identifies attributes refid and fileid as participating in a select join. In generally, oval-shaped nodes are annotations which identify certain subgraphs of the abstract syntax graph as matching the named subpatterns. The rectangles represent nodes of the abstract syntax graph. The nodes are linked together in accordance to the underlying type graph model. The indirect link between class "UpdateService" and :PTAssignment indicates that the assignment is contained in the class, i.e., abstract syntax graph nodes where omitted. Further we omitted abstract syntax graph nodes in the WHERE clause and just show the :Comparison annotation.

```
// Select join embedded in Java code
result=this.databaseConnection.selectSQLVector(
             "Select files.path
             from product, files
             where   (product.refid = files.fileid)
                      AND
                     (files.name = updateFile )"
       );
```

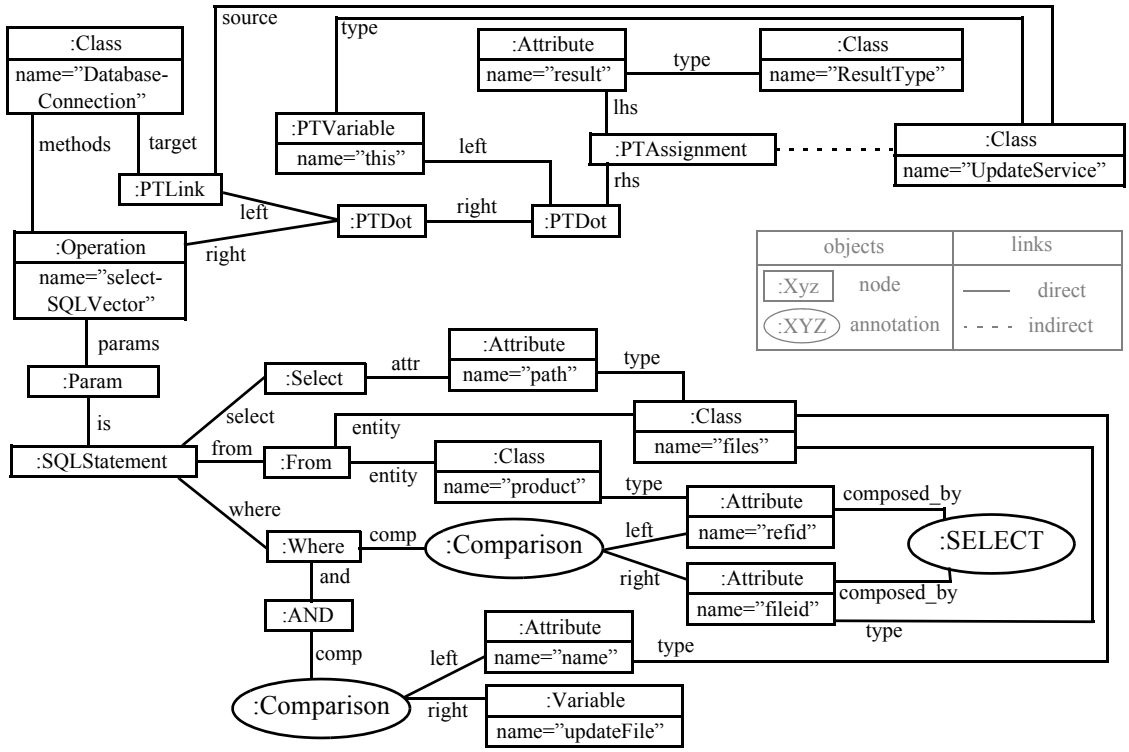


Figure 3.24: Simplified annotated abstract syntax graph instance

In our approach each pattern is formally defined by a graph production. The corresponding graph production annotates the abstract syntax graph with additional nodes and links (the oval-shaped nodes and corresponding links in Figure 3.24) to indicate which subgraph of an abstract syntax graph correspond to the pattern. Such subgraphs can then be used by rules defining other patterns that contain the defined pattern as a constituent part.

The rule definition is graph based, means the approach needs only a graph representation of the schemas, code or any other graph representing information of a system, e.g. data-flow or control-flow graphs. Therefore, the presented approach is not bound to any particular program language or any particular programming paradigm.

As an example of such a pattern definition, Figure 3.25 shows the graph production defining a pattern which is an IND between two entities[1]. In the notation used, the subgraph to be matched in the host graph is defined by the black nodes and edges. The subgraph to be added is defined by the grey (green) node annotated with the stereotype <<create>> and grey (green) edges. This simple notation can be used because the rules only add information to the host graph and never delete any. The formal definition and theory underlying such graph productions is given in [SWZ95]. The stereotype <<trigger>>, as well as the values 100 / 0, will be explained later in this chapter.

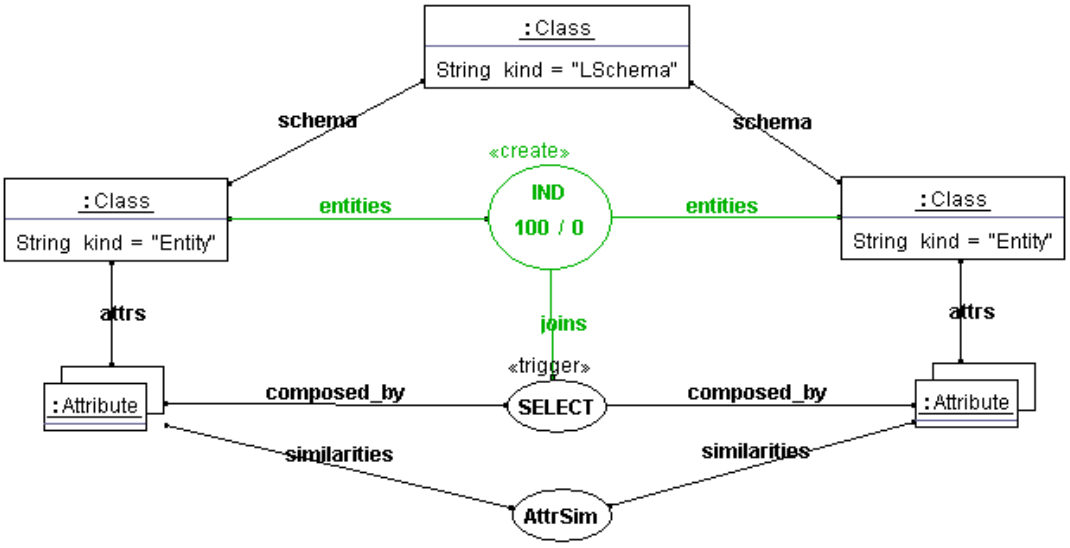


Figure 3.25: IND pattern definition

Figure 3.26 shows an example of an SQL query which contains two INDs. We have occurrences of the SELECT and AttrSim pattern for (users.userid, roles.userid) and

1. Note that internally, i.e., in the abstract syntax graph, all schema elements are represented in an object oriented graph like defined in [Zün01] and [Nie03].

(roles.projectid, projects.projectid). Applying the graph production from Figure 3.25 to this code will result in the annotations of two INDs. One between entities users and roles and one between entities roles and projects. These three entities are part of the same logical schema, i.e., from the DSD database (cf. Chapter 2). The resulting IND annotations in the abstract syntax graph are shown in the lower part of Figure 3.26.

```
// get the email addresses from project +mailModuleName+ for all persons having
the role rname ...
result=this.databaseConnection.selectSQLVector("Select users.email from us-
ers,roles,projects where (users.userid = roles.userid ) AND (roles.projectid =
projects.projectid) AND (projects.name='"+mailModuleName+"') AND (roles.role-
name=rname)");
```

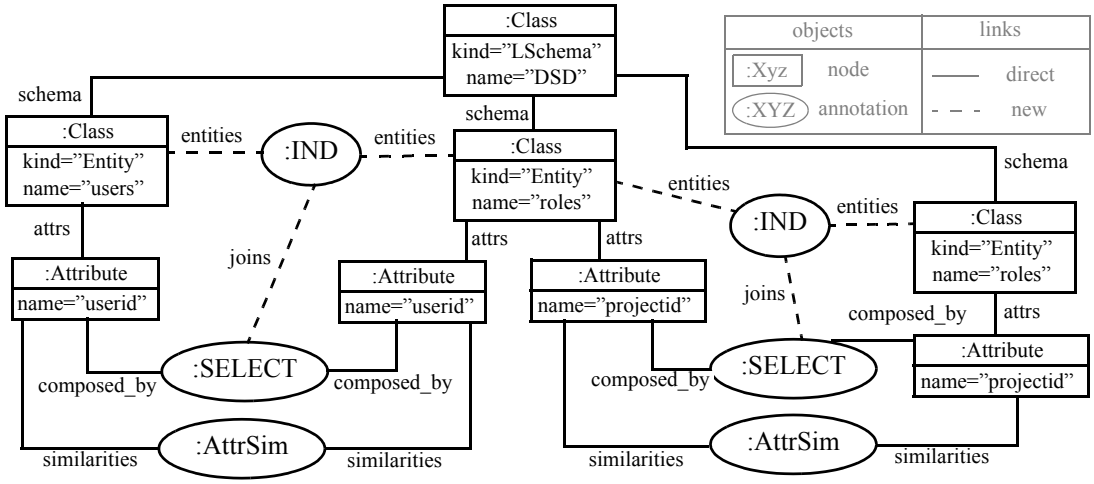


Figure 3.26: Sample IND code and annotated abstract syntax graph

Figure 3.27 shows the IND annotations presented to the reengineer, i.e., between entities in the EER diagram.

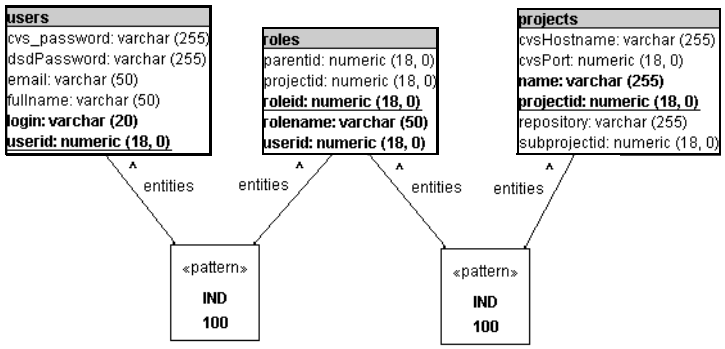


Figure 3.27: IND annotations

An IND is a subpattern for an R-IND. In case that two entities were annotated with an IND and that one side of the participating Attributes are a PrimaryKey we have an R-IND, cf. Figure 3.28. To ensure that the Attributes composing the IND are chosen the SELECT annotation is used. Finally, we ensure that the IND was not already annotated as IIND because I-INDs and R-INDs exclude themselves mutually, cf. [FV95]. The IND examples of Figure 3.26 are R-INDs because userid and projectid are primary keys of users and projects, respectively.

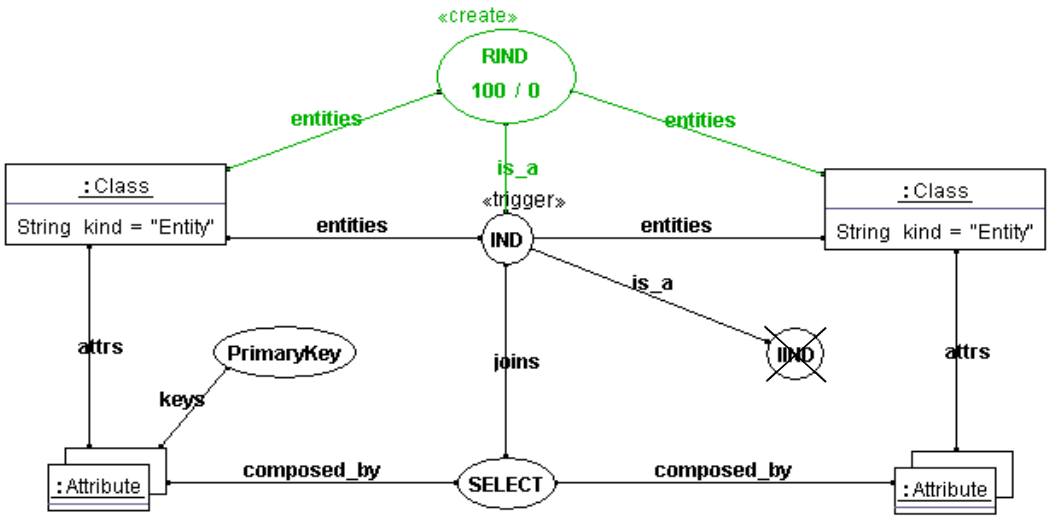


Figure 3.28: R-IND pattern definition

The graph productions for I-INDs and C-INDs are similar to the R-IND rule. Definitions of these patterns and subpatterns, e.g., AttrSim, PrimaryKey or SELECT, can be found in [Bew98] and [Jah99]. Note that annotations can also by added be the reengineer.

In case of inter-schema relationships the pattern definition, i.e., the graph productions, are composed of elements of the conceptual schemas. Figure 3.29 shows the Association pattern specification which is in analogy with the specification of an R-IND (Figure 3.28). The code example shows an association between tblOutcomes in database outcomes_be and Patient in database hospdata. The key is VHS_ID# from Patient. In analogy with R-IND, Association and Inheritance exclude themselves mutually.

Finally, we present a definition of a redundancy dependency namely replication. The example code of Figure 3.30 shows an INSERT query for tblDose followed by a nested UPDATE / SELECT query that indicates replication. Attributes LastDose, LastPurpose and LastRoute from hospdata.DrugProfile are updated with the corresponding attribute values of Dose, Purpose and Route from tblOutcomes.tblDose. The WHERE clause can be discarded for replication retrieval. The Replication pattern definition of Figure 3.30 exactly covers such situations. However replication can be implemented in different ways.

```
// get the the symtomes for a patient
SQL SELECT Patient.VHS_ID#, tblOutcomes.symtomeDescription, tblOutcomes.Date,
tblOutcomes.Time FROM outcomes_be.tblOutcomes, hospdata.Patient WHERE tblOut-
comes.VHSID = Patient.VHS_ID# AND Patient.SURNAME = :SN AND Patient.FIRST NAME
= :FN;
```

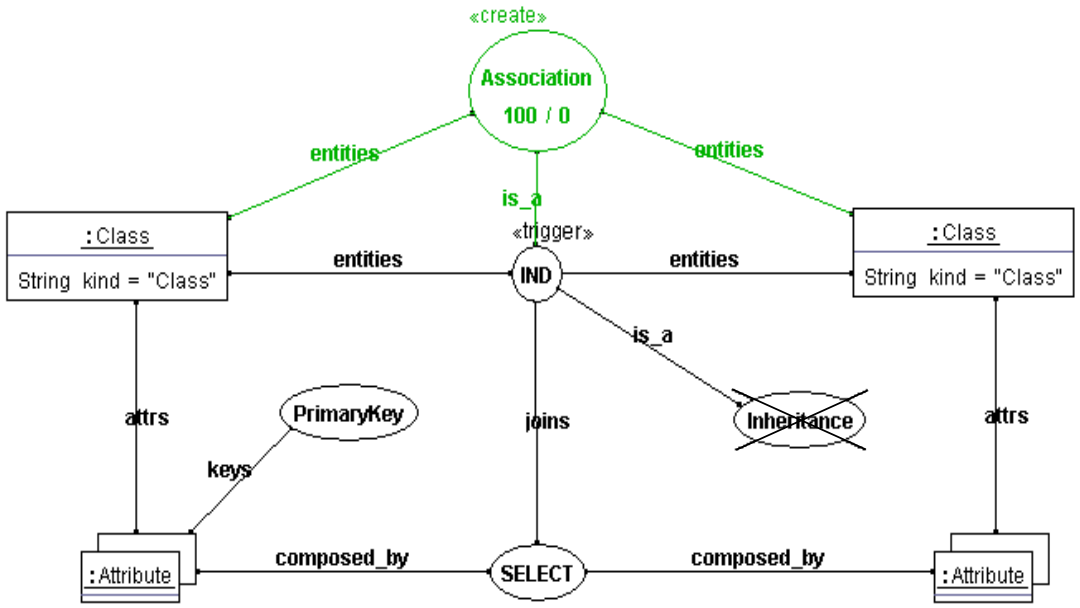


Figure 3.29: Association code example and pattern definition

Defining a pattern for every possible implementation would result in a multitude of pattern definitions and consequently graph productions. For this reason we group the definitions. An example is the Comparison pattern:

- `product.refid = files.fileid` (Figure 3.26)
- `DrugID = (SELECT DrugID FROM ...` (Figure 3.30)

In both cases a comparison of the attributes is done within the WHERE clause from a SELECT statement. The abstract syntax graph enables that one pattern definition covers both implementations. Another example is the R-IND and Association pattern definitions. These two patterns differ only in the schema they can be applied. If we ignore that the participating entities are either entities or classes and treat an IIND as an Inheritance, both pattern definitions are identical.

Figure 3.3 represents an excerpt of the domain model that underlies the pattern definition, i.e., it gives an overview of the relationships and data dependencies. In the domain model we do not distinguish between elements of the logical or conceptual schema. For example we have entity, attribute, PrimaryKey, IND, association, inheritance, etc. Depending whether we are in the logical or conceptual schema an entity is an entity or a class, or an

```
SQL INSERT INTO tblOutcomes.tblDose VALUES (dose_id, PatientID, date, time,
drug, dose, route, purpose, provider, NULL, NULL, NULL);
...
SQL UPDATE hospdata.DrugProfile SET LastDose, LastPurpose, LastRoute SELECT
Dose, Purpose, Route FROM tblOutcomes.tblDose WHERE (VHSID = PatientID) AND
(DrugID = (SELECT DrugID FROM hospdata.Drugs WHERE Description = drug)) ;
```

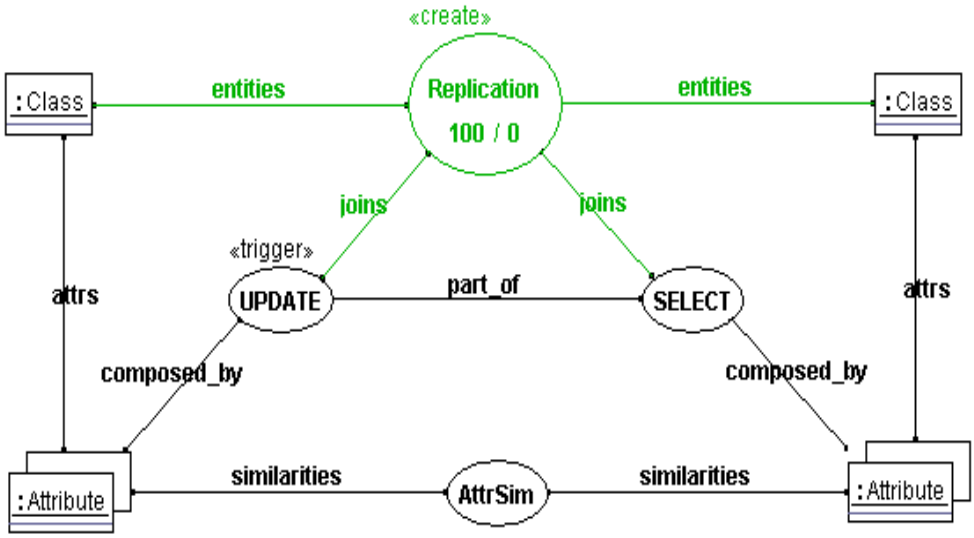


Figure 3.30: Replication code example and pattern definition

inheritance is an IIND or inheritance. Note that, e.g., INDs are annotated in both schemas but only shown in the logical schema representation (EER diagram).

Nevertheless, grouping of annotations in the domain model and of pattern definitions still results in a multitude of graph productions. We refer to [NWW03] were this is shown for association detection in Java code. We overcome this problem by introducing uncertainty in our graph productions and thus pattern definitions. This will be discussed in Section 3.3.4.

3.3.3 Pattern Instance Retrieval

We described an effective formalism for defining a catalogue of patterns as the basis for pattern instance retrieval. The retrieval process for pattern instances (in web information systems) is inevitably an iterative one. Typically, the reverse engineer first applies an initial set of patterns, then repeatedly examines the results, adjusts the patterns to address perceived deficiencies and reapplies them until a satisfactory outcome is achieved. To support this process the engineer needs tool support that applies the pattern instance retrieval process to the system involved and displays the results obtained.

To devise a tool that meets this requirement we adopt a threefold strategy. Firstly, we minimise the rule scalability problems by adopting the best available analysis algorithm. Secondly, we adapt this algorithm to deliver useful results incrementally rather than on completion. Thirdly, we involve the reverse engineer in the analysis process, to avoid unnecessary computation of unwanted analysis results. We give an overview of this strategy, deeper details can be found in [Nie03].

The basic analysis algorithm

Pattern-based retrieval is a deductive analysis problem where patterns, or rules, are repeatedly applied to the abstract syntax graph to arrive at the most complete characterisation of the system permitted by the rules. Pure deductive analysis algorithms typically apply the rules involved level by level, bottom-up[1], according to their natural hierarchy, and produce useful results only when analysis is complete. Results from other researchers, such as [Qui94] and [Wil96], suggest that a reverse engineering tool providing fully automatic analysis based on this approach cannot scale for larger software systems.

Where patterns are defined as graph productions, as in our case, graph transformation systems are the natural choice for implementing the tool. However, the scalability problem also applies to graph transformation systems such as PROGRES [Zün95] or AGG [AGG], which apply the rules in an arbitrary sequence usually determined by the internal data structures used.

In practical application it has been experienced that rules are generally applied in a given context. Therefore, FUJABATS, in contrast to other graph transformation systems, applies rules given a context, normally one object in the graph. The advantage is that it reduces the runtime complexity of the rule matching algorithm to polynomial size, whereas the original sub-graph matching problem is NP-complete [Meh84]. This restriction to the original theory of graph grammars has been shown not to be a problem. For more details we refer to [Zün01] and [FNTZ98].

By adopting FUJABATS as the platform for our tool, we therefore reduce the problem of scalability compared to systems using standard approaches to deductive analysis. For the reverse engineer, however, this does not necessarily solve the performance problems involved.

Adapting the analysis algorithm

Although FUJABATS reduces the computational complexity of analysis, a fully-automatic tool based on FUJABATS is still undesirable, as the results are made available only when

1. In comparison, pure top-down approaches starting with top-level rules in the topology hierarchy are only of theoretical interest, because of the search-space implied. Even when a specific rule is identified for application, without an adequate starting context its top-down application is impractical.

analysis is complete. Given that reverse engineering is an iterative process, such tool behaviour does not lead to an efficient overall process.

Suppose, for example, the Duplication pattern of Figure 3.31 does not include the attribute similarity (AttrSim) check. The resulting false positives manifest themselves early in the analysis, but the reverse engineer has to wait until analysis is complete to recognise them. For reverse engineering, therefore, a semi-automatic process is likely to be more effective, in which useful intermediate results are produced and the engineer is allowed to interact with them, either to add information and request that analysis continues or to revise the rule definitions and restart analysis.

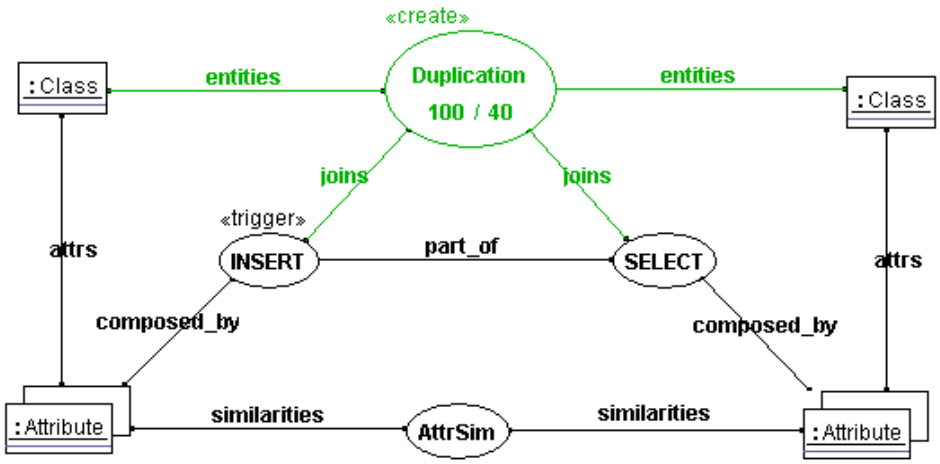


Figure 3.31: Duplication pattern definition

To support such a process, the analysis algorithm itself must produce intermediate results useful to the engineer as early as possible, and be amenable to interruption and resumption without loss of results to date. Since the results most useful to the engineer are those produced by rules at the highest levels in the rule hierarchy, we adopt an analysis algorithm which combines a bottom-up strategy and a top-down strategy. Note that the algorithm affects only the execution sequence of patterns and does not violate their formalisation as graph productions.

To define the algorithm, the dependency hierarchy of the rules is levelled, such that each rule has a level number. A rule depending only on objects in the initial abstract syntax graph gets number 1. A rule depending on other rules, i.e., whose definition includes annotations created by other rules, gets a higher number consistent with the natural topological order of the rules. Rules included in cycles concerning their dependencies get the same level number and are marked as recursive.

Figure 3.32 shows a snapshot of our analysis algorithm. The grey rectangle at the bottom represents all objects in the abstract syntax graph. The black oval identifies an annotation

already created by bottom-up analysis (with a link to the annotated object a1) while grey ovals, together with the grey links, represent a top-down analysis in progress. Directed arcs indicate the scheduling sequence of the rules. Variables at the arcs represent objects passed to the scheduled rule as context. We omit the links for :AttrSim, :NS and :TC for readability reasons.

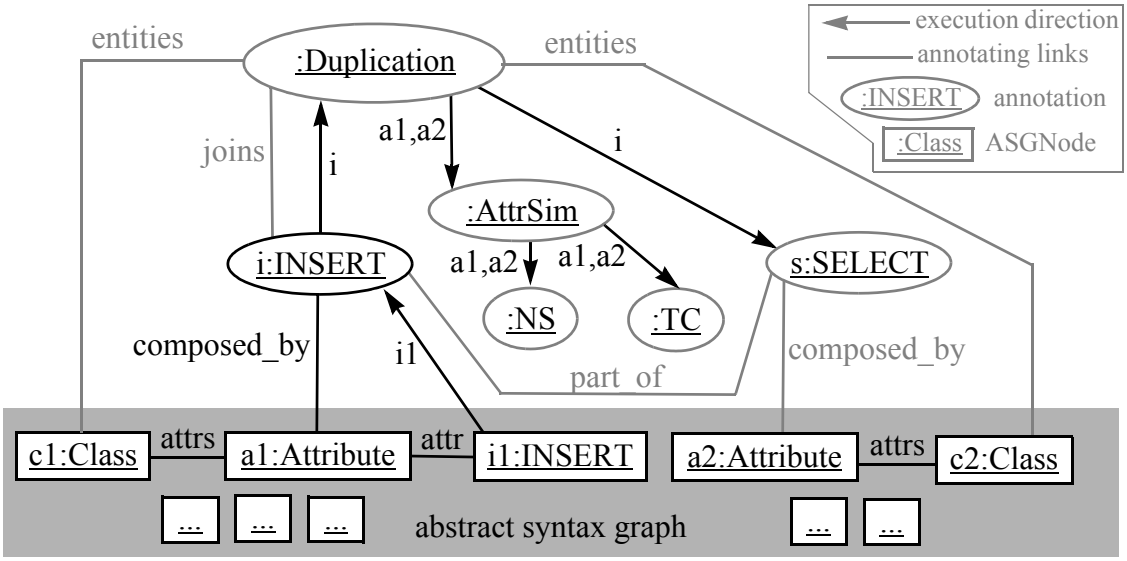


Figure 3.32: Sample analysis execution

Bottom-up strategy

After the abstract syntax graph is created, the analysis starts in bottom-up mode. Initially, all abstract syntax graph objects schedule level 1 rules, i.e., those depending on abstract syntax graph objects only. Scheduling initially only level 1 rules is sufficient to ensure that all necessary rule applications are eventually considered. It avoids many top-down failures that would otherwise occur, because the information available is not enough to establish a high level rule. Consider, for example, a Duplication rule scheduled by a single class or attribute. The inherent search space is too large to justify its top-down investigation.

An object *o* scheduling a rule *R* creates a rule/context pair *R(o)* which is added to a bottom-up priority queue held in descending order of rule level number. The use of rule level numbers to order the rule/context pairs in the bottom-up queue is not critical. Any ordering that promotes higher-level rules will do. This fact can be exploited to further tune the algorithm [NSW+02].

The algorithm continues in bottom-up mode by dequeueing the first rule/context pair, in our example INSERT(i1), cf. Figure 3.32. This rule is immediately applicable, so an INSERT annotation i is created, annotating the attribute a1, which is accessible via the INSERT object i1. In contrast to abstract syntax graph objects, which schedule level 1 rules

only, creation of i schedules all rules depending on the INSERT rule, e.g., the top-level Duplication rule.

Since Duplication is a top-level rule, the pair Duplication(i) is taken next from the bottom-up queue. At this point, however, Duplication(i) cannot be applied successfully, since annotations have yet to be created by the other rules on which Duplication depends (SELECT and AttrSim).

Top-down strategy

When a rule that depends on other rules cannot be applied in bottom-up mode, the algorithm switches to top-down mode, which uses a separate top-down priority queue. The top-down strategy tries to make the other rules create the missing annotations based on currently available information. In this case, the search space is quite strictly delimited by the information available, e.g., that inherent in the i:INSERT.

Consideration of Duplication(i) in top-down mode thus schedules the SELECT and the AttrSim rules, cf. Figure 3.32. Where such rules depend on other rules, rule scheduling continues recursively. In our case, for example, the AttrSim rule now schedules NS rule and TC rule, as Figure 3.3 implies.

To establish a Duplication consistent with i, the relevant context for the SELECT rule is the INSERT annotation i. For the AttrSim rule the attributes a1 and a2 obtained from i and s, respectively, are an alternative context, which have to be considered.

The rule/context pair at the front of the top-down mode queue is not dequeued if the rule involved schedules other lower-level rules. Instead, pairs added to the top-down queue are queued in ascending order of their level number. This means that the higher-level rule will be reconsidered after the lower-level rules on which it depends (if these succeed). Using a priority queue rather than a stack means that the top-down algorithm goes as far down the abstract syntax graph as quickly as possible. This encourages earliest possible failure in top-down mode, while maintaining an appropriate sequence of rule applications for top-down success. If a rule marked as recursive is added to the top-down queue, however, stack behaviour is adopted until all rules marked as recursive have been removed from the stack/queue.

When the rule at the front of the top-down queue can be applied, a corresponding annotation is created, all dependent rules are scheduled for bottom-up consideration, the front entry of the top-down queue is dequeued and the next element of the top-down queue is considered. The newly scheduled rules join the bottom-up queue since they represent analysis results that would have been created later in bottom-up mode anyway and need further investigation.

The algorithm runs in top-down mode until the top-down queue is empty or a rule in the queue fails with no alternative contexts left to explore. The first case means that the rule that started this top-down phase has been successfully applied, in our example the Dupli-

cation rule. In the second case the starting rule cannot be applied in the given context. In either case the algorithm switches back to bottom-up mode.

Intermediate results

With the algorithm as described, each annotation once created represents an intermediate result that is not affected by subsequent analysis. In principle, therefore, the execution can be interrupted for inspection of results by the engineer at any stage. In practice, however, it is illogical to allow interruption during a top-down interlude, when some but not all of a closely related set of annotations may have been created.

Since the algorithm tries to establish high level rules using the top-down strategy, the intermediate results are likely to be useful information for the reverse engineer, e.g., redundancy dependency patterns. The engineer reviews such patterns to determine if the analysis should continue on the current basis. The algorithm is also robust to certain changes by the engineer prior to resumption. Addition of annotations by the engineer is valid at this stage, provided these add all corresponding rule/context pair for dependent rules to the bottom-up queue. Marking a rule as 'to-be-deleted'[1] is also acceptable, as the consequences of deletion can be systematically propagated to both the results to date and the resumed analysis. Such actions may be useful to the engineer as 'proofing actions' prior to permanent change to the rules themselves. Any addition or modification to the rules, however, invalidates the analysis to date and requires restart of the overall analysis.

The overall analysis finishes when the bottom-up queue is empty. In this case the algorithm has analysed all abstract syntax graph objects and created annotations on the objects for all rules that could be applied.

Integration of the reverse engineer

Integrating the described analysis algorithm into a semi-automatic reverse engineering process is easy because it is interruptible. Figure 3.33 shows our pattern retrieval process as a statechart. The process starts by creating abstract syntax graph representation, followed by loading a particular pattern catalogue. The engineer can then make initial modifications before starting the analysis algorithm by sending a start() event.

The complex state on the left-hand side with its two internal states bottom-up strategy and top-down strategy represents the analysis algorithm described above. The algorithm halts, and the reverse engineer can look at the results, if the algorithm has finished or the reverse engineer interrupts the execution by sending a stop() event. As mentioned above, it is logical to confine such interruptions to bottom-up mode purely for pragmatic reasons.

The reverse engineer then has the opportunity to look at the results to see if the patterns selected still seem appropriate. By sending an adapt() event, he/she can mark patterns for deletion or create annotations that will steer the algorithm to a part of the source code that

1. Note that the rules are not deleted but only marked as 'to-be-deleted' for exploration.

he/she wants to have analysed. Resuming rather than restarting the algorithm systematically propagates the consequences of such changes to both the prior and subsequent analysis. If the analysis to date fails to meet the engineer's needs in other ways, the patterns can be modified, but in this case the whole analysis must restart from the beginning.

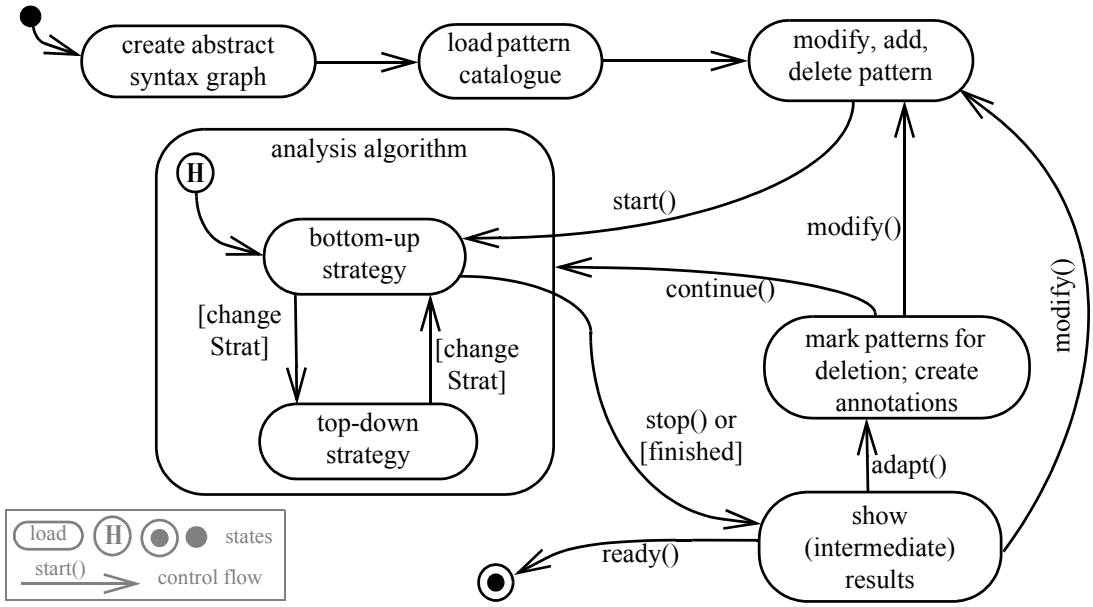


Figure 3.33: Retrieval process statechart

3.3.4 Handling Uncertainty using Fuzzy Beliefs

The performance of the retrieval process is crucial regarding the involvement of the reengineer in the process. The presented retrieval process solves the performance problem only partially. As mentioned before large abstract syntax graphs, i.e., a large search space, creates this performance problem. The large number of graph productions describing precisely a pattern intensifies the performance problem. Reducing the search space means that the system is only analysed partially, i.e., information is lost [NWW03, Nie03]. Reducing the number of graph productions means to reduce the number of patterns covered and thus get incomplete results. Identifying common parts of different graph productions and replace them by one graph production leads to many false positives. The reengineer is then not longer able to distinguish correct results from false positives.

Our approach is to allow reducing the number of graph productions and therefore get imprecise results but also allow expressing the degree of impreciseness by assigning fuzzy beliefs to the graph productions. In addition to the fuzzy results, our approach allows to filter the results with fuzzy values higher than a certain threshold. This enables the reengineer to profit from appropriate time-limit results and valuing the found matches of a pattern. Our experience shows that even precise pattern definitions produce false positives. Such this approach is a trade of between the number of graph productions and the number

of false positives to produce rapid intermediate results with the benefit that the uncertainty is presented to the reengineer.

Indicator Impreciseness

Another reason for adding impreciseness to graph productions is the impreciseness of indicators that can be found in the code. This uncertainty problem of indicator-based retrieval processes is discussed [Jah99] in the context of (single) database reengineering processes. The code example 2 of Figure 3.34 shows such indicator impreciseness for a Duplication pattern. The two INSERT statements get the same value :DD and are preconditioned to be „close“ to each other, i.e., within the same transaction boundaries. Nevertheless we do not have the certainty that the value of :DD is not changed between the two statements.

```
// example 1
SQL INSERT INTO bvmtdata.Deceased (DEATHDATE) SELECT Patient.(DEATH DATE) FROM
hospdata.Patient WHERE Patient.VHS_ID# = Deceased.VHS_ID#;
...
// example 2
SQL INSERT INTO bvmtdata.Deceased (DEATHDATE) VALUES :DD;
...
SQL INSERT INTO hospdata.Patient (DEATH DATE) VALUES :DD;
...
// example 3
DD = „SQL SELECT Patient.(DEATH DATE) FROM hospdata.Patient WHERE Pa-
tient.VHS_ID# = PatienID;
SQL INSERT INTO bvmtdata.Deceased (DEATHDATE) VALUES :DD;
...
```

Figure 3.34: Code samples for duplication

Figure 3.34 shows three examples of possible code fragments that indicate duplication. In each example the DEATH DATE of Patient is duplicated into DEATHDATE of Deceased. These three occurrences of a duplication are covered by the graph productions of Figure 3.31 and Figure 3.35.

We introduce a so called fuzzy belief for each graph production that expresses its preciseness. The fuzzy belief of a rule is a value between 0 and 100. By this value, the reengineer expresses his estimation that, e.g., 20% of all matches are false positives. Thus, 80% of all matches would be correct and the value would be 80. In Figure 3.31 and Figure 3.35 the fuzzy belief is defined in the green Duplication annotations that are created when the graph productions are applied. The fuzzy belief is the first value in the pair of numbers; the second is explained in the adapted analysis execution. Note that a fuzzy belief of 100 for all graph productions is equivalent to the retrieval process where uncertainty is not considered like described in Section 3.3.3.

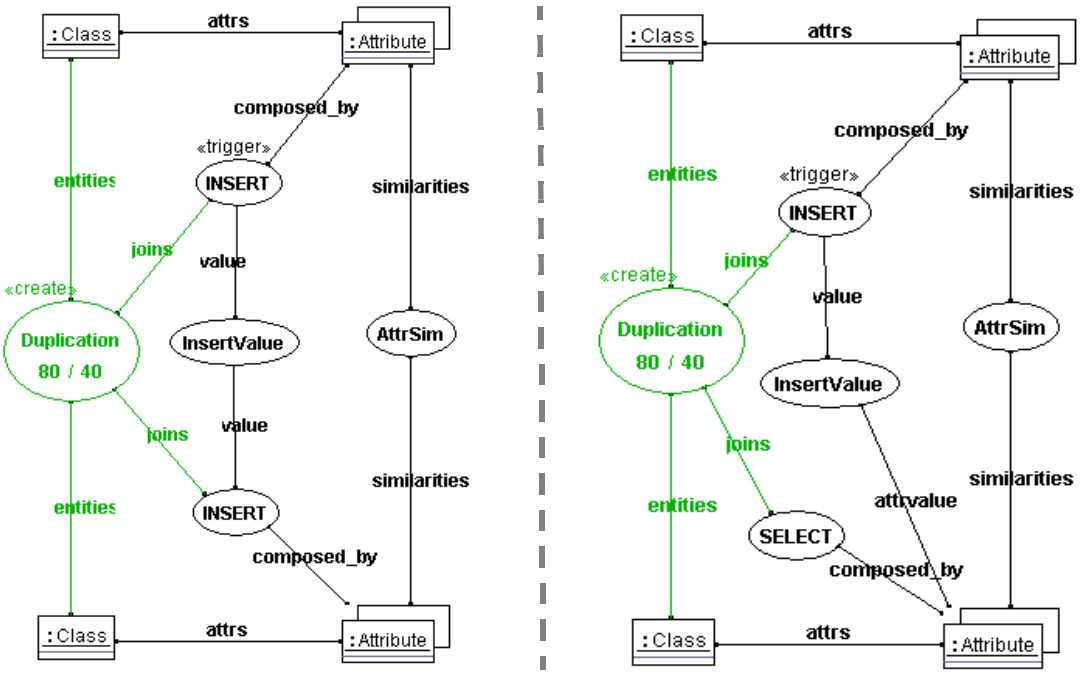


Figure 3.35: Alternative duplication pattern definitions

Managing Uncertainty

An exact graph production for each implementation variant has the benefit of few false positives in the analysis results, but it leads to performance problems because of the high number of rules. We introduce further abstraction into the rule definitions so that one or only few graph productions cover a large variety of implementations. This reduces the number of rules and improves the performance. To define a new graph production with further abstraction the reengineer, first, chooses a set of implementations or pattern definitions that should be covered by the new rule. Common parts have to be identified. The common parts, then, form the new graph production. All implementations or patterns from the set are then at least covered by this new graph production. In addition, there will be implementations or patterns that were not intended to be found by the rule. Some of them are not instances of the searched pattern, i.e., false positives. Others are correct matches that were not considered when defining the graph production. This is a kind of uncertainty that has also to be managed.

Figure 3.36 depicts a new graph production for Duplication that is designed to replace all three graph productions from Figure 3.31 and Figure 3.35. Therefore, the definition of INSERT and InsertValue were revisited. The nested INSERT / SELECT of example 1 as well as the examples 2 and 3 in Figure 3.34 are covered by imprecise InsertValue graph

productions. Consequently, all three examples are covered by the new INSERT definition and thus by the Duplication graph production.

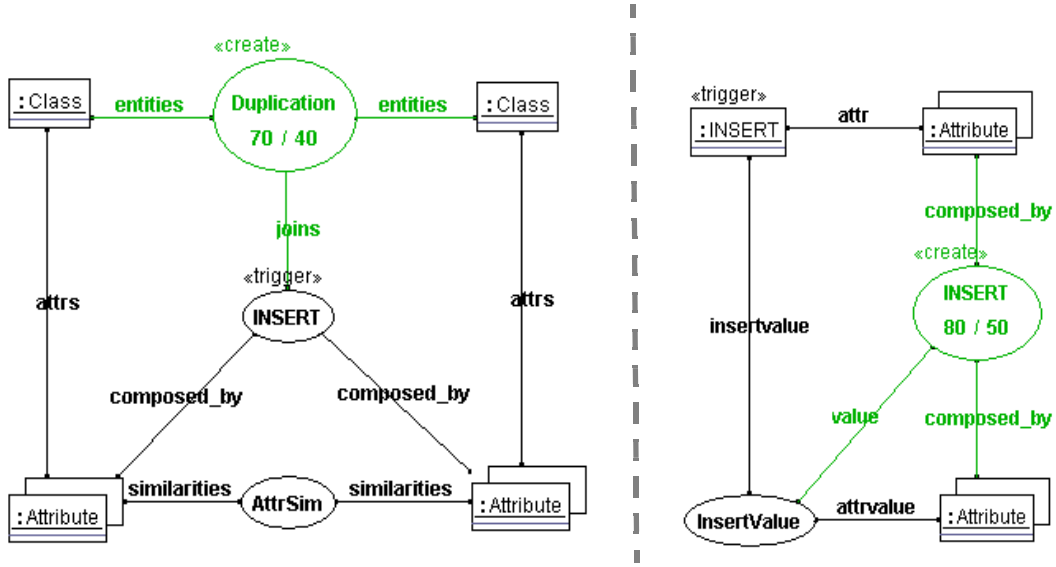


Figure 3.36: Imprecise duplication and insert pattern definitions

The impreciseness of a graph production that stems from reducing the number of rules has to be valued to be useful to the reengineer. The value should describe the ratio between correct matches and all matches of a certain rule including false positives, i.e. the preciseness of a rule. After all applications of a rule, this ratio can be calculated. At the time of the rule definition the number of false positives produced by the rule can only be estimated.

In our example we estimate that 30% of all matches for Duplication are false positives. We reduced it compared to the three „precise“ graph productions. Only the indicator impreciseness was considered so far. Thus, the ratio is 70. For the INSERT annotation we estimate the ratio to 80.

Adapted analysis execution

In the following we revisit a part of the algorithm to show how the fuzzy values of the results are calculated and influence the analysis execution. The graph productions create annotations with a certain fuzzy value. The fuzzy value is calculated during rule application for each match of the graph production. They are stored in the fuzzyValue attribute of the Annotation nodes (cf. Figure 3.23). The fuzzy beliefs of the graph productions are the basis for the fuzzy value calculation. The fuzzy value of a match is computed as the minimum of the fuzzy values of all annotation nodes occurring in the match and the fuzzy belief of the rule [Jah99, Nie03]. Thus, calculation of the fuzzy values is similar to fuzzy grammars, cf. [Zad65].

To apply the Duplication graph production of Figure 3.36 two annotations (:INSERT and :AttrSim) are needed. The fuzzy belief of the Duplication graph production is 70. In Figure 3.37 a cut-out of the graph during an analysis execution is shown. The INSERT graph production has created an annotation i with value 80 on insert i1. Upon this annotation i the Duplication graph production has created annotation d. Since a Duplication also needs an AttrSim annotation top-down analysis is performed to detect a with value 95. A fuzzy value of 70 (the minimum of 80, 95 and 70) is assigned to the annotation node d. The fuzzy belief of 70 of the Duplication graph production is the minimum, compared to 80 and 95 of the other two annotations. Thus, the fuzzy belief of the graph production is an upper bound for the fuzzy values computed.

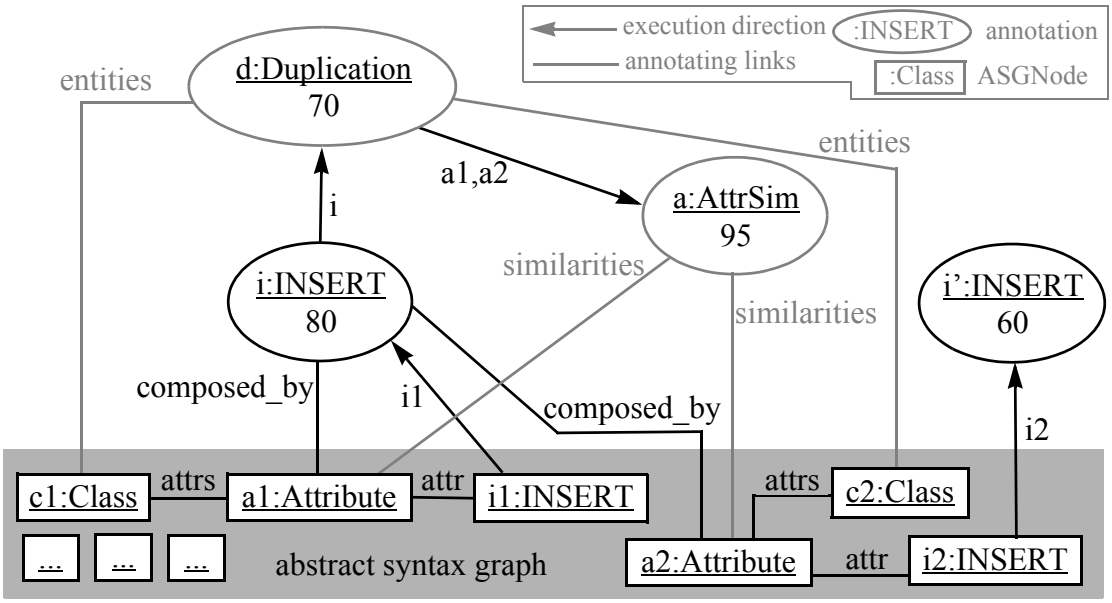


Figure 3.37: Sample analysis execution with fuzzy values

A node in the graph production may have multiple valid matching annotation nodes in the host graph that have different fuzzy values. In that case the annotation node with the maximum fuzzy value is chosen as match such that the new annotation created by the graph production application uses the most reliable source of information. Assume that later a second annotation i':INSERT with value 60 is detected. This annotation would not be considered furthermore because it has a fuzzy value of 60 which is lower than the fuzzy value 80 of annotation i used in the match of the Duplication. Therefore, the fuzzy value of an annotation corresponds to the maximum fuzzy value of all derivations of the annotation, cf. [Zad65].

As a way to limit the graph production applications to reasonable cases we introduced thresholds, cf. attribute threshold in Figure 3.23. In Figure 3.36 the threshold is defined in the grey (green) annotation nodes that are created when the graph productions are applied.

It is the second value in the pair, i.e., 40 for Duplication and 50 for INSERT. In our sample analysis execution of Figure 3.37 the value of the annotation i is 80. The annotation d is created because 80, as well as 95 later for a, is greater than 40 (the Duplication threshold).

If any annotation that is part of the subgraph to match has a fuzzy value lower than the threshold, the graph production would not be applied. This helps to minimise computation time and memory resources that would otherwise be used for investigation of unreliable and imprecise information. Thus, the thresholds improve the scalability of our approach. They are chosen by the rule developer based on personal experience and/or historical data.

Note that the threshold of the Duplication (40) is lesser than the threshold of INSERT (50). Defining the threshold of a subpattern (INSERT) higher than the threshold of a pattern (Duplication) based on this subpattern makes only sense if the pattern is based on another subpattern (AttrSim). In case that the annotation a would have a fuzzy value of 30 the Duplication is not created because the value 30 is lesser than the Duplication threshold of 40, but a does not affect the INSERT subpattern with threshold 50.

Tuning Fuzzy Beliefs and Values

One problem remaining in this incremental semi-automatic pattern instance retrieval process is the exact choice of the fuzzy beliefs and thresholds of the graph productions. Both values are estimated by the reengineer or at the best adapted manually over the time by the reengineer during the incremental process. In [Nie03] an automatic adaptation of the fuzzy beliefs and thresholds is presented. This adaptation is based on the comparison between the given values and calculated values from pattern instance occurrences based on the corrections of the reengineer.

The impreciseness of the indicators can be further reduced. Therefore validation of detected relationships, which results in the diminution, confirmation, attenuation or augmentation of the associated fuzzy values, is done by several satellite activities.

One possibility to verify assumptions is looking at the data, e.g., checking a primary key by validating the uniqueness of each data column. Assume that a column, which is annotated to be a primary key, contains identical values several times. This indicates that the primary key assumption may be false. The number of counterexamples relative to the number of present data determines the fuzzy value in this case, cf. [Jah99]. Note that if no counterexample is found for enough present data the fuzzy value will be augmented.

Finally, the reengineer has to accept or reject the annotations [Jah99, Wen01, Nie03]. An annotation that is confirmed by the reengineer is attributed a fuzzy value of 100. In case of rejection the fuzzy value is set to 0. For this activity of confirming or rejecting annotations, the reengineer has the possibility to look into the code. Slicing the involved entities for a recovered relationship shows uses of the entities different from pattern instances indicators. Some code fragments may not be covered by the island grammar parser and/or the pattern instance retrieval and can therefore provide further information to the reengineer.

Figure 3.38 shows some annotations from the examples of Section 3.3. We see the Association between Patient and tblOutcomes (Figure 3.29) and the Replication between DrugProfile and tblDose (Figure 3.30). The Duplication between Patient and Deceased (Figure 3.34 and Figure 3.36) is Confirmed by the reengineer. The association profile was retrieved as R-IND in the logical schema and mapped to the conceptual schema.

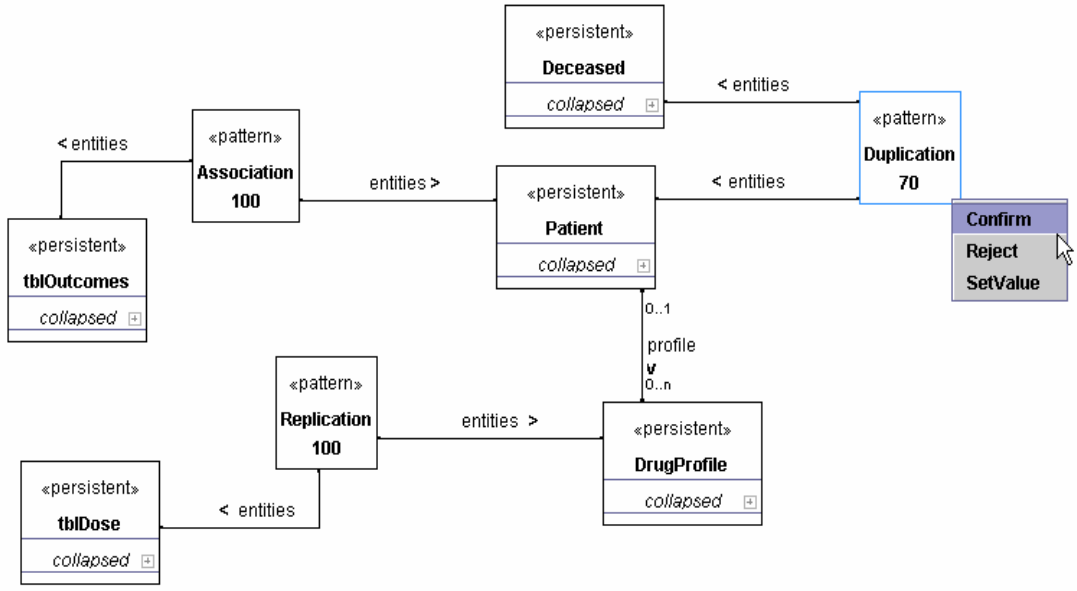


Figure 3.38: Annotated conceptual view

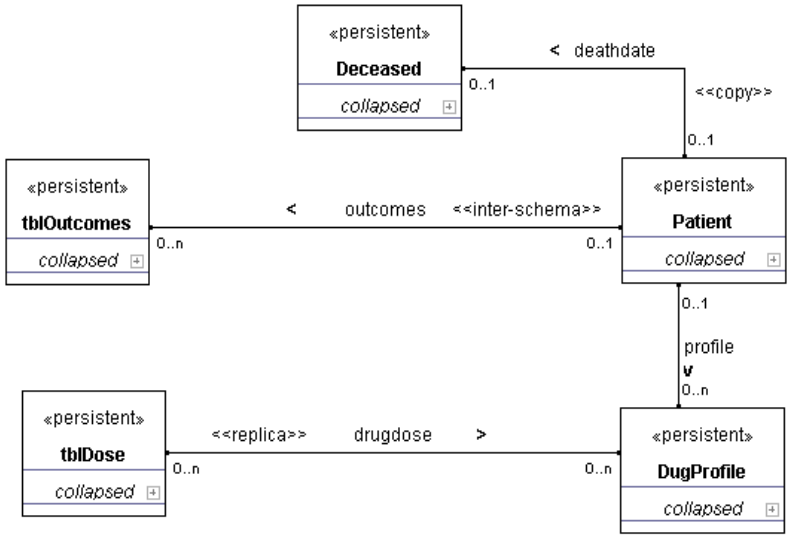


Figure 3.39: Conceptual schema view (data dependencies)

Finally, when the reengineer confirms a data dependency pattern instance it is transformed into an association with the corresponding stereotype, cf. Figure 3.39. Note that the reengineer has the possibility to confirm all pattern instances greater or equal a customisable fuzzy value.

3.4 Tool Support

The tool support for the data-oriented reengineering process is located in the REDDMOM project. REDDMOM is mainly based on the FUJABATS and part of the FUJABA Reengineering tool (FUJABARE). FUJABATS has a plug-in architecture allowing developers to add functionality easily while retaining full control over their contributions. This plug-in architecture enables extension and integration at the meta-model level. Details can be found in [BGN+03].

REDDMOM (and FUJABARE) uses this plug-in architecture to reuse and combine existing tools. Figure 3.40 shows an overview of the REDDMOM tools for data-oriented reverse engineering. Several figures in the precedent sections show screenshots from the different tool user interfaces. The grey rectangles represent tools from other universities that we use. Dashed rectangles express that the tools in question are (largely) generated. The tools inside the grey area were developed in FUJABARE. The remaining tools were constructed in REDDMOM.

Data Model Parser

We use the JDBC API as interface to parse the physical data models of the different data sources. Presumed that a JDBC driver exists for a data storage facility the Data Model Parser can parse the data structure. Note that meta-data can only be parsed if they were introduced in the data repository. An XML parser and especially an XMLSchema parser is currently inserted in FUJABARE. From the output of these parsers the corresponding abstract syntax graph is directly created.

EER Editor

The EER Editor represents the logical (schema) abstract syntax graph in an EER diagram, i.e., entities, attributes and relationships. We added specific data reverse engineering constructs, e.g., views or variants, to the editor. It is similar to the analysis Front-end of the VARLET ANALYST [Jah99]. The schema elements and detected pattern annotations are represented. The reengineer can apply schema transformations (e.g. createRIND) and annotation related operations (e.g. confirmAnnotation or setFuzzyValue).

Triple-Graph-Grammar Editor

To provide inter-model consistency between the logical and conceptual schema we use triple-graph-grammar rules, cf. Section 3.2.3. These rules can be defined in the Triple-Graph-Grammar Editor. From these rules in a first step story diagrams are generated and

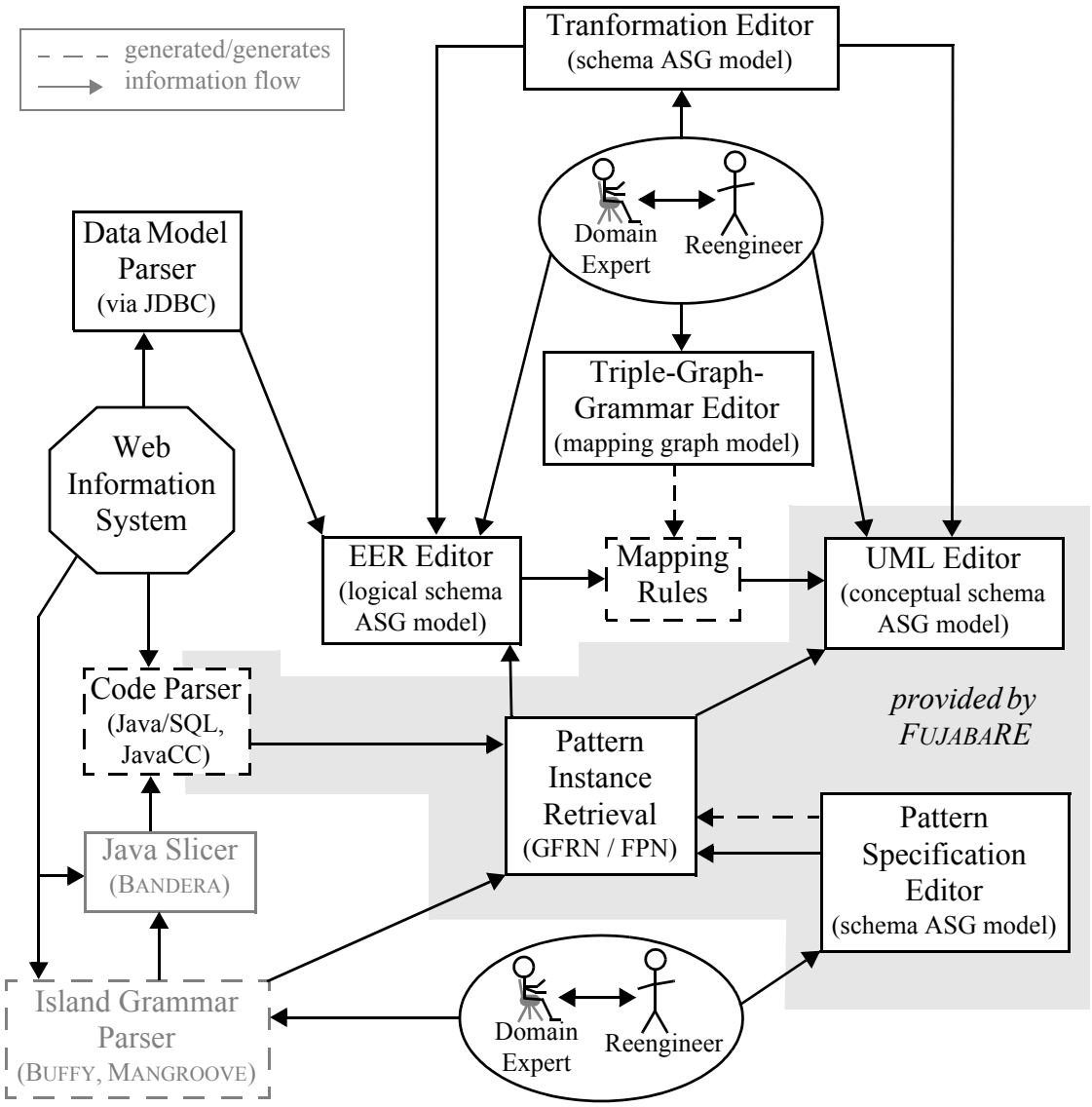


Figure 3.40: Architecture of the REDDMOM reverse engineering tools

in a second step Java code is generated with FUJABARE, see Figure 3.12 to Figure 3.17. The mapping graph model (Figure 3.11) connects the logical and conceptual model via the FUJABATS meta-model integration pattern [BGN+03].

Mapping Rules

The Mapping Rules are the generated Java code from mapping rules defined as triple-graph-grammar rules in the Triple-Graph-Grammar Editor. For runtime optimisation we ordered the rule application sequence and assign each rule a context with an input parameter, i.e., a node of the abstract syntax graph where to start the sub-graph matching.

Transformation Editor

The reengineer can define basic graph transformations with REDDMOM, cf. Figure 3.18. The Transformation Editor enables the reengineer to construct composed graph transformations out of basic or composed graph transformations. In REDDMOM the Transformation Editor permit the construction of graph transformations for the logical as well as the conceptual schema, cf. Figure 3.41. Wildcards are used to assign the parameters of the SplitClass operation to the parameters of createClass, createAssoc, moveAttribute, etc., cf. Figure 3.19.

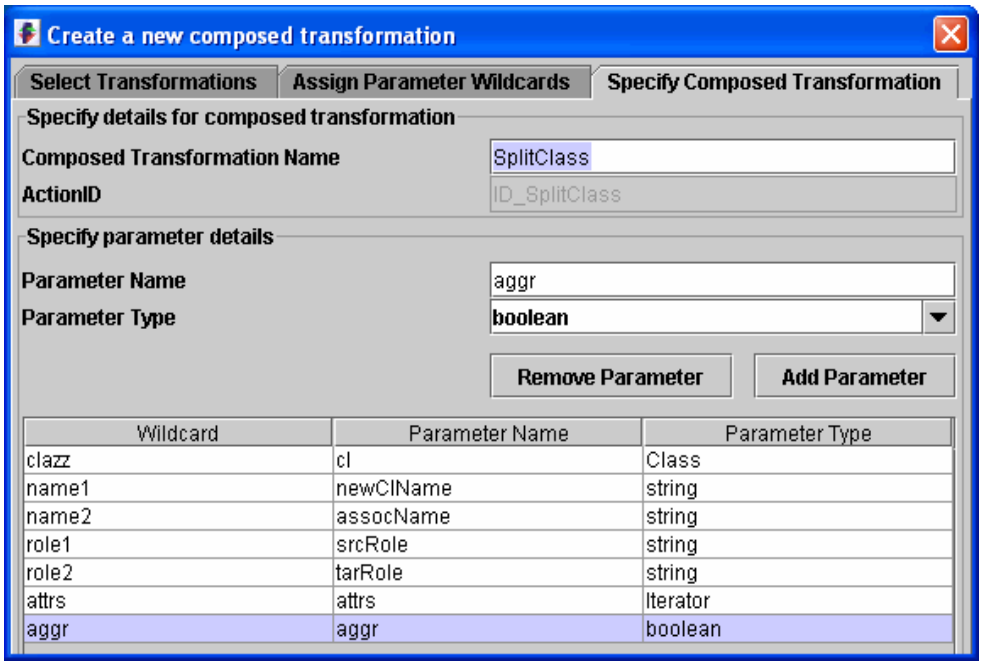


Figure 3.41: SplitClass composed transformation example

Code Parser

The abstract syntax graph is produced by the JavaCC source code parser [JCC]. Our Code Parser can read SQL and Java. The Java parser stems from FUJABARE and REDDMOM added the SQL parser. The Java parser is incremental, i.e., classes, attributes and method headers are directly parsed and presented in an UML class diagram. The method bodies are only parsed on demand when a user wants to see a method body. Both parsers are currently adapted to incrementally parse files, like described in Section 3.3.1. We refer to [Sch01] for incremental parser details.

UML Editor

The UML Editor covers UML class diagrams, views [Rec01] and story diagrams [FNTZ98] with the respective functionalities. It is the core of FUJABATS together with the Java code generation. REDDMOM added stereotypes for associations to the class diagram. The story patterns, i.e., the activities in story diagrams, were extended with a path construct.

Pattern Specification Editor

The Pattern Specification Editor was developed in [Pal01] and part of FUJABARE. In Section 3.3.2 several pattern definitions are shown. Further, the Pattern Specification Editor contains a pattern catalogue, that is automatically created during pattern specification, which shows the dependencies between the pattern graph productions, cf. [Nie03].

Pattern Instance Retrieval

The Pattern Instance Retrieval tool is based on the Generic Fuzzy Reasoning Nets (GFRN) [JSZ97, Jah99]. The analysis execution takes place on Fuzzy Petri Nets [KM96]. Details of the implementation can be found in [Wen01] and [Nie03]. This tool is part of FUJABARE and runs the algorithm presented in Section 3.3.3 and Section 3.3.4.

Island Grammar Parser

Island Grammar Parsers can either be generated by MANGROOVE or BUFFY. Both tools are prototypes from other researchers; we present them shortly.

Mangrove [Moo01] is a parser generator that takes an island grammar in SDF [HHKR89] format and produces an extractor for this (partial) language. The definition of an island grammar requires an intimate understanding of the concept of island grammars and is a highly explorative process. Still, simple island grammars can also produce good results.

BUFFY [Chu04] generates extractors based on code fragment examples. The user introduces an interesting code fragment into BUFFY. Then, BUFFY suggests a set of island productions for this instance. Subsequently, the user can interactively correct and refine these productions to characterise the associated island. Then, the user can run the generated extractors against the code and verify the island definition. Depending on the result of this verification step the user might iteratively refine the description of the island.

Java Slicer

BANDERA [HDZ00] is a Java model checker which incorporates a Java Slicer. This slicer enables the detection of method calls of code fragments of interest such as described in Section 3.3.1 and the slices for entities involved in a retrieved relationship, cf. Section 3.3.4.

3.5 Related Work

Considerable effort has been made to develop concepts and methods to reverse engineer legacy information systems. Some of these methods have been implemented in computer-aided reengineering tools to automate laborious activities and reduce the complexity of the data reengineering problem. As the persistent data structure is the central part of a legacy information system [Aik96], many approaches focus on

- data model (database schema) analysis and/or
- model (schema) translation and transformation.

Most existing approaches to data model analysis aim to recover a complete model (logical schema), e.g., [PB94, SLGC94]. Premerlani and Blaha propose a set of simple, loosely coupled tools for textual search and data analysis, e.g., grep, awk-scripts, and predefined database queries [PB94]. Signore et al. present a knowledge-based approach based on backward reasoning to infer schema constraints from collected indicators [SLGC94]. An example for an approach that annotates schemas semantically is presented in [RH96, RH97] based on the idealistic assumption of a structurally complete schema description [HCTJ93].

Various algorithms for canonical translation of logical to conceptual schemas (e.g., [BDH+87, NA87, JK89, MM90, SK90, And94, PKBT94, MCAH95, Fon97, RH97]) are fully automatic approaches and often make unrealistic assumptions about the quality of the legacy system, like structurally complete schema description. All these approaches provide very little support for iteration and in particular lack the ability to detect and propagate inconsistencies between a (modified) legacy system and its representation.

Vossen and Fahner suggest combining an initial automatic translation with a subsequent manual transformation phase but do not provide any tool support for this human-intensive activity [FV95]. In [BGD97], Behm et al. propose an interactive schema migration environment that provides a set of alternative schema mapping rules. In this approach, which is similar to the migration environment presented in [JSZ96], the reengineer repeatedly chooses an adequate mapping rule for each schema artifact that has to be mapped. It turned out that the set of rules become very large and that a transformation phase after an initial translation phase is much easier for the reengineer. Jeusfeld and Johnen propose an approach to schema migration that employs a generic meta-model as mediator [JJ94]. The advantage of this approach over direct translation is questionable as it was only evaluated for translation of relational schemas to ER schemas.

The observations described by Blaha and Premerlani motivate user-centric, interactive reverse engineering approaches [BP98]. Indeed, one of the most important limitations of most data reverse engineering tools is that they do not consider the evolutionary and exploratory nature of the reengineering process [HEH+95]. Therefore, Hainaut et al. skip the initial translation step completely and also use a common generic data model that subsumes conceptual constructs as well as logical (and physical) constructs [Hai89,

HHH+96]. In the DB-MAIN tool [EH99], the reverse engineering process is invoked by predefined scripts which look at the application code [HHH+99] to extract data structures. Data depenpency elicitation is supported through variable dependency graph and program slicing [HH01]. Based on the common data model, they have defined a catalogue of schema transformations which are used to gradually replace low-level implementation constructs by more abstract concepts [HTJC94]. However, the execution of in-place transformations impedes the iterative process because the original (logical) schema implementation is lost during the process.

A notable exception is VARLET [Jah99] that covers all phases of database reverse engineering from schema recovery up to building a conceptual schema. In VARLET an interactive process to handle uncertainty and inconsistency during recovery of information models (comprising relationships) is based on Generic Fuzzy Reasoning Nets [JSZ97] which revert to code and data analysis. Moreover it considers variants structures that are largely used in forward engineering [BCN92, HHEH96] during analysis. This analysis phase is followed by an initial automatic translation and schema transformation. It provides many adapted conceptual redesign transformations proposed by [BP98]. The VARLET approach is limited to single schemas.

Valuable stand-alone approaches exist that focus on the retrieval of semantic constraints. Relationship retrieval based on stereotypical (code) patterns are presented in [PKBT94, PTBK96] and [And94]. Soutou presents an algorithm to recover n-ary associations [Sou98b] followed by a method to retrieve aggregations [Sou98a]. In [BR97] methods from the inductive logic programming domain are adopted to detect relationships.

All these data reverse engineering approaches are limited to one model (single schema) analysis and do not consider distributed schemas. They lack the flexibility for recovering inter-schema relationships. An overview of reverse engineering methods and tools that can further be used and adapted for data reverse engineering is given in [MJS+00]. The field of recovering inter-schema relationships is poorly explored.

A catalogue for inter-database/-schema dependencies has not been published yet. Rusinkiewisz et al. [RSK91] present two examples of interdatabase dependencies: (1) replicated data that is characterised as "identical copies of data in two or more databases" for which "we can tolerate inconsistencies (...) for no more than one day" by the authors; (2) existential constraints which are, e.g., referential integrity constraints requiring immediate updates. Both correspond to the redundancy dependency with respect to copies that have to be kept consistent. Our approach also discovers possible redundant data by duplicated schema elements.

A theory of attribute equivalence in databases on a semantic basis is presented in [LNE89]. The approach uses semantic attribute equivalence for integration of database schemas. Therefore, the characteristics of the attribute equivalence are very detailed and restrictive. In contrast to schema integration, schema reverse engineering needs flexible and general attribute property (characteristic) definitions.

Identifying and solving conflicts in inter-schema knowledge in cooperative information systems has been presented in various references, e.g. [BLN86, CL93, TGF00]. In these approaches the discovery and representation of inter-schema assertions is studied to "make explicit the knowledge which a human integrator uses implicitly to identify semantic similar schema concepts" [TGF00]. This is different from the inter-schema knowledge, i.e., explicit dependencies between the distributed databases, we retrieve.

Several approaches exist for pattern recognition. Harandi and Ning [HN90] present program analysis based on an Event and Plan Base. The analysis process starts by firing rudimentary events constructed from source code. Plans define the correlation between one or more (incoming) events and they fire a new event corresponding to the intention of the plan. Each plan definition corresponds to exactly one implementation variant, which leads to a high number of definitions. This applies also to the approach of Paul and Prakash [PP94], where a matching algorithm for syntactic patterns based on a non-deterministic finite automaton is introduced.

An approach to recognise clichés, i.e., commonly used computational structures, is presented in [Wil96], within the GRASPR system. Legacy code to be examined is represented as flow graphs by GRASPR, clichés are encoded as an attributed graph grammar. The recognition of clichés is formulated as the sub-graph parsing problem which is NP-complete [Meh84]. This approach allows analysing only some thousand lines of code, which was sufficient to detect data structures or search and sorting algorithms. Applying the approach to larger programs had failed.

Krämer and Prechelt [KP96] use Prolog in order to detect design patterns [GHJV95] in C++ source code. The source code is parsed into facts and rules describing the relations. Prolog's execution mechanism applies the rules in arbitrary sequence and uses backtracking where necessary. The approach is able to analyse larger programs, but its preciseness is very low, because the approach uses information of header files only. An analysis of method bodies is not supported.

An approach producing more precise results is presented by Antoniol et al. [AFC98, TA99]. They use metrics, such as the number of method calls within a method body, to include a method body analysis without time-intensive graph productions. Unfortunately, the used metrics are inadequate to express detailed information, e.g. method calls within loops. Therefore, a lot of false positives remain.

Keller et. al. present an approach [KSRP99] to recover design patterns. Patterns are defined using UML and a pattern matching algorithm matches patterns on an abstract syntax graph representation of the source code, also using the UML notation. The matching process is executed using scripts and adoption of patterns is hard to follow especially when patterns are highly interrelated. In addition Seemann and von Gudenberg [SvG98] present an approach to recover design patterns starting with inheritance relations, call graphs, naming conventions, and programming guide lines. The pattern definition of higher order patterns allows a reverse engineer to compose patterns out of subpatterns and reduces

thereby the number of definitions. Both approaches are also feasible for the pattern based analysis task, but cannot deal with large programs.

Radermacher [Rad99] uses the graph rewrite system Progres [Zün95] to match patterns on the program. Patterns are defined as graph transformation rules and thus similar to ours. Radermacher uses the execution mechanism of the Progres environment, whereas the execution is not incremental.

3.6 Summary and Future Work

In this chapter we presented our data-oriented reverse engineering process to legacy (web) information systems based on existing approaches and tools. The chapter started with a process overview. Then we presented a classification for relationships of data component models in web information systems. After sketching our way of data component model representation, the process was presented. Based on the VARLET approach [Jah99] we presented the data model recovery that consists of schema recovery, hidden schema part retrieval, schema mapping and schema refactoring. The relationship retrieval started with code fragment extraction and parsing based on island grammars [vDK99] and slicing [Wei84]. The pattern instance retrieval, based on Niere's approach [Nie03], was presented in three steps: the pattern definition, the inference algorithm and the handling of uncertainty. Finally, we pointed out tools that support the iterative nature of this process.

The hard part of this process is the relationship retrieval. Several additional indicators can be used for tuning the certainty of the retrieved relationships. Data profiling tools, e.g., SAS/STAT®[1], dfPower® Studio[2] or Evoke Axio™[3], permit to find schema overlapping, schema incompleteness or redundancy. Rank aggregation techniques, e.g., like presented in [SA03], enables a relationship weighting. In analogy, the number of identical (similar) indicators can be used for tuning. This also holds for software clones. In case that we find multiple indicators for the same relationship but that the code fragments in question are clones, the fuzzy value may be adapted. Dead code analysis that finds out if the code fragments are still executed, may also be used to adapt the fuzzy value of a retrieved relationship. Note that relationships detected in dead code are also of interest and should not be rejected automatically. Finally, the pattern based recovery can be extended to dynamic analysis [Wen03]. For code fragment extraction the *Multi-Language Tool* [LCBO03], which detect dependencies between Java and C/C++ code, can be adapted to our purposes

1. http://www.sas.com
2. http://www.dataflux.com/
3. http://www.evokesoft.com

CHAPTER 4: DATA COMPONENT EXTENSION

He who moves not forward, goes backward.

JOHANN WOLFGANG VON GOETHE
German dramatist, novelist, poet & scientist (1749 - 1832)

4.1 Extension Approach

After understanding the web information system, the next step is to extend it in order to meet new requirements. Our approach focusses on the modelling of new applications and schemas, followed by the integration of these models. In Chapter 3 we describe how conceptual data component models of a legacy web information system can be retrieved. We base our extension approach on these retrieved models, i.e., models of the old data components, and the models of the new data components. Figure 4.1 shows the system parts considered for data component extension. The central parts are the conceptual models that permit, among other things, to generate access to the databases of the legacy web information system.

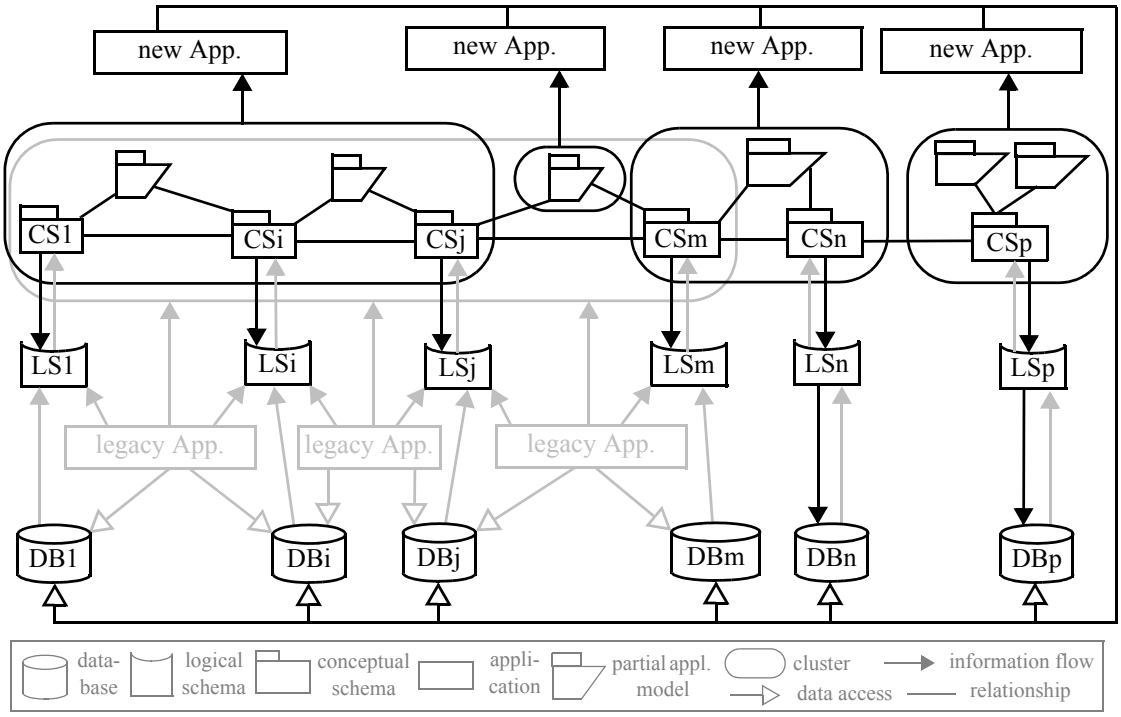


Figure 4.1: Web information system adaption

The process of data component extension consists of integrating the different retrieved parts and new parts to (clusters of) data components. The integration of data components requires the following activities: Data Component Clustering, Classification and (Re)Design, cf. Figure 4.2. These activities are preformed iteratively by the reengineer. Then, we (re)structure the web information system with architectural patterns. We conclude the integration with the Generation of a data access layer and Model Execution.

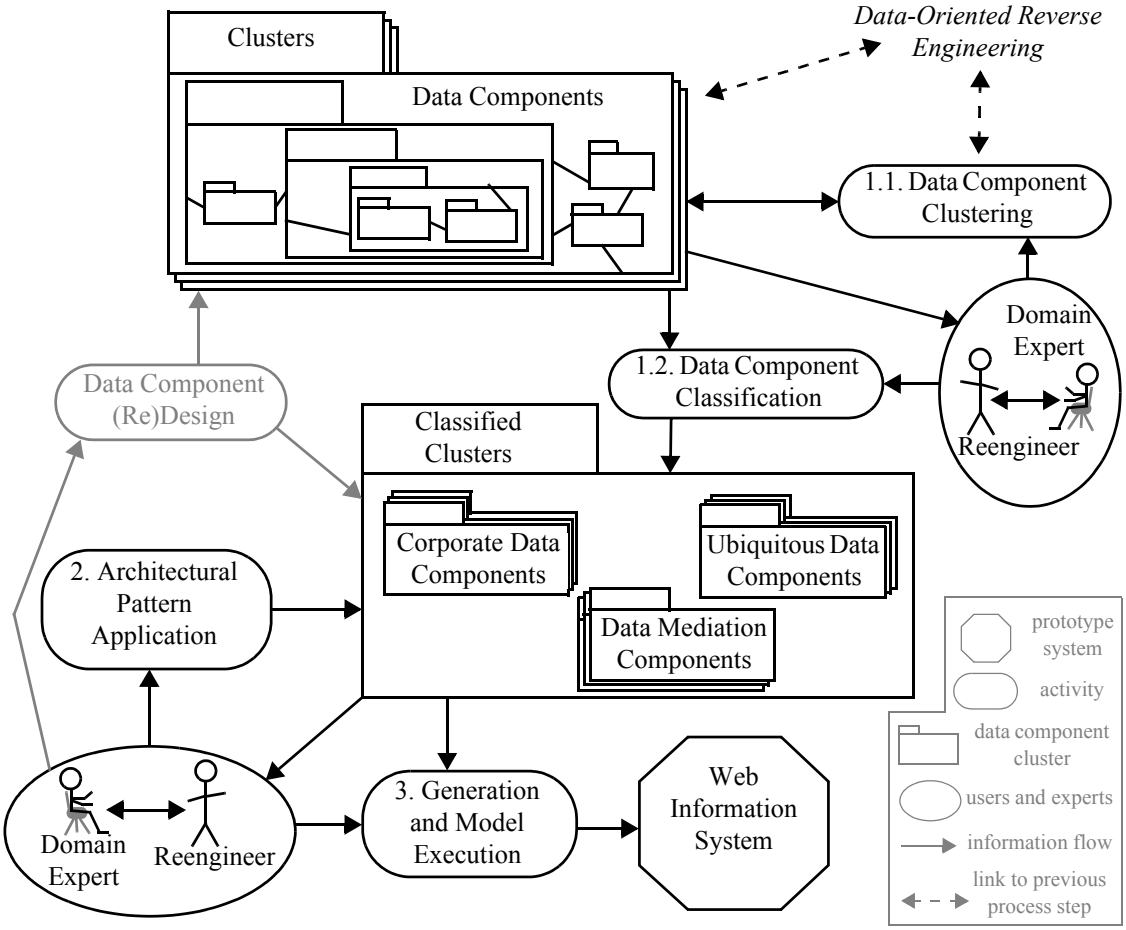


Figure 4.2: Data Component Extension: Overview

The Data Component Clustering is the starting point of our approach. We use the data components knowledge retrieved during the reverse engineering phase to cluster the system into data components. This supports the reengineer during data component integration.

Then, the data components are classified in either Corporate Data Components, Data Mediation Components or Ubiquitous Data Components. This classification helps to describe the data components' functionality and the actions they perform. Of course, during

the classification some data components may need to be adjusted concerning their cluster affiliation.

During Architectural Pattern Application the web information system is (re)structured following four architectural patterns we propose.

During these activities, the reengineer may redesign data components that exist. Data component parts, that are strongly related, have to be refined and / or adapted. Missing data component parts or data components which are needed for new functionality have to be designed. Data Component (Re)Design enables the reengineer to implement these parts with design patterns, like suggested by [GHJV95], and/or to use *(Re)Design Transformations* we propose. Since design patterns and model-driven design are well documented and understood, we pass on explaining it in this thesis and only consider it for tool support, cf. Section 4.5.

The retrieved models enable the Generation of object-oriented access layers for the legacy databases. Of course, this can also be done for new schemas/databases if they are modelled accordingly. Further, this approach enables the modelling of functionality such that data components can interact, i.e., exchange data. We propose to construct prototypes to validate the (re)design. If the reengineer follows this model-driven design approach, theses prototypes can be run by Model Execution.

4.2 Data Component Clustering and Classification

Before the reengineer designs new system parts, he may rearrange the existing web information system by splitting it up into clusters (data components) and classify them. Thus, the first activity of data component extension is data components clustering followed by their classification. Generally the retrieved system parts will not fit the reengineers mind model of the system. Thus, redesign takes place in parallel.

Before explaining the clustering, we clarify the membership of persistent classes to data components. A persistent class represents the data structure from its corresponding database entity, but a persistent class can exist and be deployed independently. This means that a persistent class and its corresponding database entity can be located in different data components. A persistent class can access the data via a defined interface. Consequently, persistent classes representing entities from the same database do not necessarily belong to the same data component, i.e., the same cluster.

4.2.1 Data Component Clustering

Resulting from the reverse engineering we have the logical schemas and the redesigned conceptual schemas, including the mapping from the logical schemas to the conceptual schemas. To facilitate (re)design we build data component clusters, which are classified in the next step.

These clusters enable different views on the system. Entity and schema comparison along the relationships, especially the redundancy dependencies, is possible. For example, strongly related entities of different schemas can be grouped to sustain schema integration. Further the classification of clusters is simpler and faster than the classification of each data component by itself. Finally, clustering helps to integrate new data components and thus identifying interfaces and connectors.

The advantages and benefits of clustering are primarily the semi-automatic aspect of the task and the endorsement of our iterative reengineering approach, i.e., repetitions of the same clustering tasks are allowed and wanted. Further, the clustering can be used for understanding by constructing views.

We use the BUNCH tool [MMCG99] to compute clusters. Therefore we will shortly introduce the clustering approach used in BUNCH. The basic assumption is that the modules and dependencies are mapped to a *Module Dependency Graph* (MDG):

> „ *MDG = (M , R)* is a graph where *M* is the set of named modules of a software system, and $R \subseteq M \times M$ is the set of ordered pairs $\langle u , v \rangle$ that represent the source-level dependencies (*e.g.*, procedural invocation, variable access) between modules *u* and *v* of the same system. “ [MMCG99]

In our case *M* is the set of named entities or data components (modules) and such $\langle u , v \rangle \in R$ are pairs of entities or data components that are related. Further in BUNCH it is possible to attribute a weight, in form of an integer, to each pair. The clustering problem is to find a „good partition“ of the MDG. A „good partition“ is the decomposition of all modules into disjoint clusters, where (highly) interdependent modules are grouped into clusters and in opposite (highly) independent modules are assigned to different clusters.

Based on our experience and the observations in [Lun98] difficulties and limitations of clustering activities can occur because:

- Reverse engineering does not generate precise and complete information, i.e., not all relationships are considered or identified
- Occurrence of omnipresent modules [MOTU93] or data components, i.e., data components that have much more connections than other data components and which seem not to belong to any cluster
- Some data components perform a specific task and are comparatively small to other data components
- The existing partitioning was not well done by the designer(s)
- Inappropriate choice of clustering techniques

These limitations coincide with the limitations that were detected in a first version of BUNCH and are mainly resolved in the BUNCH version we use [MMCG99]. Two lists of omnipresent modules, one for clients and one for suppliers, can be specified either manually or automatically. These client and supplier modules are assigned to two separate

clusters. Further, user-specified clusters enable the handling of predefined (by domain-knowledge) or existing clusters. Optionally the locking of these clusters can be enabled and the addition of modules to the user-specified clusters is possible. To handle the integration of new modules or when modules undergo structural changes the orphan adoption technique [TH97] was introduced to BUNCH. Scalability and performance problems are minimised because of the predefined clusters, e.g., conceptual schemas, and algorithms for sub-optimal results, i.e., hill-climbing and genetic algorithms. For more details we refer to [MMR+98, MMCG99].

Figure 4.3 shows an excerpt of the palliative care conceptual schema, which contains 66 classes and about 100 relationships. The different relationships are marked by stereotypes, except the intra-schema associations.

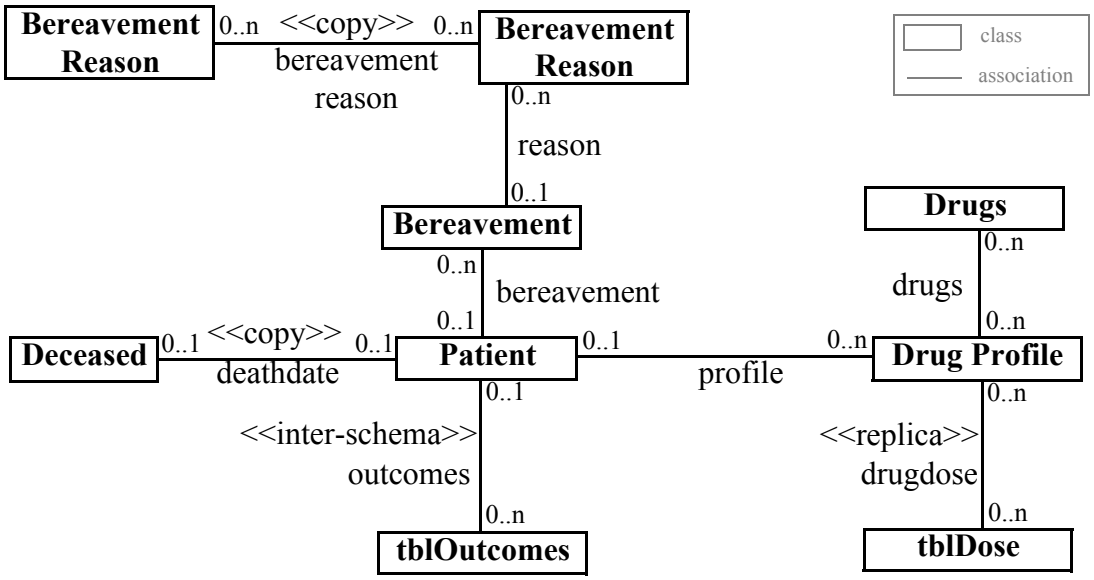


Figure 4.3: Sample of the palliative care conceptual schema

Figure 4.4 and Figure 4.5 depict clustering results. The clusters, which are or can be seen as subsystems, are represented with UML package diagrams. This corresponds to the UML notation of subsystems as suggested in [HC01]. Each cluster is visualised by one package; such a package can contain a class diagram or other packages. We do not visualise relationships between elements of different packages for clarity reasons. Such relationships are bundled to one relation between packages. Details of the relationships are shown as a list.

Figure 4.4 shows the clustering of the conceptual schema corresponding to the physical schemas. The conceptual schema is divided into the three database schemas bvmtdata, hopsdata and outcomes_be. Relationships between packages, i.e., the inter-schema relationships, are bundled and the details available as a list. In this sample this is the case for

the duplications bereavement reason and deathdate, the replication drugdose and the inter-schema association outcomes.

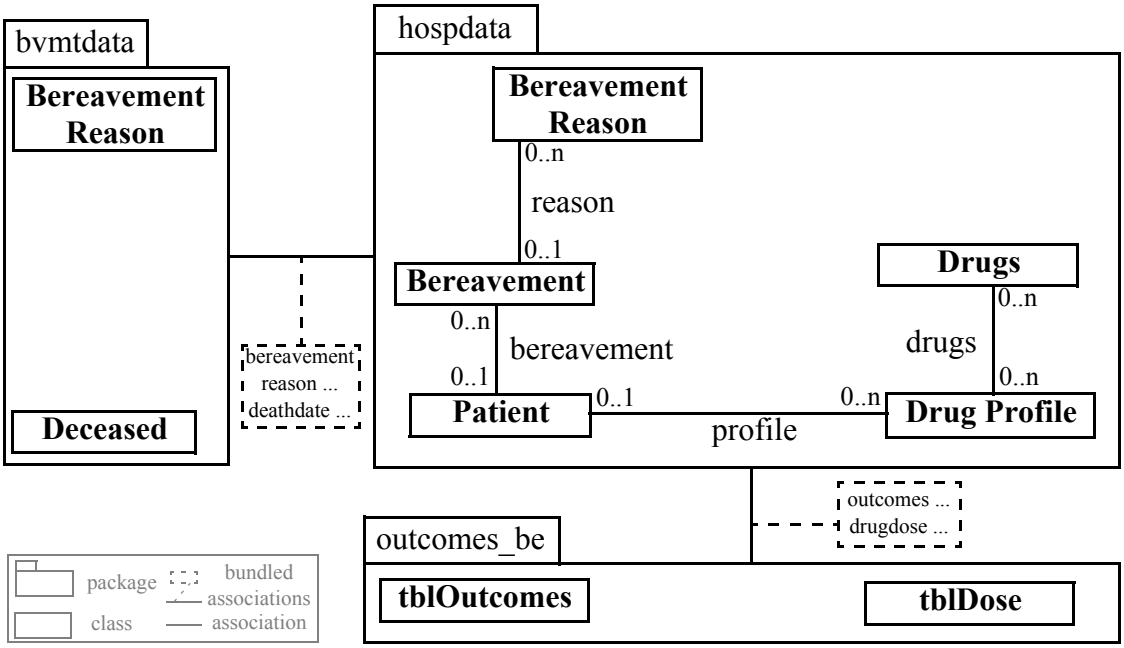


Figure 4.4: Clustered palliative care conceptual schema: sample 1

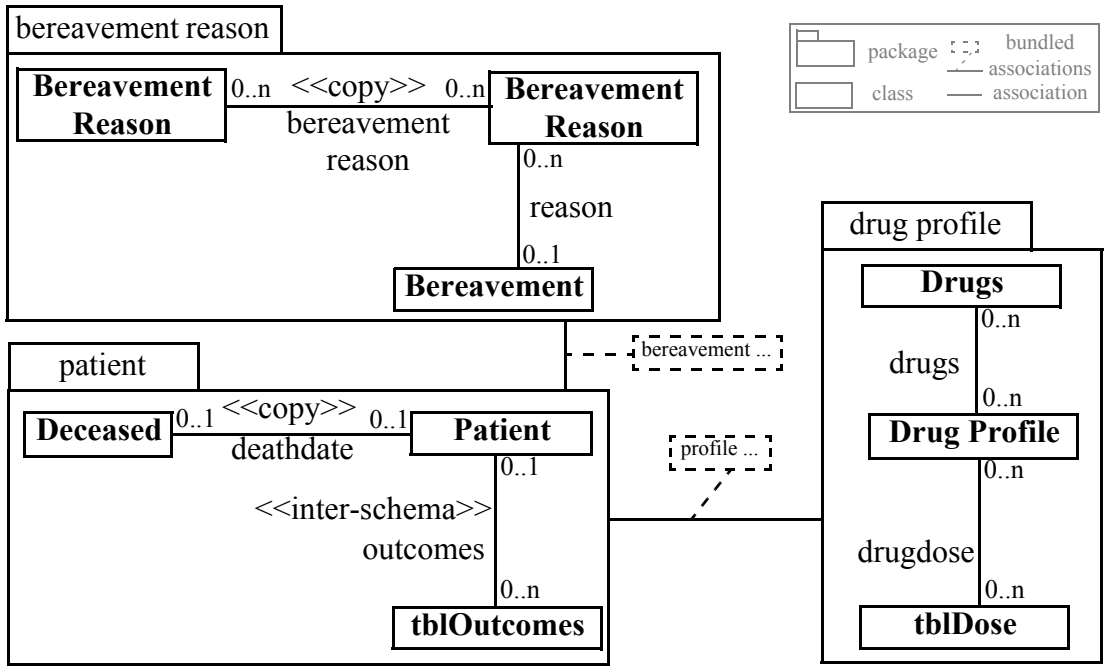


Figure 4.5: Clustered palliative care conceptual schema: sample 2

Figure 4.5 shows the clustering of classes related by redundancy dependencies. We attributed a weight of '10' for every redundancy dependency and a weight of '1' for all other relationships. In Figure 4.5 we can see that classes participating on a redundancy dependency, i.e., the redundancy dependency pairs (Bereavement Reason, Bereavement Reason), (Patient, Deceased) and (Drug Profile, tblDose), are located in the same packages.

Changing the clustering parameters does not affect the result except for one class: Bereavement is either located in the bereavement reason package or the patient package. Omitting the weights for the relationships has only a limited impact, although the redundancy dependency pairs are not clustered in the same package each time. The reason for this limited impact is the size of the chosen sample which does not allow much variations.

The result of the clustering is the partitioning of the web information system into data components.

4.2.2 Clustering Strategies

For data component extension, we emphasize the integration of existing data components with each other or the integration of existing and new data components. To determine the way of clustering. i.e., the *clustering strategy*, we need to determine the integration strategy. To clarify the integration scenario we assume a three tier architecture. The schema tier describes the information data structure (schemas) of our web information system. The application tier contains all the functionality to manipulate the information stored in the schemas. The graphical user interface (GUI) tier represents the interface to the real world, i.e., the users.

Schema versus Application Integration

Integration of web information systems can be done in various ways. To discuss the integration strategies we consider the two following extreme strategies: (1) schema integration which provides a consistent virtual schema located between the schema and application tier containing the integrated schemas; (2) application integration which results in an additional virtual application tier in-between the application and GUI tier.

In Figure 4.6 the schema integration is presented. The different application specific schemas are integrated into an overall virtual schema (grey). The schemas can be merged into a single (virtual) schema. A virtual schema can be a physical virtual schema which is served by a single homogenous distributed database management system. A virtual schema can also be realised by code with a schema access layer. System evolution in form of an additional application would result in (optionally) a new schema, an updated virtual schema, a new application and (optionally) a new GUI.

The application integration scenario in contrast would integrate an additional application without modifying the different schemas, cf. Figure 4.7. Instead, the virtual application tier (grey) is used to coordinate the applications as required. This additional application tier does further permit to use different heterogeneous and physically distributed database

management systems when support for distributed transaction processing standards [XA94] is present. For example this is the case when extending the system by a new application with its own new schema and GUI, cf. dashed parts in Figure 4.7. The required coordination however has to be realized by code in the virtual application tier. The identified application integration strategy is strongly related to enterprise application integration [Lin99] approaches, but we assume that the application composition does not involve event processing as realised either by polling on a shared database or using specific application APIs.

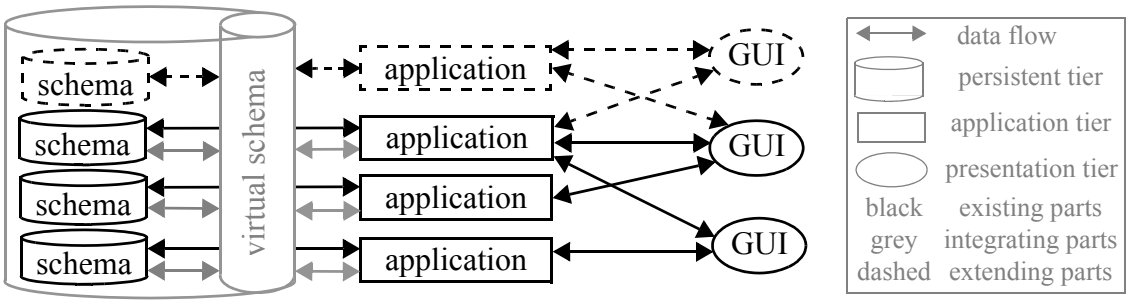


Figure 4.6: Schema Integration

The application integration approach requires the realisation of coordination activities. A suitable solution is to employ available middleware technology. For the schema integration approach the same technology can be applied when the virtual schema tier is realised by code, still, also database technology like views can be used. Depending on whether a code or strict database solution has been chosen, the flexibility of the virtual tier permits variations to a great extent. A more detailed comparison of both integration strategies can be found in [GW01].

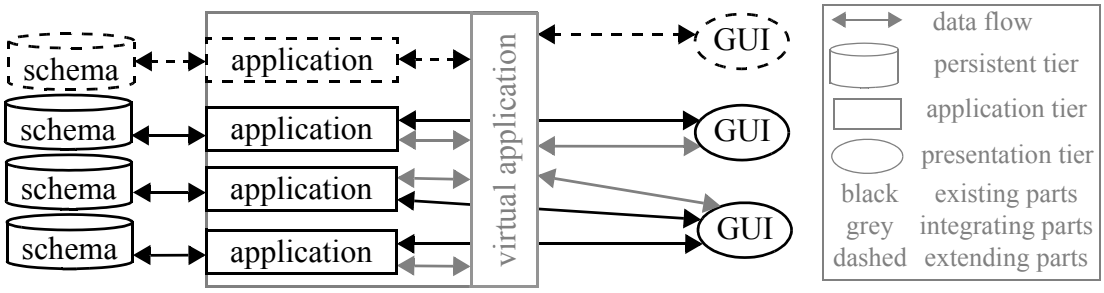


Figure 4.7: Application Integration

Clustering Strategy Decision

In addition to the integration strategy, the reengineer has to decided which aspect is more important for the integration and consequently for the clustering. We distinguish between two aspects. Firstly, the aspect of *data similarity* preponderates, i.e., the integration is con-

centrated around data that contains similar information. Secondly, the aspect of *functional similarity* preponderates, i.e., the integration is concentrated around application functionality that perform similar data manipulation tasks.

Based on these criteria, Table 4.1 shows which relationships have to be highly weighted for the clustering.

Regarding schema integration and the data similarity aspect, two relationship kinds give the best hints. Entities that have similar primary keys generally contains similar information. Thus clustering classes together, where the corresponding entities have similar primary keys, enables schema integration regarding the data similarity aspect. This also holds for classes related by redundancy dependencies because the redundancy dependencies express that similar data is stored in the participating classes (entities).

In case of schema integration and the functionality similarity aspect, several relationship kinds have to be considered. Since the preponderate aspect is functionality, relationships that relate classes by means of data manipulation purposes are relevant. In our case this includes replication, constraints, usage and inter-schema relationships. Replication is done for purpose of data access efficiency or data safety. The other three relationships form the basis for data manipulation, accessing data (usage), relating data (inter-schema) and controlling data (constraints).

	Schema integration	Application integration
data similarity	similar primary keys redundancy dependencies	usage relationships preceded by similar primary keys and/or redundancy dependencies
functional similarity	usage relationships inter-schema relationships replications constraints	usage relationships preceded by replications, constraints and inter-schema relationships usage relationships preceded by intra-schema relationships

Table 4.1: Relationship Weighting for Clustering

Clustering for application integration is harder. Regarding the data similarity aspect, the first step is the clustering of persistent classes that contain similar information. Then from these predefined clusters, the transient classes related via usage relationships are clustered. Thus, the transient classes manipulating similar information are clustered together.

Regarding application integration and the functional aspect, two scenarios exist. Firstly, like for schema integration, relationships that relate classes by means of data manipulation (except data access) purposes are relevant, i.e., replication, constraints and inter-schema relationships. After clustering the persistent classes in analogy with the schema integration case, the transient classes related via usage relationships are clustered considering the predefined clusters. Secondly, we start with the clustering according to the physical sche-

mas (intra-schema relationships). Based on these predefined clusters, the transient classes are clustered together that manipulate the same physical schema (via usage relationships).

Based on the clustering startegy, two ways of clustering are possible in our case: from coarse-grained clusters to fine-grained clusters and vice versa.

In the first case we start with a general clustering to find a coarse-grained partition of entities and data components. Next these coarse-grained clusters are then partitioned in smaller clusters by weighting the dependency types.

In the second case the fine-grained clustering by weighting the dependency types takes place first. Then the resulting clusters themselves are clustered by weighting the number of dependencies between the clusters. This weighting can be refined by a combined calculated value of the number of dependencies and the weight of dependency types. In both cases predefined clusters, e.g., conceptual schemas, have to be respected.

Which way of clustering is best for data component integration in web information systems depends on the reengineering settings. In case of multiple (small) systems that have to be integrated into a web information system a first coarse-grained clustering is certainly meaningful. This gives the reengineer an overview about how the existing and new data components are related. This holds especially when each involved conceptual schema stems from exactly one legacy system.

The opposite case is when few (bigger) systems, that should remain independent as far as possible, have to be integrated. The most data components will remain as they are and only for the new data components a fine-grained clustering is needed. Next, these new fine-grained clusters can be clustered together with the existing ones from the legacy systems. The final decision of how the clustering takes place is therefore the reengineers' task, depending on his experience and the settings. Note that in some cases both ways may be tried, followed by a partition comparison.

4.2.3 Data Component Classification

In Chapter 3 we introduced three data component kinds: corporate data components, data mediation components and ubiquitous data components. In this thesis every data component is assigned to exactly one of these three kinds.

Once the clusters are determined, the data components can be classified according to the three data component kinds. In case that a cluster contains more than one data component, all these data components normally are of the same kind. Therefore, when a data component kind is assigned to a cluster, this means that this kind is assigned to all data components in this cluster. In case that a data component in a cluster has a different kind than the cluster kind, we assigned this different kind to the data component in question. Such an exception is a cluster of omnipresent data components.

A data component (cluster) should be classified as “corporate” if it is part of mediation within the same organisation. From a user perspective such components (clusters) look like a single system. Corporate middleware from one vendor is generally used for integrating such data components. In general, a data component cluster that incorporates a data source will be classified as “corporate” or „ubiquitous“. Indeed, the storage of information mainly takes place within an organisation or a mobile device. Note that the data interfaces and portals are classified accordingly.

Omnipresent data components, i.e., a data component that is connected to many other data components, should be avoided. This situation corresponds more to a monolithic system than to a web information system. Still, within a cluster, i.e., corporate or ubiquitous data component, such central data components can exist. This can be seen as „local omnipresence“ which is resolved locally depending on the technology employed in the organisation or mobile device in question. The reengineer should distribute the data components to existing clusters or created explicitly a cluster with them. Nevertheless, such omnipresent data components can be interesting for reuse purposes.

Data component clusters that do not contain a data source and are not omnipresent, i.e., remain, should be classified as „mediation“. Indeed, these remaining clusters generally exchange or preprocess (format) data. They connect largely autonomous data components, i.e., they typically handle data that is replicated, duplicated and synchronised according to certain interoperability policies.

During the classification inconsistencies may emerge that require a cluster reevaluation or even a data component redesign. For this reason iteration support is needed. In a simplified way two iteration loops exist. Firstly, iteration that goes back to the clustering and redesign. Entities, relationships or data components may be changed (redesigned) or relocated in another cluster. Further, the clustering may be redone or reevaluated, e.g., by applying the orphan adoption technique. Secondly, a wider iteration can even go back to the reverse engineering. The integration steps or problems during integration may suggest further and deeper investigations of the underlying legacy systems.

4.3 Architectural Patterns for Data Mediation

As stated before we propose architectural patterns to (re)structure the web information system. These architectural patterns can be implemented by instantiating design patterns. Note that we do not restrict the design patterns those proposed by [GHJV95].

Traditional architectures for web information systems are based on the procedure-call paradigm, i.e., clients call service operations on server objects. This traditional development paradigm implies that the client programmer knows about the servers at the development time of the client software. This is disadvantageous in case of rapidly evolving architectures. Therefore, software engineers have recently started to migrate to a new paradigm called component-oriented software development.

The component-oriented software development paradigm promotes connection-based programming. This means that components are defined with well-defined interfaces in partial ignorance of each other. The actual connections among components are instantiated and deployed later. In other words, connections are now treated as “active” first-order citizens in distributed architectures [Szy99]. This supports networked evolution because it facilitates adding and removing connections with little changes to the components in a system. The patterns we propose are component-based and promote connection-based programming for web information systems.

In general, architectural patterns define the responsibility of typical parts of a system. Furthermore, they provide rules and guidelines for relationships between those components. Architectural patterns express and separate the concerns of fundamental structures in software systems [BMR+96].

We present four architectural patterns for data mediation:

- Data Portal
- Data Fusion
- Data Transducer
- Data Connection

Figure 4.8 depicts their relationships. Distributed information management (corporate or ubiquitous data) components are interfaced by means of the Data Portal pattern. The Data Portal is just one of three general Mediator Component patterns that can be connected through instances of the Data Connection pattern. The Data Fusion pattern is used to merge information from separate sources. Finally, the Data Transducer translates data into a different structure. This pattern is used for mediating among different data representations, as well as for rendering data for presentation to human clients (e.g., in web browsers). We describe these four architectural patterns in the next sections.

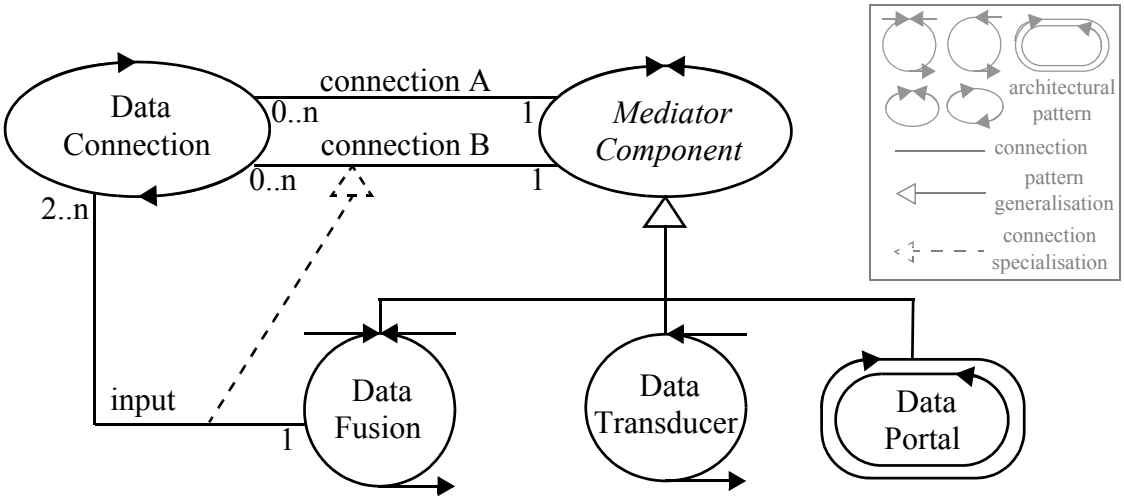


Figure 4.8: Architectural patterns: overview

4.3.1 Architectural Pattern: Data Portal

Name

Data Portal

Intent

A Data Portal is an interface of web information subsystems to the world. Its purpose is to make selected parts of the data (maintained within a subsystem) accessible for authorised external clients (services or users). We distinguish between two different versions of this pattern: *Export Data Portal*, which is responsible for making internal data available outside the subsystem; and *Import Data Portal,* which is used to import information from external sources into the subsystem data source(s). In practical applications, a clear distinction between these two patterns might not always be possible. The combination of exporting internal data and importing external data is an *Export & Import Data Portal*.

Motivation

Three main motivations exist for the Data Portal, namely resolving heterogeneity, providing data safety, and increasing data availability. Heterogeneity reflects on the fact that different web information system participants utilise various heterogeneous platforms and technologies for their data repositories. Interoperability between these different participants requires that this heterogeneity is resolved. The Data Portal serves this purpose by exploiting interoperability standards.

The requirement for data safety stems from the fact that, in many cases, external clients should not have access to all the data stored in internal databases (e.g., for protecting intellectual property, personal or sensitive information, etc). The Data Portal provides a level of isolation between the external schema (as it is perceived by the external client), and the source schemas of the internal databases in question.

Finally, the Data Portal facilitates fast access to a unified view of internal data structures. This is needed because data is often distributed among various different transactional and analytical repositories within organisations. The Data Portal serves as a façade for these

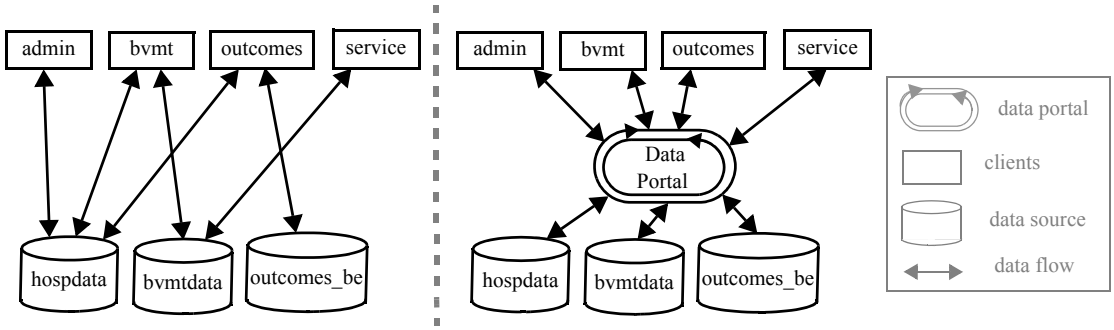


Figure 4.9: Data Portal Pattern use

various data sources, in which the external schema is a buffered view of the collection of schemas of all the involved data sources.

The Data Portal provides more than only a unified interface, it also handles safety and heterogeneity aspects. An example for the use of a data portal is given in Figure 4.9. The palliative care data sources are encapsulated for its accessing clients.

Applicability

The Data Portal pattern can be used whenever a subsystem needs to make available part of its data to participants outside its intranet. The Export Data Portal specialises in allowing external participants to query the internal data sources, while the Import Data Portal is used to update the internal data sources with data from the outside of the subsystem.

Structure

Figure 4.10 shows the structure of the Data Portal pattern.

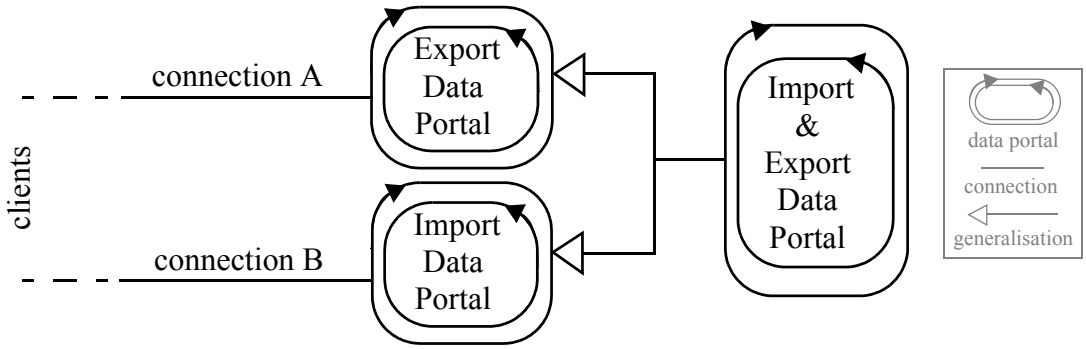


Figure 4.10: Structure of the Data Portal Pattern

Participants

Figure 4.11 shows the participants of the Data Portal pattern.

- source schemas
  are the schemas of the different data sources of the web information subsystem.
- external schema
  is the schema of the subsystem data as seen by the external clients.
- mapping function
  is a function that converts the source schemas (each conforming to the different data sources) to the external schema.

Collaborations

Each of the internal data sources is described with its own schema (a source schema). An external client, however, is not expected to be able to see any of these source schemas.

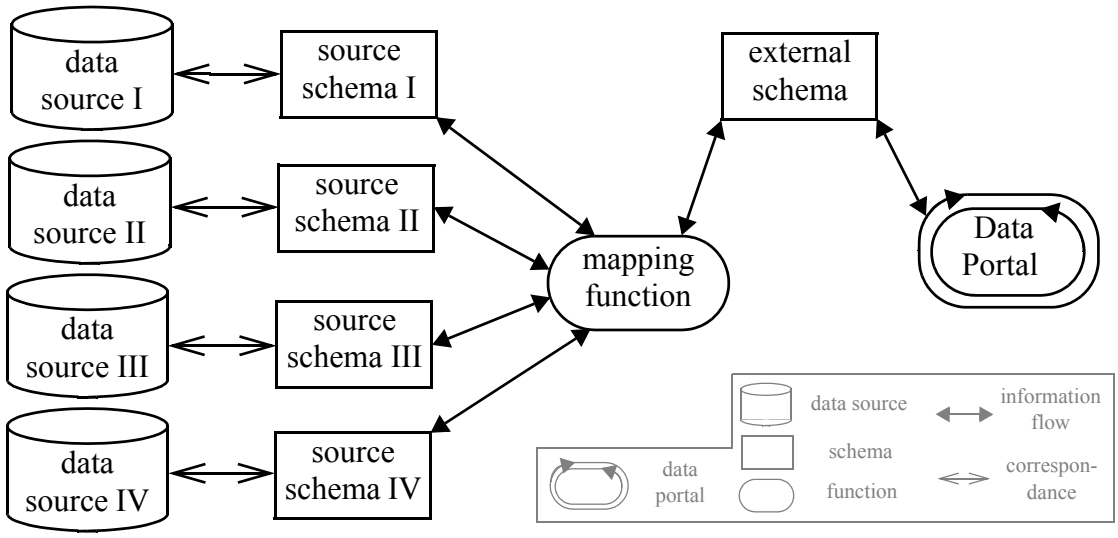


Figure 4.11: Participant View of the Data Portal Pattern

Instead, the client has access only to the "view" that the subsystem allows. This view is described using the external schema. The mapping function is responsible for resolving a request from the client (based on the external schema) into a request to the internal databases (described by the source schemas).

Consequences

There is a single entry point to external access and updates of the internal subsystem's databases. From the point of view of the (external) client, there is a single schema that corresponds to a single data source. The client does not have to worry about the different data sources, their types (whether they are ODBC, file-based, or other) nor how to query or update them individually. The subsystem, on the other hand, can filter and block access to sensitive information in the source databases. A disadvantage is that the complexity of building this access layer is significantly higher than using direct access, e.g., by using JDBC.

Example (Case Study)

Figure 4.12 shows a sample instantiation of the pattern. Each of the three databases of the palliative care hospice has one (logical) source schema. These schemas are mapped to a (conceptional) external export schema and a (conceptional) external import schema. These schemas are the basis for the Export Data Portal and Import Data Portal. An additional Import&Export Data Portal is built for clients that receive and pass data to the databases. The used Data Connection pattern is explained in Section 4.3.4.

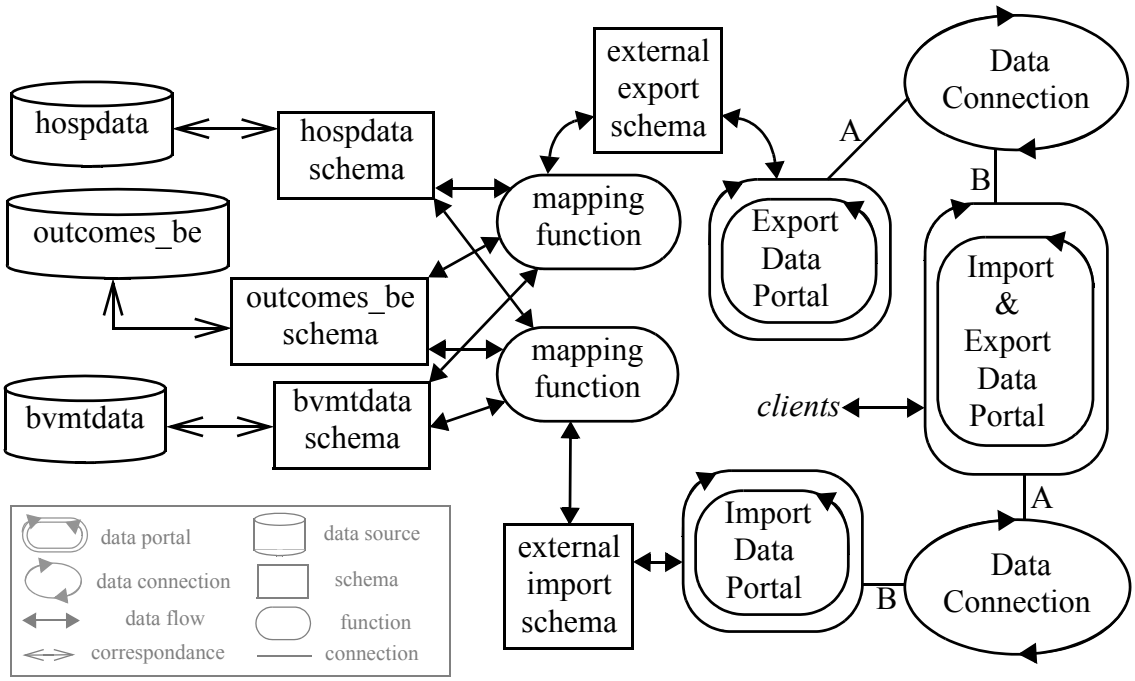


Figure 4.12: Data Portal Pattern sample

Related patterns

Facade: Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use [GHJV95].

Patterns similar to the Facade pattern found in [Ris00] are: *Facade* [Coc96], *Whole-Part* [BMR+96], *Wrapper Facade* [Sch99], *Abstract Database Interface* [ABM96], *Shared repository* and *(Legacy) Wrapper* [Mul95].

Data Abstraction and Object-Oriented Organization: In the architectural style based on data abstraction and object-oriented organisation, data representations and their associated primitive operations are encapsulated in an abstract data type or object [SG96].

4.3.2 Architectural Pattern: Data Fusion

Name

Data Fusion

Intent

To combine the data received from two or more data sources, into a single, unified data source.

Motivation

One of the goals of the semantic Web (as defined by the W3C) is the availability of a variety of data sources. These data sources will be mined by intelligent agents, which will understand their schemas, manipulate and combine their data, and then present it to the client as a unified data source. The client does not need to deal with these complexities, and instead, can assume a unique data source. Figure 4.13 shows a sample for the use of the Data Fusion Pattern. Data is collected from two data sources (:Source1 and :Source2). Then it is merged and sent to the :Client.

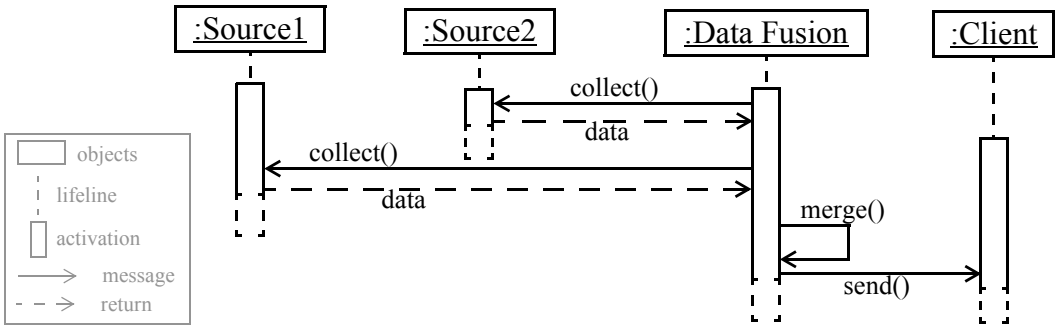


Figure 4.13: Data Fusion Pattern use

Applicability

A web information system often combines and aggregates data from several sources into a common data stream.

Structure

Figure 4.14 shows the structure of the Data Fusion pattern.

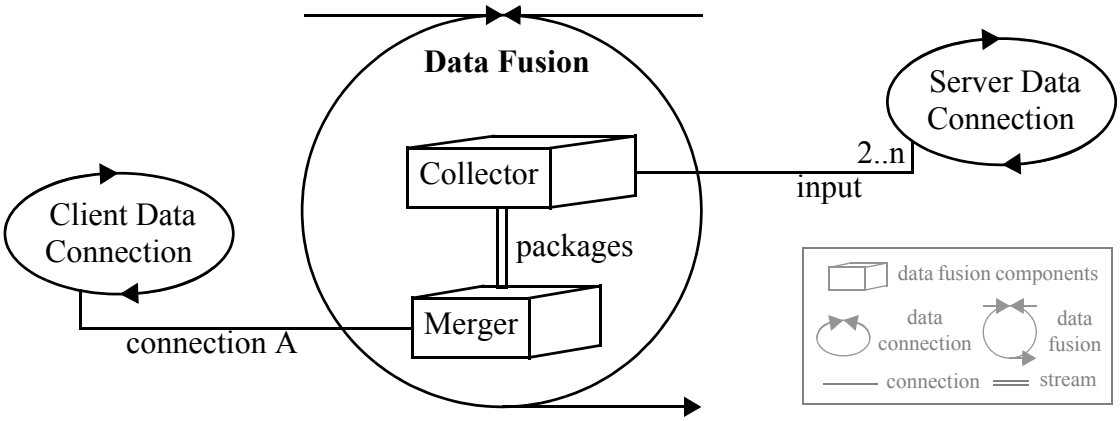


Figure 4.14: Structure of the Data Fusion Pattern

Participants

- Server Data Connections
  A Data Connection is created to connect the Data Fusion to each of the servers.
- Client Data Connection
  The Data Connection to the client.
- Collector
  Queries the different servers and receives their results.
- Merger
  The Merger is responsible for merging the results from the different data sources.

Collaborations

The instance of the Data Fusion pattern creates Server Data Connections to each of the servers. When the instance of the pattern receives a query, it translates it into a sequence of queries, each intended for a different server. The Collector is then responsible for executing these queries (using a Data Connection to the corresponding server). The Collector receives the results and passes them to the Merger, who proceeds to combine them. The resulting data is then sent to the client using the Client Data Connection.

Consequences

The availability of data, e.g. in XML, from different Mediator Components requires the existence of Data Fusion instances that merge these sources into a unified data source. A client application that uses a Data Fusion pattern does not need to worry about the complexities of accessing and merging multiple data sources.

Example (Case Study)

The medical diagnosis expert system needs data from the knowledge base as well as from the palliative care hospice, cf. Chapter 2. Based on the patients data, e.g., symptoms and /

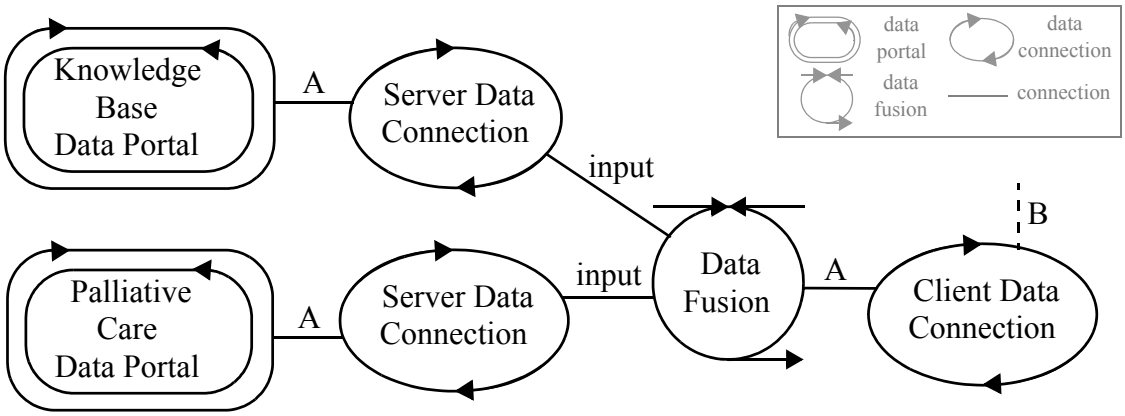


Figure 4.15: Data Fusion Pattern sample

or checkup results, the knowledge base is queried for diagnostic suggestions. Figure 4.15 shows that the Data Fusion receives data from both Export Data Portals via the Server Data Connections. The merged data, i.e., the diagnostic suggestions are then passed to the diagnosis GUI via the Client Data Connection.

Related patterns

Glue: Join a number of (multimedia) artefacts into a single composite artefact [CL00].

4.3.3 Architectural Pattern: Data Transducer

Name

Data Transducer

Intent

Convert data of a given source format into data of a different target format.

Motivation

There are cases where the client might not be able to interpret the data in its original format. In that case, the Data Transducer pattern converts the source data into a target format that the client expects. Figure 4.16 shows a sample for the use of the Data Transducer Pattern. Data sent from a data :Source is converted into the target format and sent to the :Client.

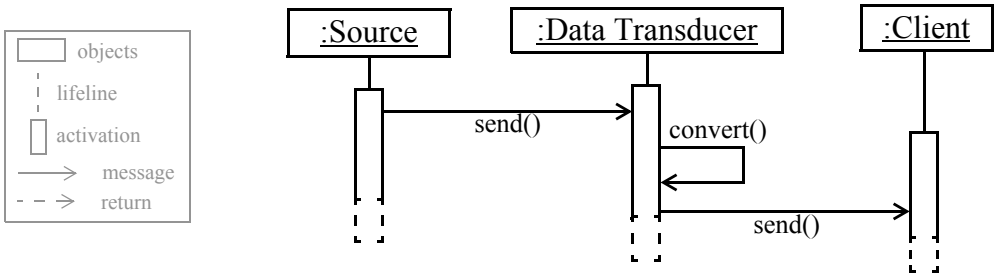


Figure 4.16: Data Transducer Pattern use

Applicability

Whenever an application requires data in a different format than the one that the data source provides, the Data Transducer pattern can be used. A client application might be designed with a given format in mind, but the server might provide data in a different format. Transducing the data from one format to another will allow both applications to interact without changes in either one of them. For example, the client expects data in an XML document with a different XMLSchema definition or DTD than the source produces.

Structure

Figure 4.17 depicts the structure of the Data Transducer pattern.

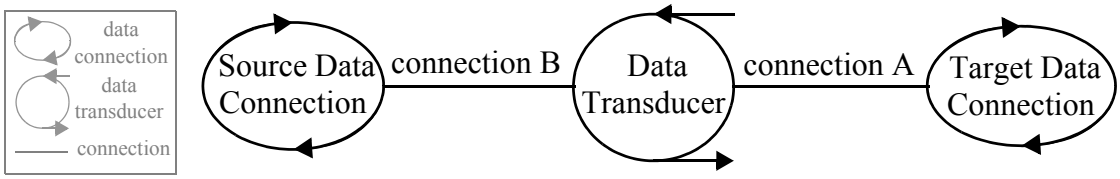


Figure 4.17: Structure of the Data Transducer Pattern

Participants

- Source Data Connection
  A Data Connection to the source of the data.
- Target Data Connection
  A Data Connection to the consumer of the data.
- Source Format
  The format of the original source data.
- Target Format
  The desired format for the resulting data.
- Transducing Function
  A function that converts the data from the source format into the target format.

Collaborations

The producer of data is connected to the Data Transducer using a Source Data Connection; similarly, the consumer of the data is connected to the Data Transducer using a Target Data Connection. The Source Data Connection pulls the data to be transduced. This data conforms to the source format and it is used as input to the transducing function. The output data, which conforms to the target format, is then fed to the Target Data Connection.

Consequences

This pattern allows the interaction of two applications that are designed with different schemas, i.e, that exchange data in different formats, to interoperate. The two applications do not need to be aware that the exchange format of the other is different. The disadvantage of this pattern is that the transduction could lose information because the target schema might not be able to convey all the data of the source schema.

Example (Case Study)

The example of Figure 4.18 shows a Data Transducer acting as renderer. The patient data from the Palliative Care Export Data Portal is rendered into a PDA compatible format.

Related patterns

Adapter: Converts the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [GHJV95].

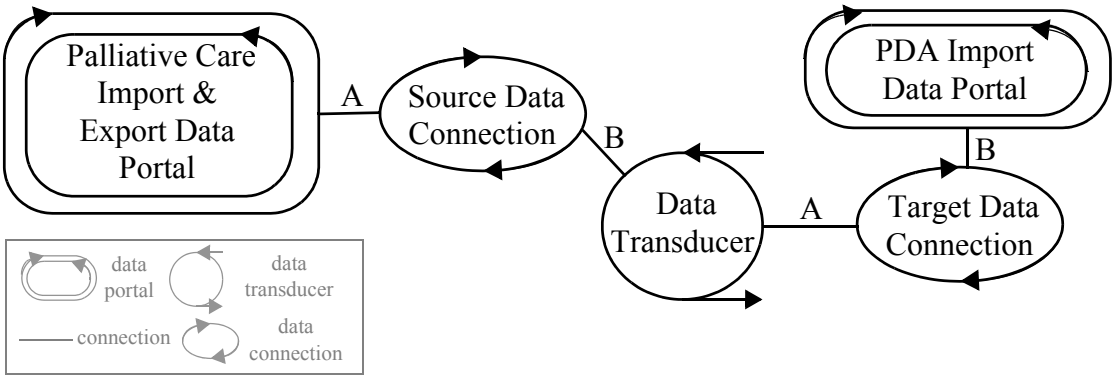


Figure 4.18: Data Transducer Pattern sample

4.3.4 Architectural Pattern: Data Connection

Name

Data Connection

Intent

Proactive service for pulling data of interest from a given data source and pushing this data to a given data sink.

Motivation

Traditionally, data exchange is controlled by either the client or the server of a web information system. This concept limits scalability and flexibility for rapidly evolving distributed architectures of web information systems. Therefore, engineers have begun to treat (data) connections as first-order citizens in their architectural designs. This trend is merely reflecting on a general trend in current software engineering practice from the traditional call-procedure paradigm to the new paradigm of connection-based programming [Szy99].

For the application domain of engineering web information systems, a Data Connection is an instance that controls the exchange of electronic data among several components. A Data Connection is not a placeholder but a system component. Figure 4.19 shows two samples for the use of the Data Connection Pattern. On the left side we have a static link to the :Source and a dynamic link to the :Client. The *Connection Policy* is an *Updater*, i.e., the :Source updates the :Client. The :DataConnection stores the data until the :Client connects to it and an update can be performed. On the right side both links are static, but we

have a *Synchroniser* Connection Policy. The :DataConnection collects the data from :Source and :Client, syncrhronises it and updates the :Source and the :Client.

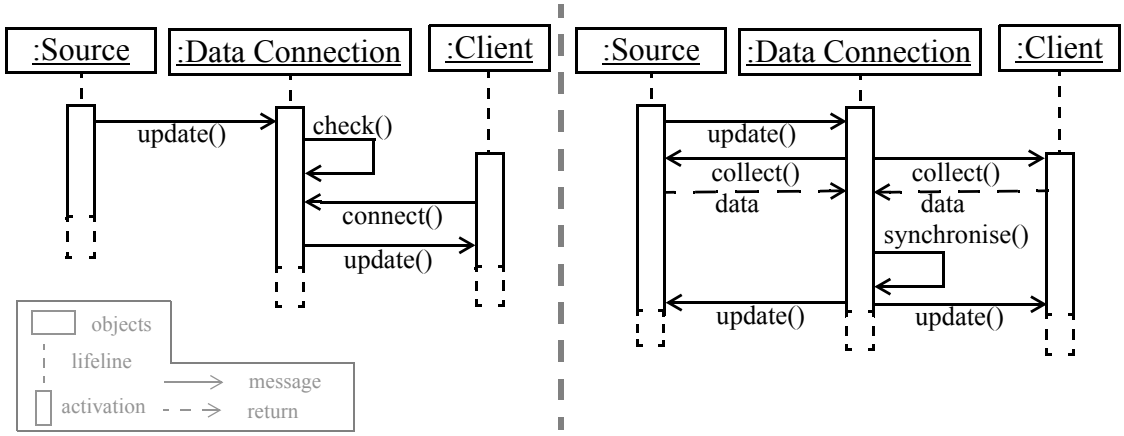


Figure 4.19: Data Connection Pattern use

Applicability

Whenever we need to constantly retrieve data from or send data to a mediator component.

Structure

Figure 4.20 shows the structure of the pattern.

Participants

- server connection and client connection
  These connections are created as instance of the Data Connection pattern to the server and the client, and can be either static or dynamic links. A Static Link assumes a reliable connection between the client and the server, while a Dynamic Link can handle a change in the Internet Protocol (IP) address of the client and temporal disconnection between client and server (both situations are common in mobile devices).
- Connection Policy
  Indicates the properties of the data connection. An Updater (data connection) is only concerned with handling continuous updates to the client (stock market prices, news updates), while the Synchroniser assumes that the client has a copy of the data and needs to synchronise it with the master copy in the server.
- Server
  The Mediator Component that exports data, i.e., to which the client wants to connect.
- Client
  The Mediator Component that imports data, i.e., that needs to connect to a server.

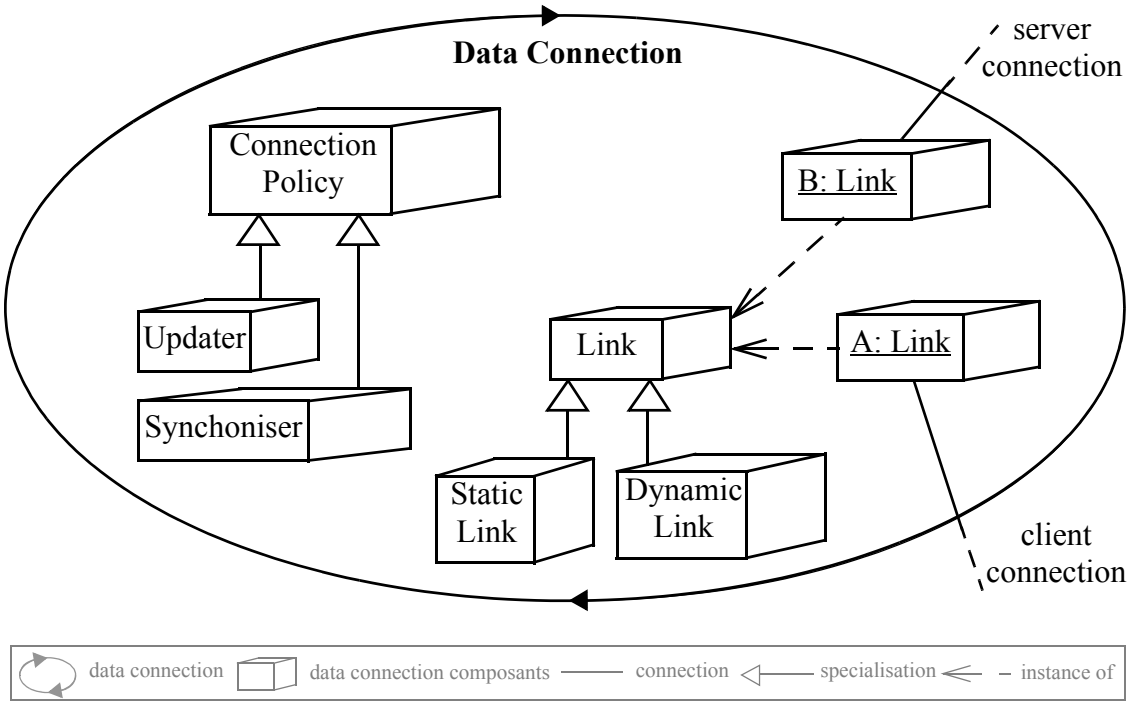


Figure 4.20: Structure of the Data Connection Pattern

Collaborations

When instantiated, the Data Connection pattern creates communication links to both, the client and the server. The instance of the Data Connection pattern, according to its connection policy, receives requests from the client and converts them into requests to the server. The reply from the server is then translated and sent to the client. The Data Connection is responsible for the issues related to the connection to the component, such as authentication, encryption, roaming, temporal disconnection, etc.

Consequences

The main benefit of connection-based programming is late binding of Mediator Components. This means that it becomes possible to develop Mediator Components with well-defined interfaces more-or-less in isolation from each other, and flexibly connect them at a later point in time. The Data Connection pattern is also responsible for handling the complexities of the communication with the Mediator Component, allowing the client to be unaware of them.

Example (Case Study)

Figure 4.12 depicts two static Data Connections with an Updater policy. A dynamic Data Connection with Synchroniser policy is the Target Data Connection of Figure 4.18.

Related patterns

Broker: Produces distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forward requests, as well as for transmitting results and exceptions [BMR+96], see also *Broker* [Mul95].

Connector: Decouple service initialisation from the services provided [Sch97], see also *Client-Dispatcher-Server* (p. 34) and *Forwarder/Receiver* [BMR+96].

Data Filter Architecture Pattern: Filters the contents of client requests in a distributed system, according to predefined policies. Filtering can occur locally or remotely [FF99].

Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [GHJV95].

Pipes and Filters: Provides filter components which encapsulate processing steps for data streams. The pipes pass through the data between adjacent filters [Meu95, BMR+96, SG96].

Proxy: Provide a surrogate or placeholder for another object to control access to it [GHJV95].

4.3.5 Architectural Pattern Application Examples

Figure 4.21 shows three prototypes in the Health Care web information system where architectural patterns are applied. The overall architecture integrates the three organisational sites, namely the Palliative Care Center Victoria, the Medical Diagnosis ES (expert system) and Information Management System and the ubiquitous site Physician PDA. Further a Physician Browser and a Patient Browser are integrated. Different Data Portals per site hide the complexity and heterogeneity.

The Physician PDA has the purpose of viewing and collecting data from clinical practice. This mediation is realised by the prototype 1 by means of two Data Connections and a Data Transducer pattern for rendering clinical data on the PDA. The link from the Target Data Connection to the PDA Import & Export Data Portal is a dynamic link and the connection policy is a Synchronizer since the PDA can be used off-line (see Section 4.3.4 and Figure 4.18).

Furthermore, Figure 4.21 shows that the Data Fusion pattern is used for combining the data sets from the different organisational sites. In cases where data sets are not structurally compliant to the chosen web information systems format, a Data Transducer pattern is instantiated to provide the structural translation, cf. data stemming from the Information Management System. All instances of Data Connection deployed between the organisational sites enact an Updater policy, because information is communicated only in one direction. Data Transducers render the data for the browsers.

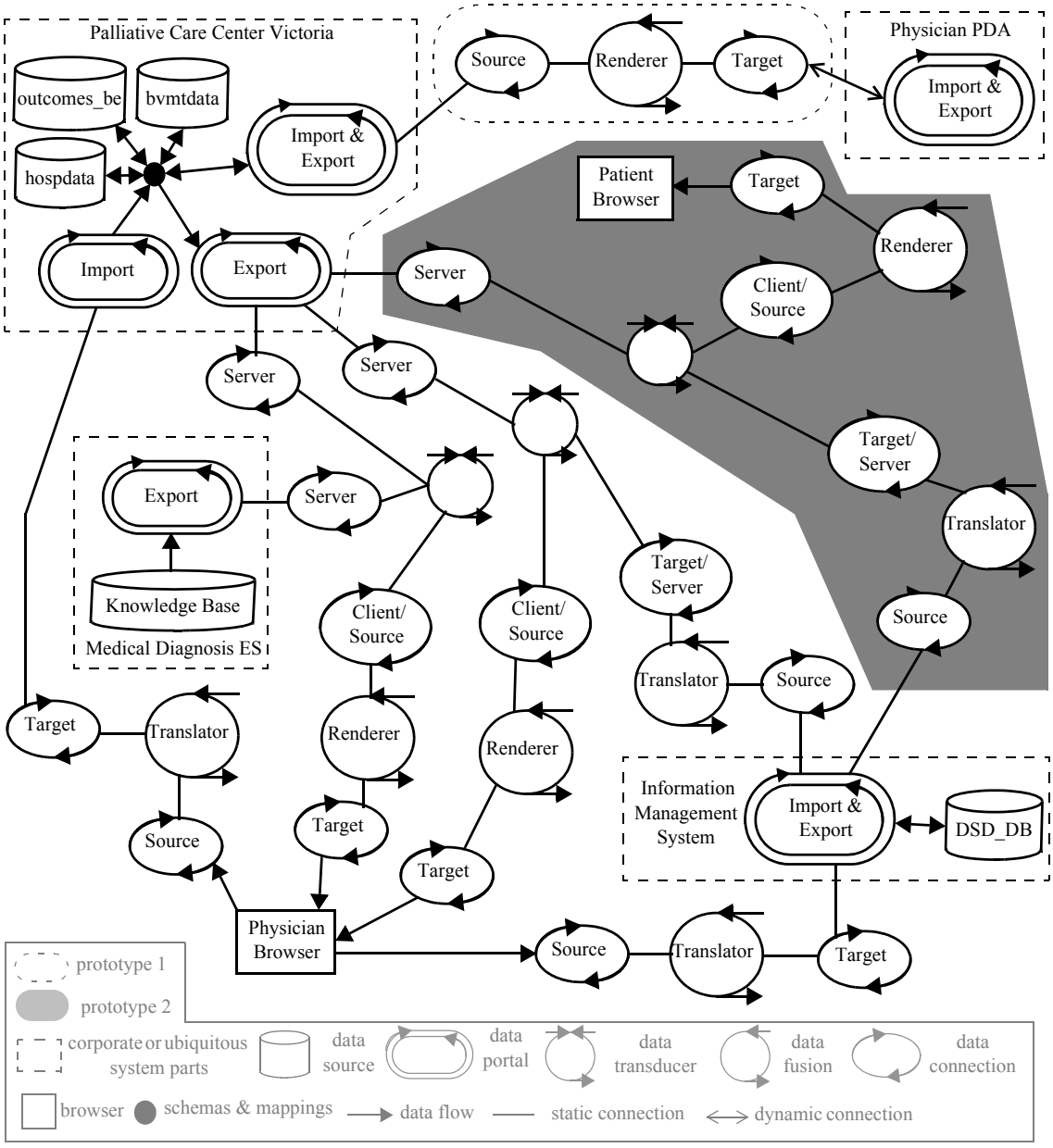


Figure 4.21: Architectural pattern application examples

Prototype 2 enables the patients to access their data via a browser (Patient Browser). The prototype 3 which consists of all architectural patterns that are not of the two other prototypes, permits data access for the physicians. Note that the input from the Physician Browser is transduced before it is passed to the organisational sites.

4.4 Access Layer Generation and Model Execution

During the extension, i.e., clustering, classifying and restructuring, the retrieved correspondences to the databases are maintained through the schema mappings. For new schemas these correspondences are established. Thus, every element of a conceptual schema has a correspondent in the logical schema and consequently in the database. This enables us to generate a transactional access layer to support open nested transactions on the extended web information system.

Further, we validate the extensions by implementing prototypes. For prototyping, we model only system (data component) parts. The focus is to provide code for the manipulation and exchange of data. The mediation layer provides transition between persistent and transient classes, i.e., usage relationships, and the manipulation of data inside the transient classes. The publishing layer reads and writes XML documents that contain the data to exchange.

Figure 4.22 depicts the layers along with their connections, the data and the users. The data access is done via the transactional access layer where the data portal pattern is implemented. Implemented data fusion and/or data transducer patterns occur in the mediation layer. The mediation layer has connections to publishing layers, which are interfaces between two mediation layers or between a mediation layer and a GUI.

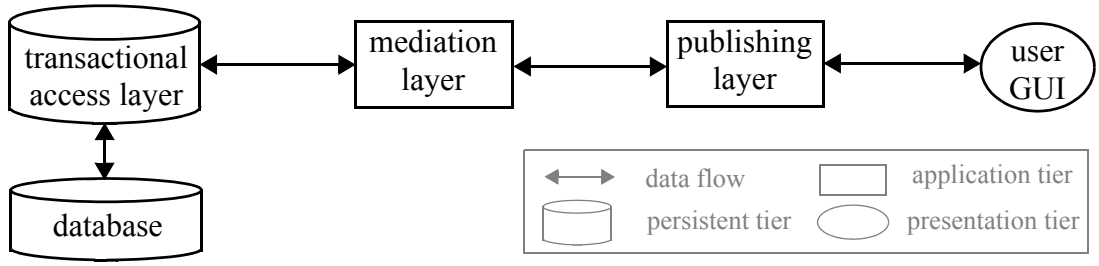


Figure 4.22: Modelled and generated layers

4.4.1 Transactional Access Layer

The complexity of building an access layer is significantly higher than using direct access, e.g., by using JDBC. Nevertheless the retrieved conceptual schemas (cf. Chapter 3) represent an object-oriented access layer to the data of the web information system. The reengineer does not have to know the logical or even physical schemas nor which specific databases are involved. An access layer can be generated from the conceptual schemas and enables database independent access to the data.

Figure 4.23 shows an overview of the access layer generator. After a conceptual schema is retrieved from the data sources, the generator uses the retrieved information to generate an access layer. The logical schemas, schema mappings and conceptual schema are

used to assign each generated conceptual schema element to the corresponding logical schema element. The configuration enables the explicit allocation of the logical schema elements to the databases internal physical schema elements via JDBC. The generated access layer can then access the data.

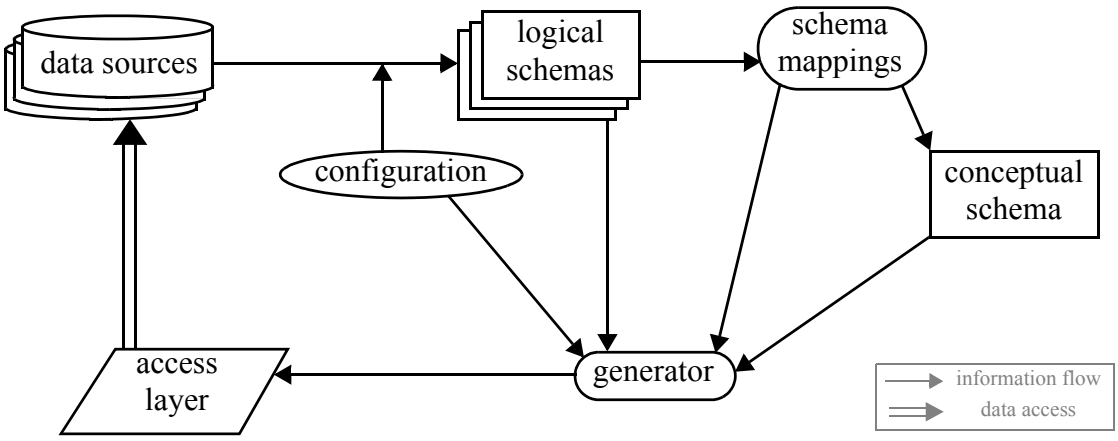


Figure 4.23: Access layer generator overview

ACID transaction

Further, the access layer has to support transactions in order to allow distributed and concurrent access. Grand [Gra99] presented four transaction related patterns. Figure 4.24 shows the four patterns which are a solid basis for transaction management. The ACID transaction pattern enables the compliance of the ACID (atomicity, consistency, isolation, durability) properties of a transaction. The nesting of transactions is done within the composite transaction pattern. The two phase commit pattern enables the atomicity for the composite transaction pattern. History management for ACID transactions is done by the audit trail pattern.

We only explain the ACID transaction pattern that directly accesses the data and omit a detailed description of the three other patterns. The ACID transaction pattern has to be realised in accordance with the generated access layer, i.e. the schemas and mapping. Such,

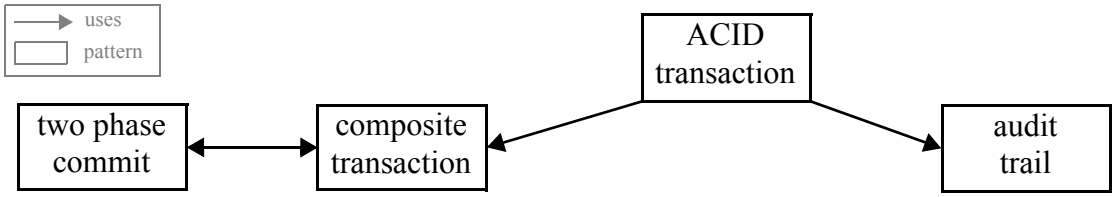


Figure 4.24: ACID transaction related patterns [Gra99]

the generation of the transactional access layer consist of two intertwined tasks: generating the (data) access layer and implementing the ACID transaction pattern.

We introduce an intermediate schema that is an object-oriented counterpart of the logical schema. A logical schema represents the physical schema and thus the data structure of the database. The intermediate schema, i.e., the object-oriented part of this logical schema, is responsible for the object-oriented encapsulation of the data. We have a one-to-one correspondence between the relational and object-oriented part of the logical schema. Figure 4.25 depicts these one-to-one mappings which precede the schema mappings. Note that the schema mappings as presented in Section 3.2.3 are not affected because the intermediate schemas are "copies" of the logical schemas.

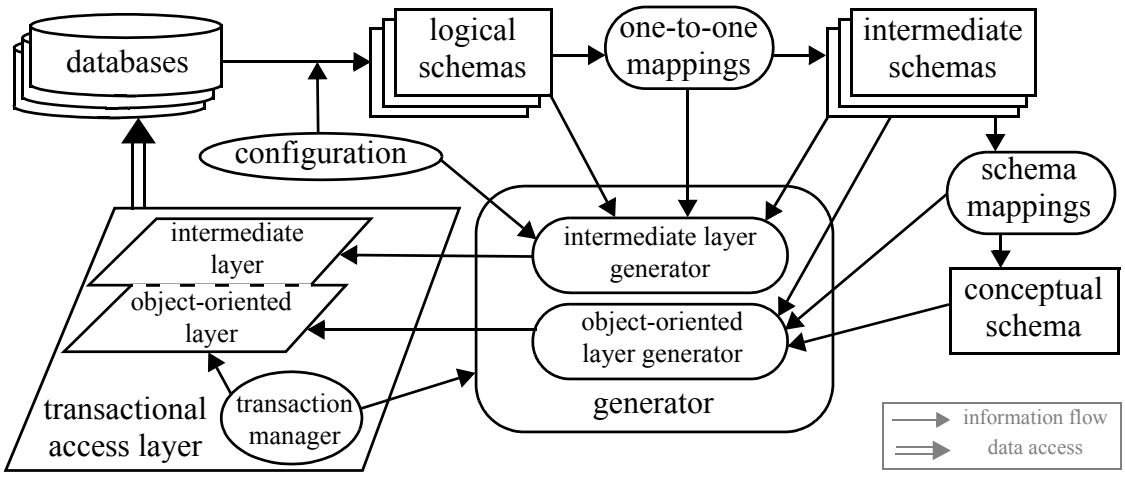


Figure 4.25: Transactional access layer generator overview

The intermediate schemas are used to guarantee ACID properties. The generator is divided in two generators: the intermediate layer generator and the object-oriented layer generator, cf. Figure 4.25. The transaction manager comprises the ACID transaction implementation, is considered for generation and used by the transactional access layer during data access. Additionally to the object-oriented layer, the intermediate layer is generated.

Since the transactional access layer accesses distributed heterogeneous data sources, not only databases, it has to guarantee the ACID properties by itself. To guarantee atomicity and isolation the additional intermediate layer is needed to cache values during transaction execution on the object-oriented layer. Atomicity is realised by caching the old values in the intermediate layer during transaction execution. Only after a successful storage in the database the new values are stored in the intermediate layer. In case of a transaction failure the old values from the intermediate layer overwrite the new values of the object-oriented layer. The same cache mechanism is used for isolation. When during a transac-

tion execution another transaction wants to access a same data value, the old consistent value from the intermediate layer will be provided. This is realised with a read-/write-lock mechanism.

Consistency and durability are harder to achieve. Consistency depends on a correct transactional access layer. This includes a correct transaction manager. Correctness is ensured through the generation of the transactional access layer. Repeated manual implementation is an error prone task compared to code generation. Durability depends on the underlying data sources. A database management system provides durability whereas it is hard to guarantee for a file system. For further details of the transactional access layer generation we refer to [Wol01].

4.4.2 Mediation Layer

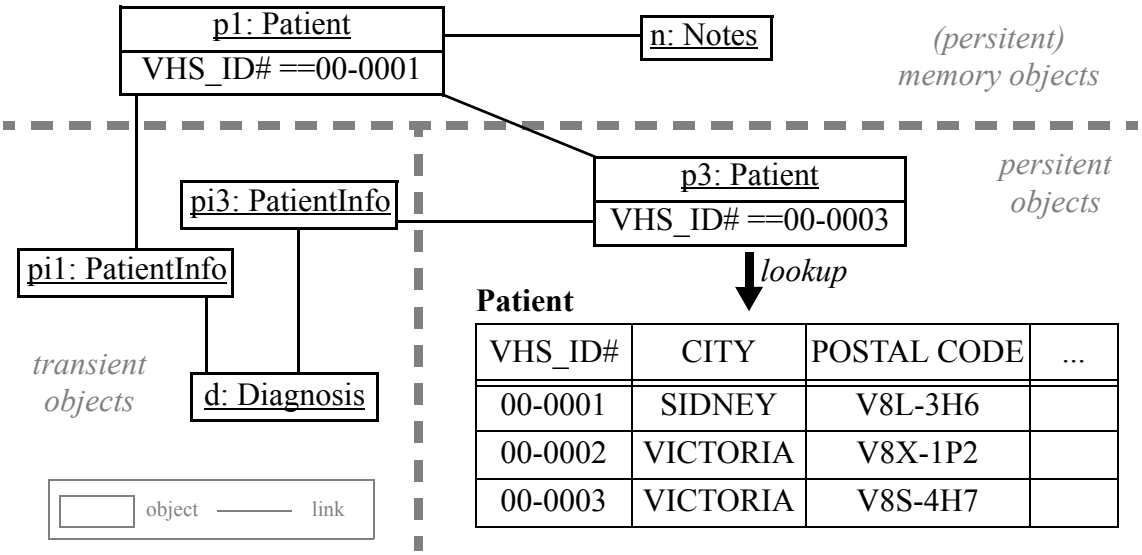
The transactional access layer provides access to the persistent data. We divide the mediation layer in data access interfaces and data manipulation services. Both layer parts are modelled with UML class diagrams and story diagrams. Code is than generated from these models. A data access interface establishes the transition between the persistent and transient parts of the web information system. Data access interfaces are normally built on top of a transactional access layer. Data processing is done within data manipulation services. Data manipulation services are generally situated between a data access interface and a publishing layer or between two publishing layers.

Data access interfaces

Data access interfaces implement external schemas. Modelling such interfaces means modelling transitions between persistent and transient objects. Links between transient objects only exist if the participating objects exist, e.g., d: Diagnosis and pi1: PatientInfo in Figure 4.26. For links between persistent objects we have a similar situation, a link exists if the participating persistent objects exist in the data source. In contrast to the transient case, a lookup into the data source may be needed, e.g., p1: Patient and p3: Patient in Figure 4.26. This is handled by the transaction manager. If the objects already exist as (persistent) memory objects the lookup is not needed, e.g., p1: Patient and n: Notes in Figure 4.26.

In all cases we have the same clear modelling concept: a link is established if the source and target exist.

For links on the transition, i.e., links between persistent and transient objects, we have two situations. Firstly, the persistent object exists already as memory object and thus can be accessed from a transient object like another transient object, e.g., pi1: PatientInfo and p1: Patient in Figure 4.26. Secondly, if the persistent object does not exist as memory object, a data source lookup has to take place to create the correspondent memory object before it can be accessed, pi3: PatientInfo and p3: Patient in Figure 4.26.



VHS_ID#	CITY	POSTAL CODE	...
00-0001	SIDNEY	V8L-3H6	
00-0002	VICTORIA	V8X-1P2	
00-0003	VICTORIA	V8S-4H7	

Figure 4.26: Examples of links between different object kinds

The present clear modelling concept is obfuscated if no distinction between these two situations can be made. Both situations take place in the transient world. In the first case the source and target of the links exist. In the second case only the source exists and the target has to be created.

The first situation is handled with the existing constructs, i.e., a link between a transient object and a persistent object only considers existing memory objects. To cover the second situation we introduced a stereotype <<search>>. When a transient object has to access a persistent object whether the persistent object already exist as memory object or not, in the story pattern the accessed persistent object and the corresponding link are marked with the stereotype <<search>>, cf. Figure 4.27. The semantics of <<search>> is as follows:

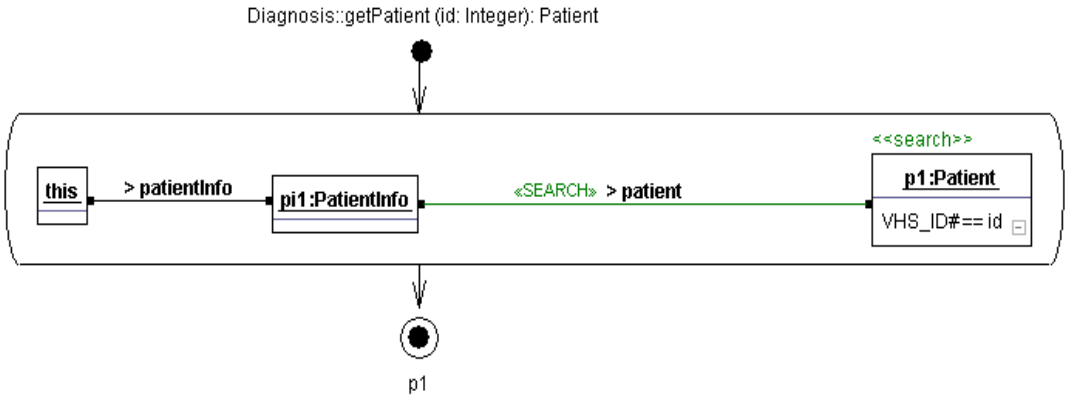


Figure 4.27: Examples of a <<search>> link

try to find the persistent object as memory object; if it does not exist as memory object try to create this object with a data source lookup; only if in both cases the persistent object cannot be matched, the link cannot be established. Figure 4.27 shows a <<search>> link for the transition between pi1: PatientInfo and p1: Patient in a story diagram.

Data manipulation services

The data can be accessed. Next, it is processed by manipulating objects inside the data manipulation services. We call data objects the objects that hold data. The data is exchanged via XML documents. These XML documents are internally represented by the data manipulation services as data objects. Data is processed either with data objects that directly access the data through the data access interfaces or with data objects that are received through the publishing layer. The object manipulation itself is done with story diagrams.

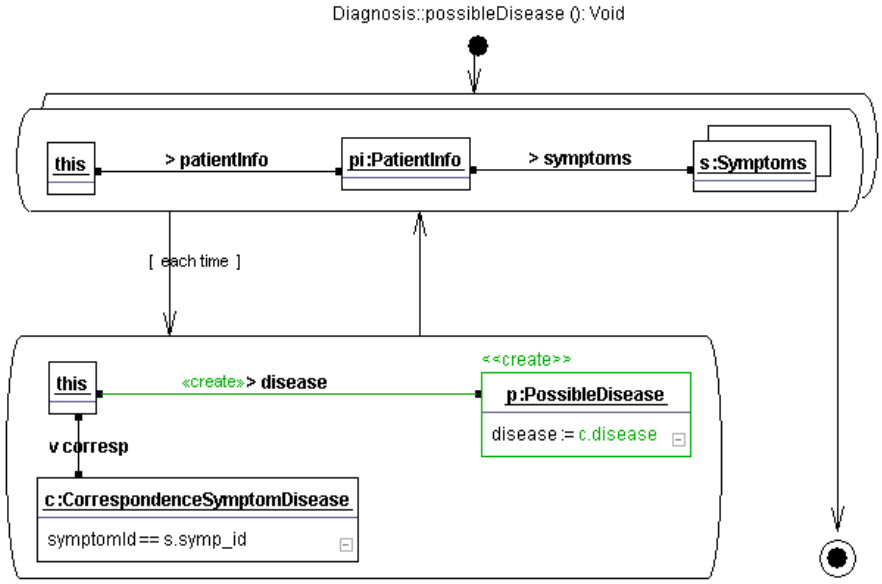


Figure 4.28: Sample Data Fusion pattern modelled with a story diagram

Figure 4.28 depicts a story diagram which models a Data Fusion pattern. Recorded patient Symptoms are collected from the hospdata database followed by a check for a possible disease in the knowledge base. Possible diseases are linked to the current Diagnosis object. The possible diseases can then be published to the physician for diagnostic support.

4.4.3 Publishing Layer

The publishing layer writes data object structures into XML documents and vice versa, i.e., reads XML documents and creates the corresponding data objects. In order to perform these read and write operations the publishing layer needs a referencing data object model for the exchanged data. Data object models are represented as class diagrams. Conform-

ing to these data object models the publishing layer reads XML documents into data objects or writes data objects into XML documents.

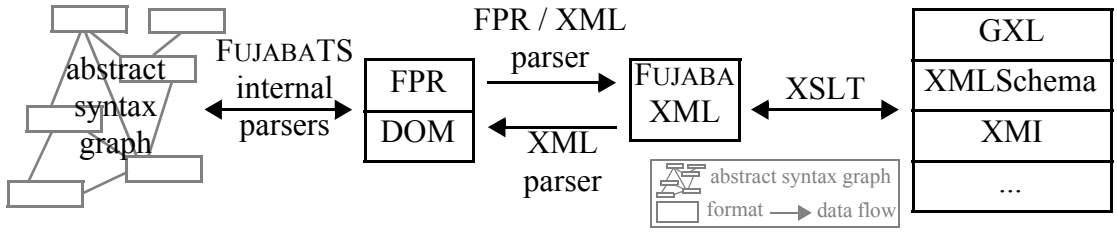


Figure 4.29: Abstract syntax graph publishing

The read and write operations from data object models to XML documents are depicted in Figure 4.29. A data object model is internally represented as abstract syntax graph. An abstract syntax graph is parsed into the FUJABATS proprietary FPR (Fujaba PRojekt file) format, which is itself parsed into a proprietary FujabaXML format. Finally this XML document can be transformed with XSLT (Extensible Stylesheet Language Transformation) to other formats like GXL (Graph eXchange Lnaguage), XMLSchema, (XML Metadata Interchange) XMI, etc. The way back is, in analogy, from XML documents to FujabaXML (via XSLT), followed by an XML parser into a DOM (Document Object Model) structure and finally into an abstract syntax graph. Details of abstract syntax graph publishing can be found in [HL02].

We give two samples for publishing layers. Firstly, we present a HTML-based data portal. which is a special case of publishing layer. The HTML-based data portal reads HTML template pages and returns them as filled HTML pages. Figure 4.30 shows a scenario for

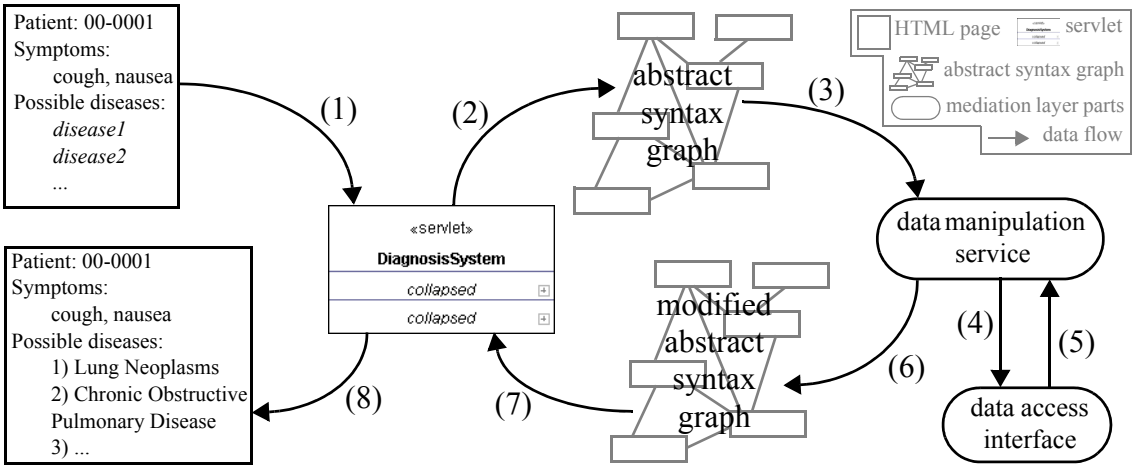


Figure 4.30: Sample publishing layer: web portal

filling a HTML template page. The template page, i.e., the page with placeholders disease1, disease2, etc., is read by the DiagnosisSystem servlet (1). The abstract syntax graph corresponding to the template page is built (2) in accordance to HTML 4.01[1]. This abstract syntax graph is then passed to a data manipulation service (3). The data manipulation service accesses the data source via a data access interface (4) and (5) and modifies the abstract syntax graph. The modified abstract syntax graph is then parsed (7) into the result HTML page and posted to the client (8). Details of the modelling and code generation of servlets are described in [Kam03].

Secondly, a publishing layer that reads data and a publishing layer that writes data for a data transducer. Figure 4.31 shows the correspondence between the input Patient description and the output Patient description. The input document only shows the Surname object description in GXL (upper part of Figure 4.31). This input document is read into an

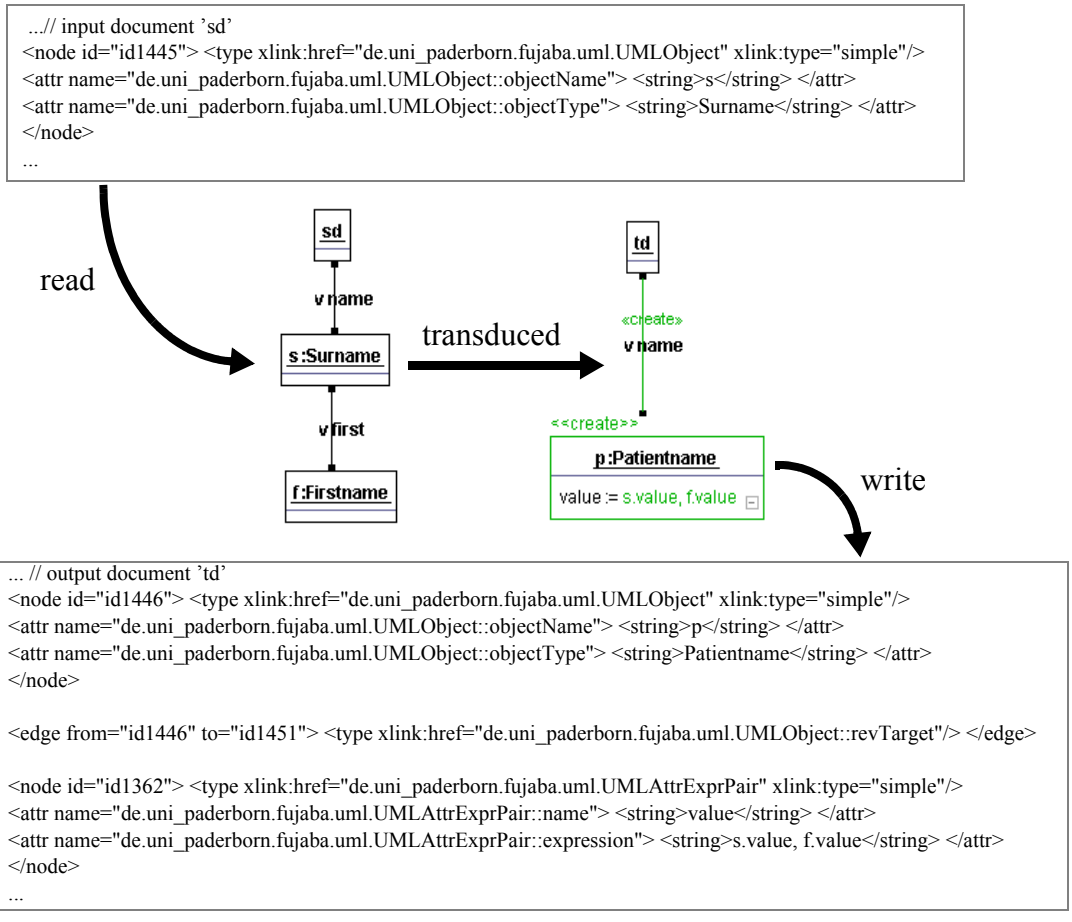


Figure 4.31: Sample publishing layers for a data transducer

1. http://www.w3.org/TR/html401

object structure with nodes for the document sd, the surname s: Surname and the firstname f: Firstname. These objects are transduced into the target document (td) object structure. Figure 4.31 shows the story pattern, which creates p: Patientname that unifies the s: Surname and f: Firstname values. Finally the output document is written. In Figure 4.31 only the Patientname object, including its value and the edge between them, is depicted. The read and write operations are performed through the abstract syntax graph publishing, cf. Figure 4.29.

4.4.4 Multimedia extension

Medical data nowadays consists among other data of multimedia data, e.g., videos or commented radiographs. Therefore the existing legacy systems have to be extended to handle multimedia data. Multimedia data owns peculiarities. We consider the following meaningful peculiarities:

- huge size (gigabytes are the lower bound),
- enclosed meta data (e.g., title and duration of a video, ...),
- particular formats (e.g., video, associated images and audio, ...),
- specific operations (e.g., displaying, real time streaming, ...),
- time dependence (e.g., life streaming audio and video synchronisation, ...) and
- combination of discrete and continuous data (e.g., associated images and audio, ...).

To sustain these peculiarities, extensions in the data tier are needed. The extension of a data source is hard to achieve. Therefore the transactional access layer has to handle some of the extensions. By dividing the data for storage and re-combining it for retrieval the size, format and meta data problems can be handled. Unsupported formats and meta data are resolved by storing extra data, e.g., format characteristics are stored additionally to the essential data.

Dividing the data is needed because of capacity restrictions for data storage and efficiency of data retrieval. This leads to data distribution and thus to the need of ACID transaction management, which is provided by the transactional access layer. The read-/write-lock mechanism has to be refined. Further the strategy pattern [GHJV95] is used for providing different partitioning/combining algorithms. For further details we refer to [Wag01].

The extensions for multimedia data in the mediation layer are numerous. Modelling for indexing, sorting and searching multimedia data are required. Further multimedia data has to be composed and synchronised before it can be published. Many approaches exist to cope with these issues, e.g., [vC97, Man99, CL00, ES02].

The publishing layer would be mainly responsible for displaying the multimedia contents to the clients. Various solutions exists how to achieve publishing, real-time displaying, etc. These problems are out of the scope of this dissertation, such we refer to the literature for details.

4.5 Tool Support

The tool support for the data component integration process is located in the REDDMOM project. REDDMOM (and FUJABARE) uses the FUJABATS plug-in architecture [BGN+03] to reuse and combine existing tools. Figure 4.32 shows an overview of the REDDMOM tools for data component extension. The Clustering Tool (BUNCH) in grey is available from the Drexel University in Philadelphia. The dashed part of the web information system expresses the generated parts. The tools inside the grey area are provided by FUJABARE. The remaining tools were constructed in REDDMOM.

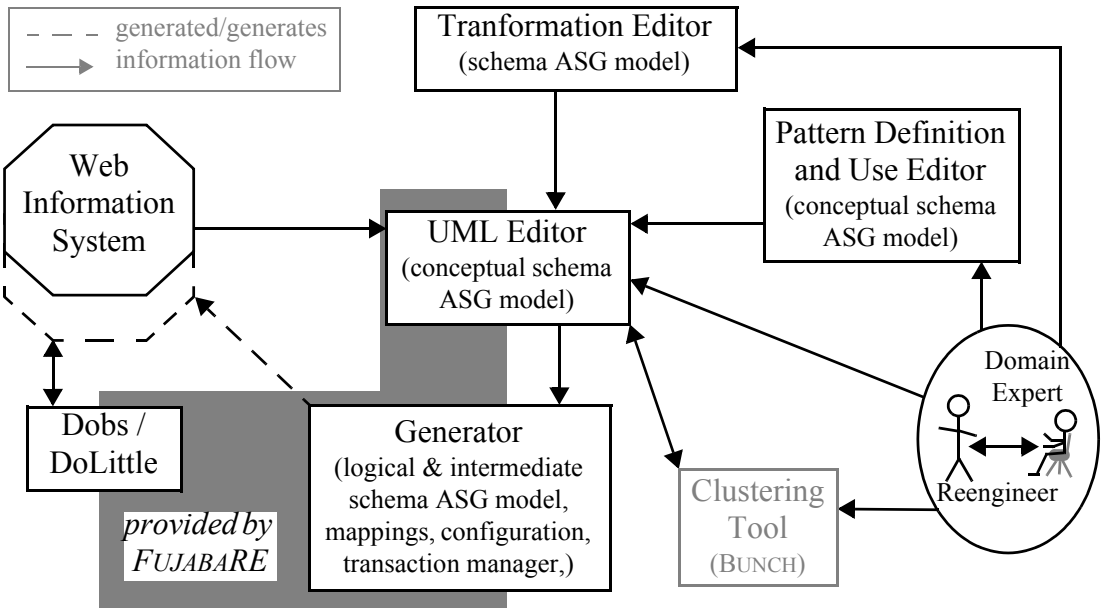


Figure 4.32: Architecture of the REDDMOM extension tools

UML Editor

The UML Editor is currently extended with UML package diagrams and consists of UML class diagrams, views and story diagrams as described in Section 3.4. Packages can contain other packages or a UML class diagram view. Further, REDDMOM introduces the possibility to import and export UML class diagrams (data object models) as (XML) documents, e.g., in XMLSchema or GXL [HWSS00]. Further, REDDMOM extended the story patterns with the <<search>> stereotype construct.

Transformation Editor

The Transformation Editor is described in Section 3.4. In analogy with refactoring operations, design operations can be defined. We give a short overview of the design operations to give the reader insight into (re)design.

Figure 4.33 depicts pushAttribute (in a story diagram) as example for a basic design operation and generalise as example for a composed design operation. pushAttribute is a simple operation, the given attribute attr is pushed from the class clazz to the given sub- or superclass cl. A concrete or an abstract class is created by generalise, followed by pushing the given attributes and relationships to the newly created superclass. The parameters for a composed design transformation are assigned in another dialog, cf. Chapter 3.

Since the design operations change the conceptual schema information capacity, these changes have to be reflected in the data sources. The logical (and physical) schema is updated by the mapping rules (cf. Section 3.2.3), e.g., the new superclass created by generalise is mapped to a new variant of the entity corresponding to the superclass' subclass. Note that also refactoring operations (cf. Section 3.2.4) can be used for implementing composed design transformations.

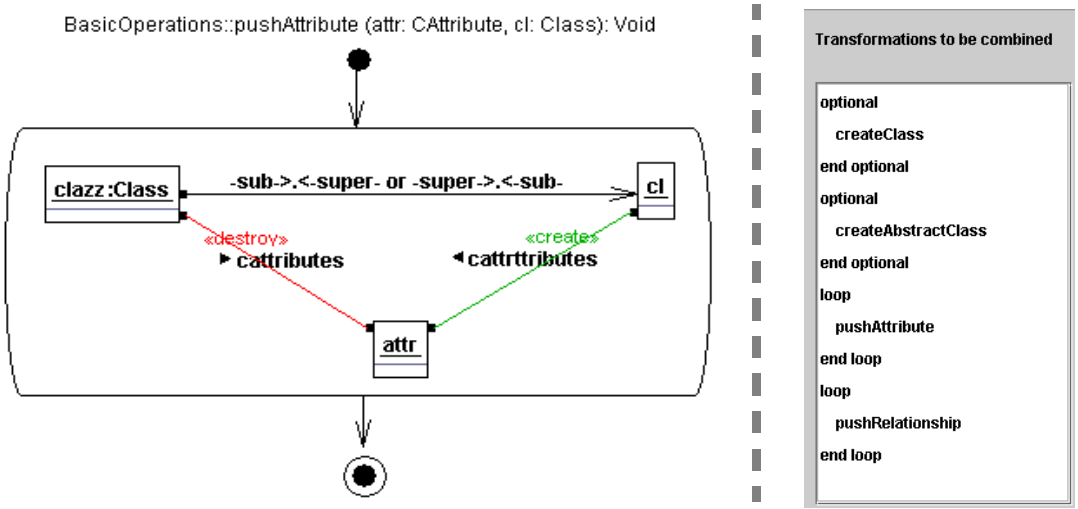


Figure 4.33: Design transformations pushAttribute and generalise

In Table 4.2 we present design operations. The refactoring operations of Table 3.2. are a subset of design operations and thus not listed again. The design operations are classified into basic design (BD) and composed design (CD) operations.

name	description	
createClass	Creates a new (concrete) class.	BD
createAttribute	Creates an attribute in a given class.	BD
createAssociation	Creates an association between two given classes.	BD
createInheritance	Creates an inheritance between two given classes.	BD

Table 4.2: (Re)Design Transformations

name	description	
createKey	Creates a key for a given classes.	BD
createPackage	Creates a new package.	BD
nestingPackage	Interleaves two given packages.	BD
addToPackage	Adds given classes to a given package.	BD
changeRelshipCard	Modifies the cardinality of an association or aggregation.	BD
changeAttrType	Changes the type for a given attribute.	BD
unifyAttributes	Merges two or more given attributes with the same type into one of the given attributes.	BD
convertAbstract	Sets a given concrete class to an abstract class.	BD
convertConcrete	Sets a given abstract class to a concrete class.	BD
pushClass	Moves a given class from its package to a given super- or sub package.	BD
pushAttribute	Moves a given attribute from its class to a given specialisation or generalisation.	BD
pushRelationship	Moves a given association or aggregation from its class to a given specialisation or generalisation.	BD
remove	Deletes a given conceptual schema element.	BD
createAbstractClass	Creates a new abstract class.	CD
generaliseNew	Creates a new concrete or abstract superclass without attributes for a given class.	CD
specialiseNew	Creates a new concrete or abstract subclass without attributes for a given class.	CD
generalise	Creates a concrete or abstract superclass for a given class and pushes up the given attributes and relationships.	CD
specialise	Creates a concrete or abstract subclass for a given class and pushes down the given attributes and relationships.	CD
superPackage	Creates a super package for a given package.	CD

Table 4.2: (Re)Design Transformations

name	description	
subPackage	Creates a sub package for a given package.	CD
removeAll	Deletes all given conceptual schema element.	CD

Table 4.2: (Re)Design Transformations

Pattern Definition and Use Editor

Today, patterns that represent good design solutions to recurring problems are frequently used. The design patterns of [GHJV95] are well known, but patterns for all kinds of application domains exist and can be found, e.g., in [Ris00].

The Pattern Definition and Use Editor enables the reengineer to define (design) patterns. A pattern consists of a structural part which defines classes and structural relationships, e.g. associations, between them and a behavioral part which defines how the classes interact. Since our design takes place on the conceptual schemas we use UML class diagrams for the specification of the structural part of a pattern. The behavioral parts, i.e., the methods, are specified with story diagrams. Finally, the defined pattern is exported to GXL.

The use of a pattern is realised by instantiating it, i.e., integrating it into an actual model of a system. The corresponding GXL description is imported into a UML class diagram. The different constituents of a pattern, e.g., classes or methods, can be matched to existing UML class diagram elements. Some of the classes, associations, and methods of the pattern may already exist in the actual design. The users then has to assign existing design

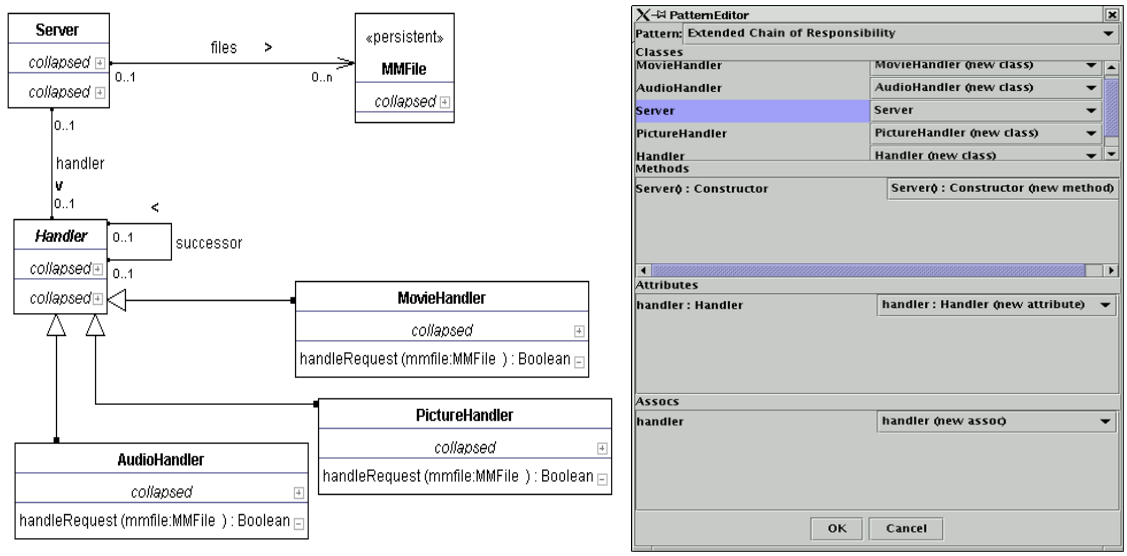


Figure 4.34: Pattern Editor: pattern instantiation example

elements to elements specified by the pattern, i.e., the existing design elements and the pattern elements are merged. The remaining elements of the pattern are then created during the instantiation. This allows a seamless integration of patterns into an existing design. Of course, for newly designed data components or data component parts, patterns can also be instantiated and build the starting point for a new design.

Figure 4.34 shows an example for a pattern instantiation. We refined the chain of responsibility pattern [GHJV95] for multimedia handling. A Server passes the file to process to a Handler which is either an AudioHandler, a PictureHandler or a MovieHandler, cf. the class diagram of Figure 4.34. Given an MMFile and a Server, the extended chain of responsibility is instantiated to handle the MMFile instances. The right part of Figure 4.34 shows the corresponding dialog. It depicts the merging of the existing Server with the pattern's Server class. All other classes are created. In analogy, this is applied to methods, attributes and associations.

Clustering Tool (BUNCH)

BUNCH is a clustering tool intended to aid the software developer and maintainer in understanding, verifying and maintaining a source code base. To do this, BUNCH lets the user evaluate the quality of an application's modularisation, by analysing the source code graph. BUNCH relies solely on the information contained in a module dependency file, considering nodes as program units or modules, such as files or classes, and edges between the nodes as calls or relationships between those modules, such as function calls or inheritance relationships. With this graph, BUNCH can find what a "good" clustering for the system is (thus helping when documentation of the code is nonexistent or outdated), and it can also use predefined clusters to measure or improve the quality of the system's clustering. The BUNCH tool is described in [MMCG99], more details can be found on the BUNCH web pages (http://serg.cs.drexel.edu/SERG/Projects/Bunch/).

Generator

The Generator for the different layers is based on the Java code generation included in FUJABARE. Extensions in REDDMOM were done for the database access layer (persistent objects and multimedia objects), the servlets and the data access interface code generation. Details for these extensions are described in [Wol01], [Wag01] and [Kam03].

Dobs / DoLittle

To facilitate simulation and graphical debugging of the generated application, the FUJABATS environment provides the DOBS (*D*ynamic *O*bject *B*rowsing *S*ystem) plug-in. DOBS uses Java runtime type information and dynamic method invocation features for analysing Java runtime object structures and for the user driven execution of application specific methods. DOLITTLE is an extension of DOBS for the transactional access layer, i.e., to browse persistent objects enclosed in transactions with DOBS. For details we refer to [Wol01] and [GZ02].

4.6 Related Work

Techniques to determine clusters (subsystems) using source code component similarity [MOTU93, SH94, CS90], concept analysis [LS97, vK99, Anq00], or information available from the system implementation such as module, directory, and/or package names [AL99] are available. An introduction and overview to clustering can be found in [Wig97]. Mostly clustering is used for software reverse engineering and re-modularisation [DB00, SMB+03]. The BUNCH software clustering system [MMR+98, MMCG99], unlike the other software clustering techniques, uses search techniques to determine the subsystem hierarchy of a software system, and has been shown to produce good results for many different types of systems.

In the context of data models clustering is used for ER schema abstraction [FM86]. Other approaches facilitate user comprehension [TWBK89], consider relationship abstraction and clustering [JOS93] or cluster relations for databases reverse engineering [SPPB02]. All these approaches have in common that they focus on large ER schemas and consequently on intra-schema relationships.

Architectural patterns are related to the notion of architectural styles [SG96]. Architectural styles do not result in a complete architecture but can rather be seen as an architectural framework. Architectural styles are specific views for one subsystem at different levels of a system. In contrast, architectural patterns [BMR+96] are problem oriented. They express and separate the concerns of fundamental structures in software systems

The *Façade* pattern defines and provides a unified interface to a set of interfaces in a subsystem [GHJV95]. The presented Data Portal pattern is an architectural variant of the Façade design pattern. Data Portals provide unified interfaces to data components of the system. A possible technology-driven instantiation of Data Portal is the *Abstract Database Interface* pattern [ABM96]. Abstract Database Interface makes an application independent from the underlying database platform.

Buschmann at al. [BMR+96] define two architectural patterns that are related to Data Connection. The first pattern is called *Broker* and coordinates communication of decoupled components that interact by remote service invocations. The second pattern is *Pipes and Filters* that provides filter components which encapsulate processing steps for data streams. The pipes pass through the data between adjacent filters. Further, Gomaa, Menascé and Shin [GMS01] described component interconnection patterns for synchronous, asynchronous and brokered communication. A designer of a new distributed application can use their patterns for appropriate component interaction. In contrast to those communication centric patterns, Data Connection mediates data and can be compared to the *Mediator* pattern [GHJV95] at architectural level.

In terms of extension, only few approaches tackle the problem of system migration. Behm et al. [BGD97] and [Fon97], e.g., propose a complete replacement of relational by object-oriented databases. Based on the schema mappings, they present algorithms to automati-

cally migrate the data. According to [War99], system „replacement“ is an expensive and risk-prone task that should only be considered when continued maintenance or reengineering is not feasible.

Data component extension is based on reverse engineered data models, thus in this thesis we focus on integration of the data related legacy system parts. A prerequisite that integration has to meet in order to fulfil the new technology requirements is to enable the legacy system to act as components, i.e., black boxes that exchange data. Sneed presents an approach to wrap legacy systems with interfaces exchanging the data via XML [Sne02]. A prerequisite of this approach is to reengineer the existing interfaces. An approach combining model-integrating computing with component middleware is presented [GSNW02]. The advantage of this approach is that components can be reused and the modelling tools have only to generate the interfaces. The drawback is that this process starts with the integrated data models.

The Clio project is a system that manages data transformation and integration [MHH+01]. It supports the generation and management of schemas, correspondences between schemas and mappings (queries) between schemas. The weakness of Clio is the inter-schema correspondences mining that could be augmented as stated by the authors themselves. Schema mappings are the basis for providing uniform data access to heterogeneous distributed database systems. Many approaches use object-oriented data models to integrate multiple databases, e.g., [RL82, ADD+91, CHK+91, FGS93]. The TSIMMIS project [PGMW95] strives to integrate a broad range of repositories. The Garlic project [CHS+95, HMN+99] provides an object-oriented view, not only to databases and record-based files, but also in media-specific data repositories with specialised search facilities. The integration itself in form of wrappers and mediators [Wie92, Wie95] are poorly supported.

The InterDB approach [THB+98] integrates the construction of abstract interfaces to access independent heterogeneous distributed databases in DB-Main. It provides generation of conceptual wrappers, i.e., software layers that interface a database based on the recovered conceptual schema [TCHH99]. These wrappers are based on data conversion programs which consist of a concatenation of the applied schema transformations. The drawback here is that this approach can hardly be used as input for commercial off-the-shelf middleware because the schema correspondences are implicitly defined in the data conversion programs. Approaches like [Sie98, CER99, Tho99, Obj99b, Obj99a] provide such support based on previous determined schema mappings.

Remaining problems are the construction of such schema mappings and that the reengineer is not aware of the overlapping schema information. The VARLET approach [Jah99] maintains an explicit schema mapping during the reverse engineering of single schemas that can serve as input. Still, given schemas of different data sources have to be related.

Schema integration and matching approaches go in this direction. Schema integration [RR98] mainly tackles the reverse engineering problem of determining inter-schema re-

lationships. Schema matching provides a mapping between two schemas that semantically correspond to each other. A detailed overview on existing partially automated schema matching techniques is given in [RB01]. Schema matching techniques are used for data integration and to detect schema overlapping. The drawback of most tools is that they assume semantically similar schemas and/or are focussed on one-to-one matching.

An approach that defines model mappings based on semantics approximation is presented in [Köl98]. The mappings form the basis to establish a cooperative coexistence of new and legacy information systems. The weakness of this approach lies in the manual analysis of the legacy systems to derive a flat reverse object-oriented model.

For multimedia extension, Cybulski and Linden [CL00] describe a pattern language that is used to define a multimedia authoring environment capable of producing and utilising multimedia components. Each of the presented patterns describes one well-known approach to multimedia authoring, e.g. joining and breaking artefact groups, defining and filling in templates, arranging and re-arranging artefact collections, creating and holding presentations, synchronising multiple multimedia channels, etc. Van den Broecke and Coplien propose a pattern-based approach to design software for the rapidly changing field of multimedia networking [vC97]. Engels and Sauer [ES02] present an approach to object-oriented hypermedia and multimedia modeling with OMMMA-L a visual, object-oriented modelling language that is an extension of the UML. To search in large amount of information, Manolescu works with an alternative, a simpler representation of the data, i.e., the representation contains only the information that is relevant for the problem at hand [Man99].

4.7 Summary and Future Work

In this chapter we presented our approach to extend the web information system. The chapter started with a process overview. The process starts with the clustering of data component models based on the BUNCH Tool [MMCG99]. After discussing different clustering strategies, the data components resulting from clustering are classified in three kinds. This classification is followed by (re)structuring the web information system based on the presented architectural patterns. The maintained data component model dependencies enable the generation of an object-oriented transactional access layer to the legacy databases. Next, a short overview for modelling prototypes and multimedia extension was given. Finally, we presented the tool support for this process.

A consequent step is the integration of patterns that enable further generation and thus model execution, like presented in [HK02] for Enterprise JavaBeans or in [LB03] for ubiquitous computing. Further, architectures of web information systems are constantly in flux and organisations have to be able to respond quickly to changing requirements. This engenders component-oriented development as presented in [CWMY02] or model-driven development of web services [Amb02].

CHAPTER 5: MODEL CONSISTENCY MANAGEMENT

Historical knowledge is indispensable for those who want to build a better world.

LUDWIG VON MISES
Austrian economists (1881 - 1973)

5.1 Data Component Model Consistency Maintenance

In the two previous chapters, we present reengineering activities for web information systems that are based on their data component models. These models represent the old and new data components at different levels of abstraction. The models depend on each other. Inconsistencies between those models often cause update problems. Whenever a reengineer discovers new information, e.g., about the real semantics of implementation constructs in the legacy physical schema, the conceptual representation that has been created so far must be updated accordingly.

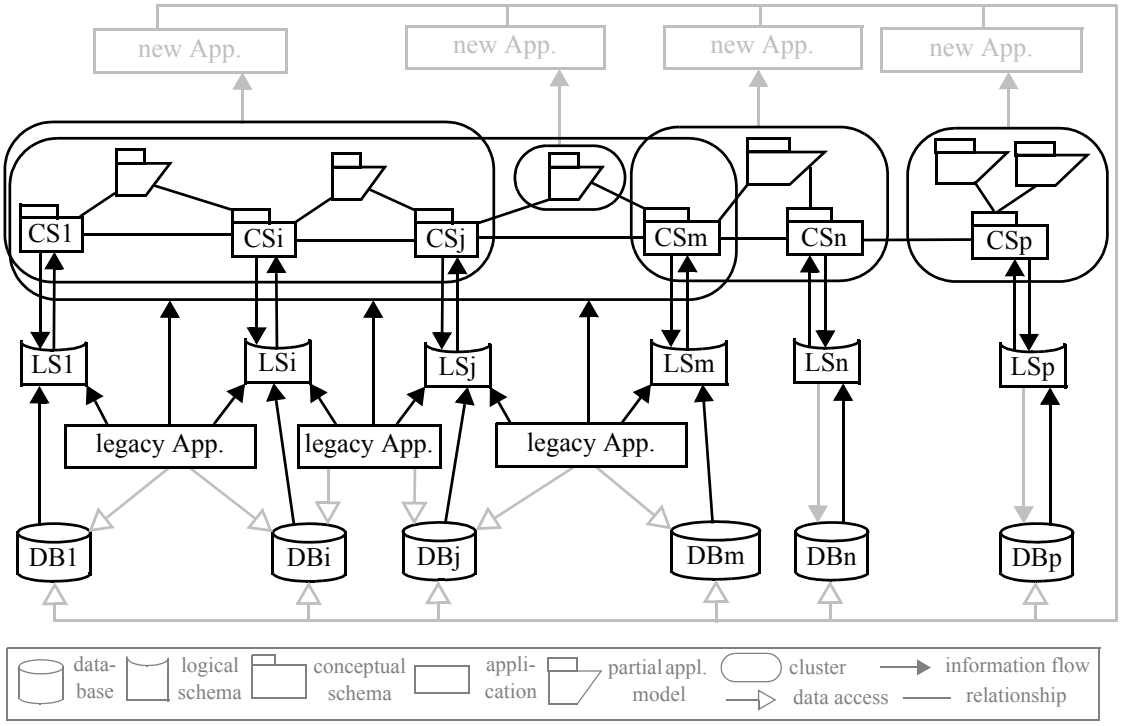


Figure 5.1: Web information system model maitenance

Figure 5.1 shows how the models depend on the databases and legacy applications; and it shows the dependencies and relationships between models. A typical source of inconsistencies are on-the-fly modifications to the implementation of the legacy data sources due to urgent requirements while reengineering activities are in progress. These iterations during the reengineering activities that imply model changes lead to inconsistencies between the different models. Detecting and eliminating such inconsistencies manually is a time-consuming and error prone activity. Hence, a commonly used alternative is to discard all created (conceptual) models of the system and generate default representations anew. In this case, the reengineering work that has been performed manually so far is lost and has to be repeated. Obviously, both alternatives are unsatisfactory.

The overall goal (of our research) is the development of mechanisms that facilitate incremental and iterative reengineering processes. Traceability is the most important prerequisite for such a mechanism, i.e., in case of process iterations, we have to be able to trace and propagate modifications of the initial representation of the legacy system to its transformed representation in order to re-establish model consistency. The requirement for traceability implies some sort of "logbook" about the interdependencies of all operations invoked by the environment, i.e., a reengineer (user) or a tool.

We ensure tracability inside and between the models representing the web information system. This means that all reengineering activities have to be model-based. Since our models are maintained as abstract syntax graphs, only modifications performed with graph productions are considered for tracability. Therefore, this only holds for model-based reverse engineering and model-driven development for the extension activities, e.g., manual code changes cannot be considered. Further, the legacy applications cannot be updated because we lack their models and thus tracability for them.

We have developed an incremental approach to traceability in graph-based activities. It turns out that graph-based structure is most suitable to maintain the interdependencies. We call the corresponding graph History Graph because it reflects the transformation history of the different models. The developed *History Graph Mechanism* provides the traceability necessary for iterative (reengineering) processes. The core idea is an explicit dependency relation among all performed graph transformations. Let us assume that a software artefact has been updated during the iteration. The dependency information stored in the proposed History Graph Mechanism enables the selective removal of only those transformations that (transitively) depend on the changed information.

Figure 5.2 shows an overview of the History Graph Mechanism that provides model consistency management. The Data-Oriented Reverse Engineering and Data Component Extension activities transform the models (abstract syntax graphs) representing the web information system data components. The History Graph Mechanism receives as input a start graph and History Graph Transformations, i.e., all executed operations that modify the web information system data components. To re-establish consistency between mod-

els the reengineer runs the History Graph Mechanism. The output produced is on the one hand the consistent Updated Models and on the other hand the Undone History Graph Transformations. The Updated Models can then be modified, e.g., reapply adapted undone operations, and further data-oriented reengineering activities can take place.

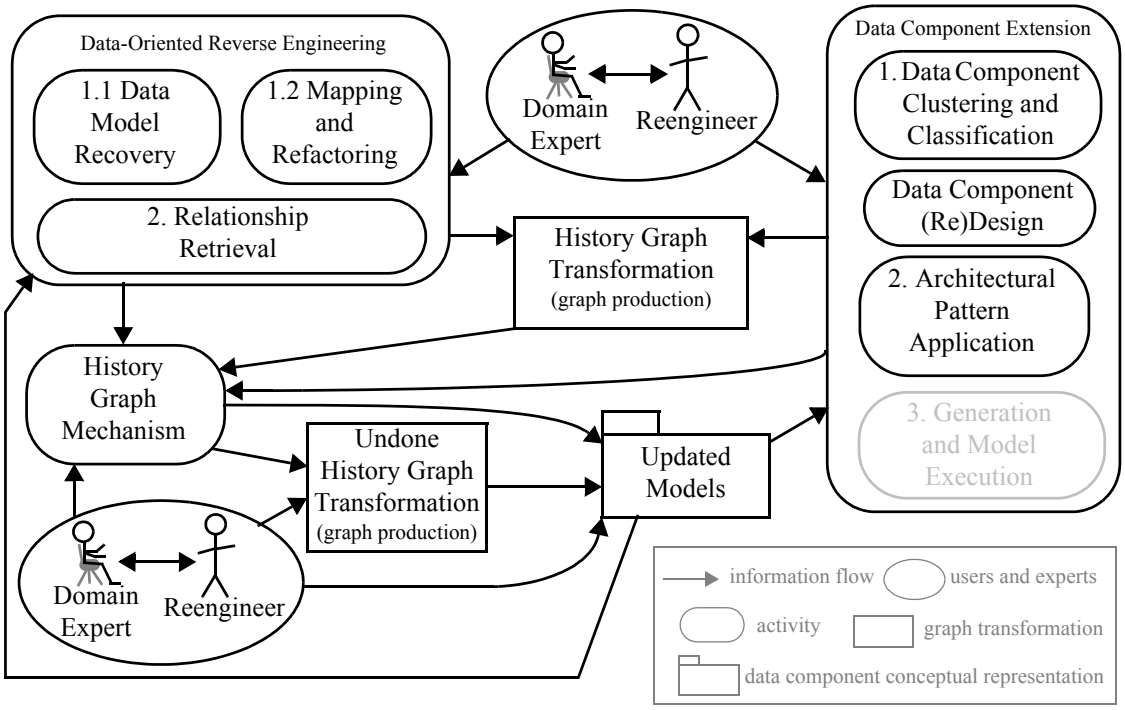


Figure 5.2: Model Consistency Management: Overview

Note that the History Graph Mechanism establish consistency for the models. The consistency to the generated code has to be established by generating the code anew.

5.2 Graph-based History Mechanism

We have employed the concept of History Graphs for consistency management during iterations in reengineering processes since 1998. We present background knowledge and basics before we explain the History Graph Mechanism in more details.

5.2.1 Background: History Graph Mechanism

Assume that midway during a database reverse engineering process the reengineer learns about additional dependencies in the legacy database. Ideally, (s)he would like to add this new information to the initially extracted representation of the legacy database and validate those operations which might have to be undone due to the new knowledge. The History Graph Mechanism was developed exactly to serve this purpose. With its help the reengineer can get back to the initial representation of the legacy database, make all de-

sired changes, and determine which operations might have lost their validity [JW99]. In the VARLET project [Jah99], we implemented a first History Graph Mechanism prototype [Wad98]. This prototype was tightly integrated with the VARLET tool and could not be reused by other existing tools.

Therefore, we have developed a lightweight History Graph Mechanism that can loosely be integrated with existing reengineering tools [JWZ02]. Graphs are commonly used by reengineering tools for internal representation of software artefacts. Of course, the specific graph models used in different tools might comprise variations with respect to their expressiveness. Nevertheless, most graph models have in common that they support different node- and edge types, attributes and directed edges. We use the GXL graph model [HWSS00]. Note that the History Graph Mechanism does not depend on this particular model.

The History Graph Mechanism makes use of the theoretical concept of graph transformation systems [Roz99]. The History Graph Mechanism is a specific implementation of the general concept of a graph process as introduced by Corradini et al. [CMR96]. Corradini defines a graph process as a partially ordered structure, plus suitable mappings which relate the elements of this structure to those of a given typed graph grammar. This theoretical basis allows us to uniformly describe various kinds of processes in terms of graph processes.

History Graph Model

The History Graph Model is based on the GXL graph model. In the following we give a brief, semi-formal introduction to the GXL graph model. We refer to [Roz99] for a complete formalisation of attributed graph models. A more in-depth discussion of GXL can be found in [HWSS00]. Figure 5.3 shows a UML specification of the GXL graph model.

A Graph can be directed or undirected, defined in isdirected from *LocalConnection*. It contains GraphElements in form of Nodes, Edges, and Relations. Relations contain a set of links pointing to one graph element each. All graph elements have unique identifiers. GXL also includes the notion of hypergraphs, i.e., graph elements can themselves contain sub-graphs. Furthermore, GXL supports typed graph elements (where the type of a graph element is itself a graph element of a type graph). We omit a more detailed discussion of the GXL typing concept since it is not necessary for the History Graph Mechanism. Note that the actual exchange of GXL graph instances is performed in a canonical textual format based on XML [HWSS00].

Further Figure 5.3 shows the graphical specification for the History Graph model. We extend the GXL graph model for maintaining the History Graph: we introduced Transformation nodes that carry a timestamp and a name. The timestamp is used to log the time when a History Graph Transformation has occurred. The name of the corresponding op-

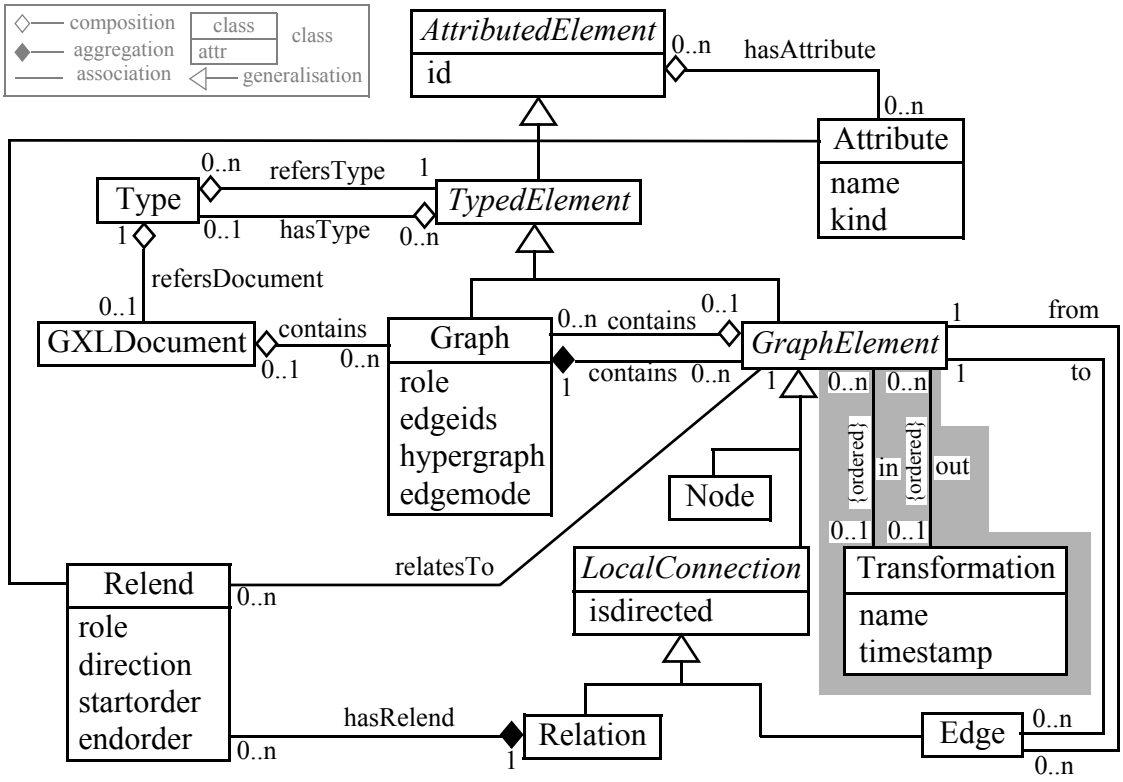


Figure 5.3: History (GXL) Graph model

eration is logged within the name attribute. Input and output dependencies are represented by the ordered associations in and out.

History Graph Transformations

We use graph productions (cf. Section 3.2.3) to formalise and implement History Graph Transformations. The left-hand side of a graph production represents the input of the corresponding (History Graph) transformation, while its output is represented by the right-hand side. For example, let us consider splitClass (cf. Figure 3.19) in Figure 5.4.

Unique identifiers of the nodes allow identifying identical nodes on both sides of the production, e.g., cl is passed as input parameter and a bound object on both sides. Node cl represents the class to be splitted. Nodes attrs represent the set of attributes that will be moved to the new class (Node newClName: Class). Nodes ca: cattributes represents the edges from the class cl to the attributes attrs. These edges do not occur an the right-hand side of the production. Consequently, these edges will be deleted during the productions application. On the right-hand side new nodes occur. Node newClName: Class represent the new class. Node assocName: Association represents the new association between the old class cl and the new class newClName: Class. The source and target edges are represented by the node :src and node :tar, respectively. Finally, the edges :cattributes between

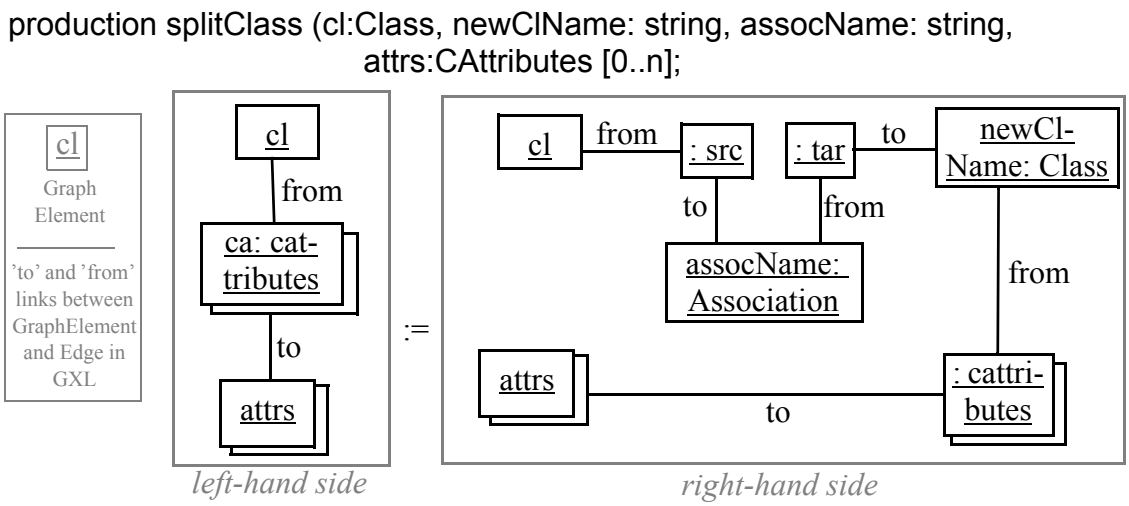


Figure 5.4: Graph production splitClass

the new class cl and the attributes attrs are created. Note that an identifier is not mandatory for nodes that are created.

In order to maintain the application contexts of transformations, i.e., if we want to maintain input/output dependencies of applied transformations, we have to store information about the matches for the corresponding graph productions. We identify the graph elements of the left- and right-hand sides and represent them explicitly in the History Graph. Figure 5.5 shows the template for transformations.

GraphElements that appear on the left-hand side but not on the right-hand side, i.e., deleted elements, are associated to the Transformation node by in links. We call such nodes *consumed* History Graph elements. In opposite, GraphElements that appear on the right-hand side but not on the left-hand side, i.e., that are created, are associated to the Transformation node by out links. These nodes are called the *current* History Graph elements. A node that appear on both sides is associated to the corresponding transformation by an

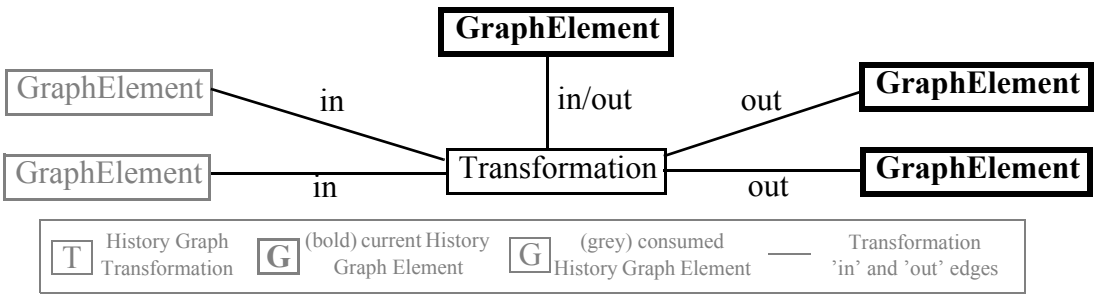


Figure 5.5: Template of History Graph Transformation

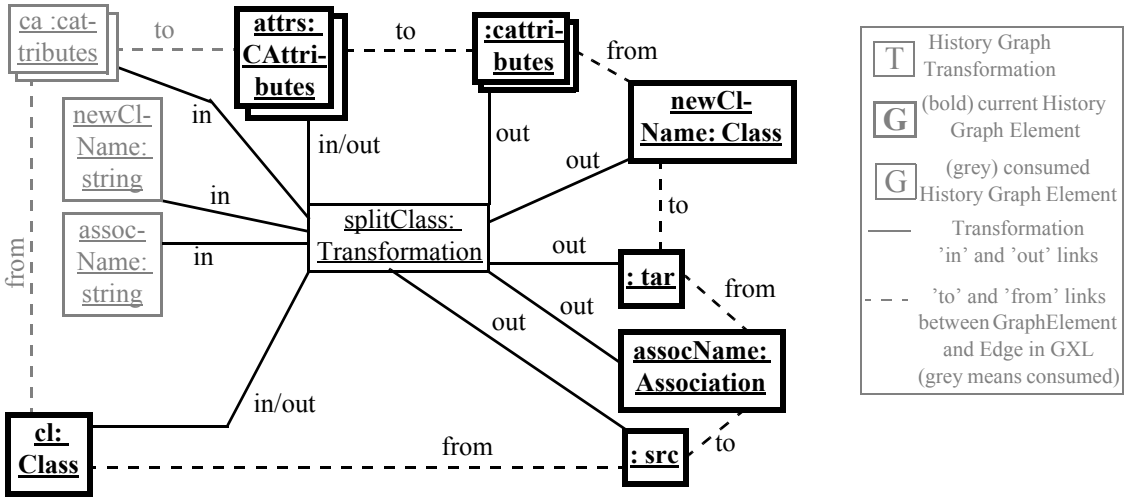


Figure 5.6: History Graph Transformation splitClass

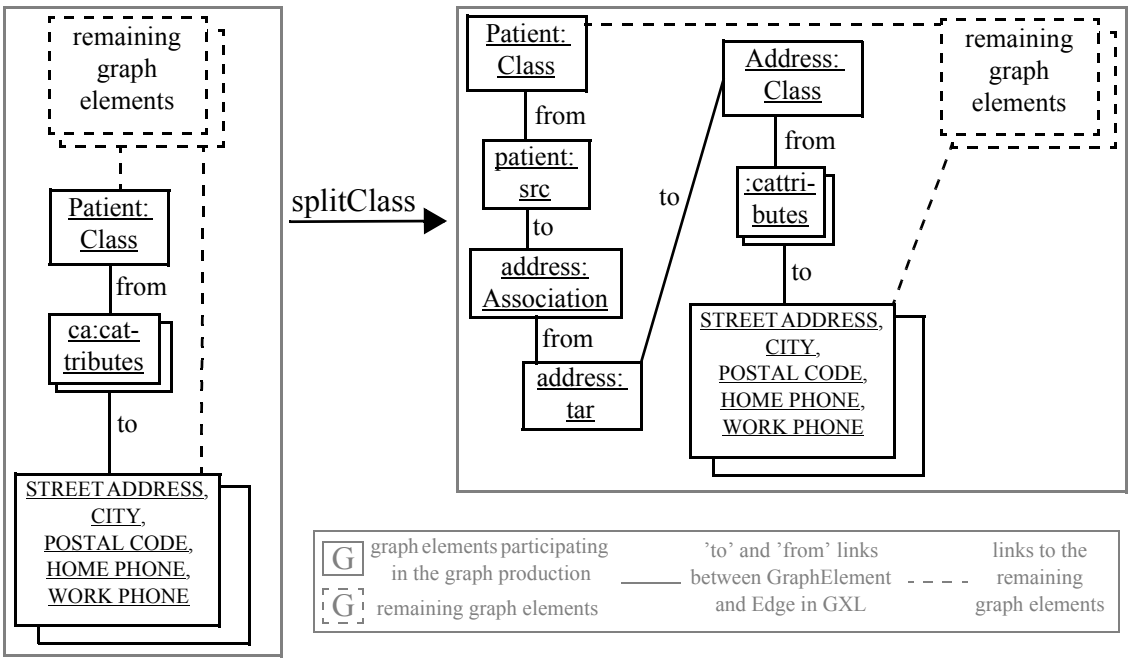


Figure 5.7: Application of production splitClass

in link and by an out link. We represent this situation with a single link labelled in/out. These nodes are also current History Graph elements.

The corresponding transformation to the graph production splitClass example of Figure 5.4 is depicted in Figure 5.6. The splitClass: Transformation has five input nodes and seven output nodes. In addition to the three left-hand side nodes of the graph production, the two (string) input parameters are represented as (consumed) nodes in the trans-

formation. This is necessary to maintain the entire application context of the transformation. The current nodes are identical to the right-hand side nodes of the graph production.

Application of History Graph Transformation

In analogy with the treatment of transformations as graph productions, we can view transformation applications as applications of graph productions. As an example, Figure 5.7 shows an application of transformation splitClass to class Patient. The left-hand side of Figure 5.7 shows the left-hand side subgraph that fulfills all left-hand side application conditions of the graph production splitClass. A set of attributes (STREET ADDRESS, CITY, POSTAL CODE, HOME PHONE and WORK PHONE) is linked to class Patient via the edge :cattributes and link from. This match is depicted by the three nodes with plain border. The remaining graph is represented by the remaining graph elements with a dashed border. Links are represented accordingly. The right-hand side of Figure 5.7 shows the subgraph that contains all elements that occur on the right-hand side of the graph production splitClass. In this example class Address and Association address, with roles (edges) patient and address, are created. The attributes are linked to class Address.

Creating Reengineering Histories with Graph Processes

Graph transformation systems are typically defined as a start graph and a set of graph productions. Graph elements on the left-hand side of a graph production that do not appear on the right-hand side are deleted. The main difference to the History Graph Mechanism is that deleted nodes are not removed (from the History Graph) but are isolated, i.e., all their incomming and outgoing links in the corresponding abstract syntax graph are deleted and only the links to the transformation nodes are maintained.

Figure 5.8 illustrates the basic structure of a History Graph: applied transformations are explicitly represented by n-nodes with corresponding input and output edges to graph elements (G-nodes). The number n gives the chronological order of the transformations according to their timestamps. The current valid graph is composed of the nodes with bold borders. Consumed graph elements, i.e., graph elements that are isolated during transformation application, have grey borders.

The History Graph, which represents a particular model transformation history, contains the current abstract syntax graph of the web information system as a subgraph. This subgraph can easily be determined by filtering all graph elements that have not been consumed by transformations, i.e., that are not exclusively sourced by an in edge that points to a transformation node (bold G-nodes in Figure 5.8). Likewise, the History Graph also subsumes all previous states of the abstract syntax graph during the interactive transformation process. Thus, it is possible to trace back to any past state in the editing history.

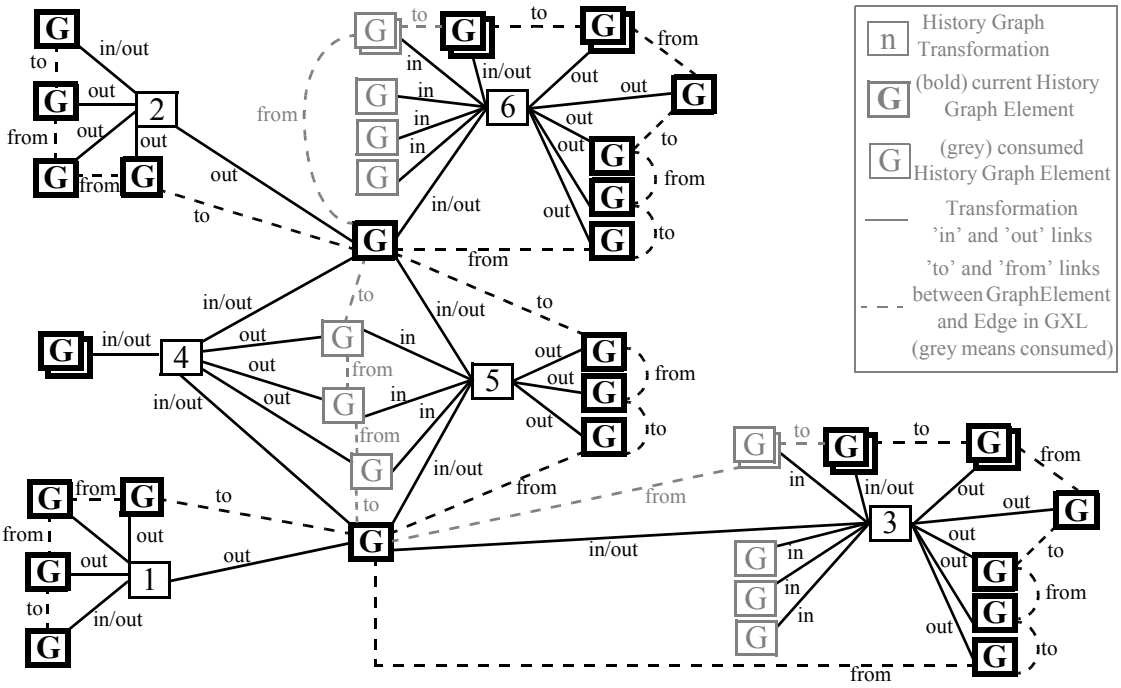


Figure 5.8: Basic structure of a History Graph

Formal definitions of the History Graph, History Graph Transformations (graph productions) and the application of transformations in the History Graph can be found in [Jah99, JWZ02, JSWZ02].

Example

Figure 5.9 shows a snapshot of a History Graph of our case study. It is the same graph as depicted in Figure 5.8, where the graph elements and transformations are represented as G-nodes and n-nodes, respectively.

Starting points are the variants Deceased and Patient of the corresponding entities. The mapping rule MapVariantToConcreteClass is applied for both variants. The corresponding nodes, especially classes Deceased and Patient, are created (cf. Section 3.2). Both mapping rules are stored in the History Graph as transformations. We replaced the timestamp of the six transformations occurring by numbers from 1 to 6 according to the chronological order they were applied, i.e., transformation with timestamp {1} was applied before transformation with timestamp {2}. Next, the transformation splitClass is applied on class Patient (cf. Figure 5.7). During recovery activity the Annotation 'Duplication' between the classes Deceased and Patient is created. We omit the precondition details and represent them as one node set. The Annotation 'Duplication' is confirmed and replaced by the Association 'deathdate' with stereotype <<copy>>, cf. Figure 3.38 and Figure 3.39. Finally, a second transformation splitClass is applied on class Deceased.

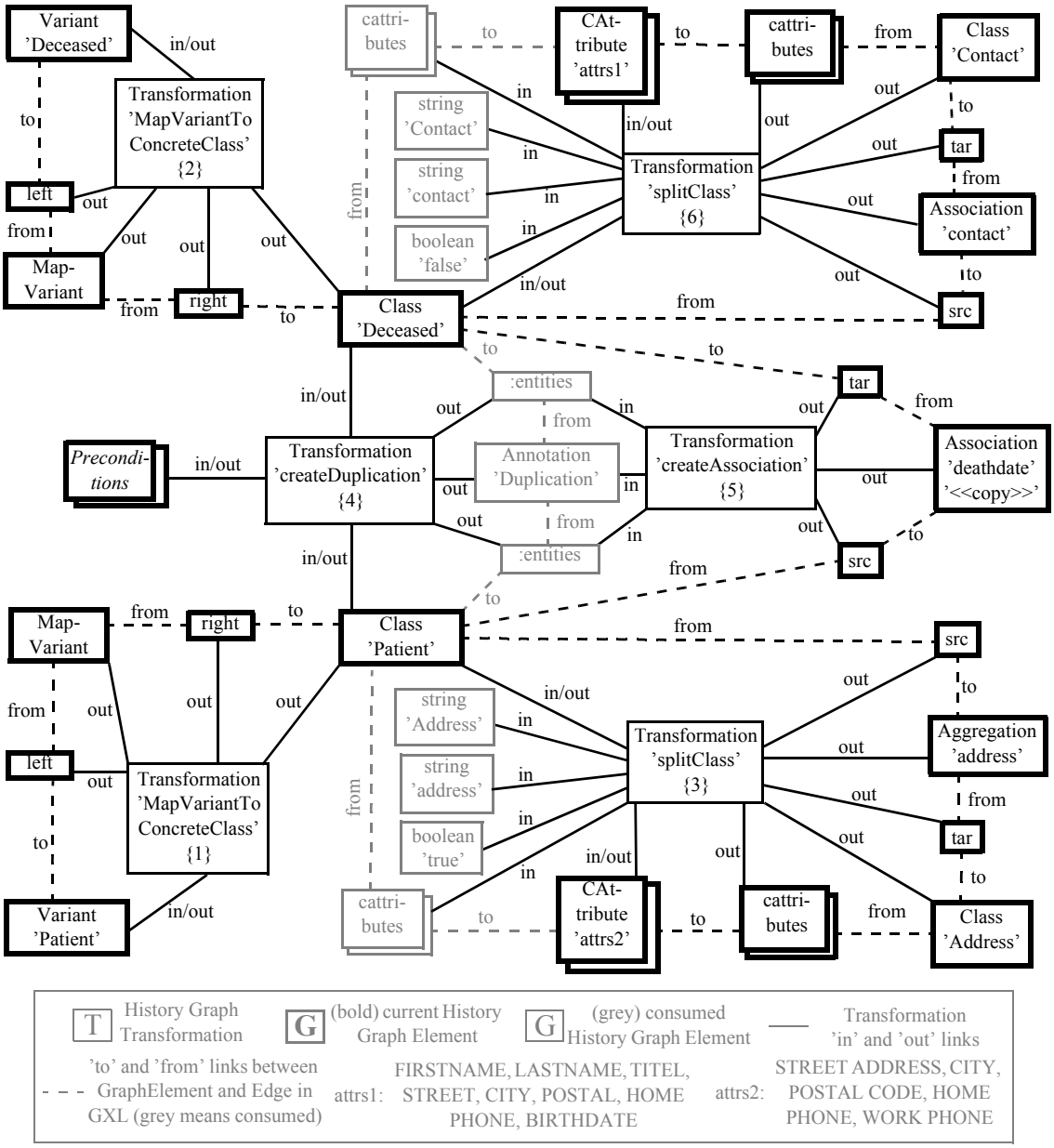


Figure 5.9: Sample History Graph

5.2.2 Simple Undo History Graph Mechanism

In this section, we describe how the History Graph can be used for incremental undo (change propagation). The sequence diagram in Figure 5.10 illustrates an interaction scenario between the environment (Tool/Reengineer) and the History Graph Mechanism. Note that the diagram covers a single iteration only.

The process starts with the creation of the start graph by passing it to the History Graph Mechanism (addGraph(startGraph)). During this operation a copy of the startGraph is made and stored. Then, a sequence of transformations is performed. These transformations are logged in the History Graph. This is done by calling function addTransformation(leftGraph_i, rightGraph_i, name_i), where leftGraph_i and rightGraph_i denote in- and output of the transformation with name_i, respectively. For each call, the History Graph Mechanism automatically creates the transformation node and returns trafo_i, the signature (name and input parameters) and the timestamp, to the environment.

At any point in time, the valid graph before the execution of trafo_j (currentGraph) or the transformation history ({trafo_1, ..., trafo_n}) can be requested. When invoking the createHistory() operation, the list of all valid applied transformation {trafo_1, ..., trafo_n} is

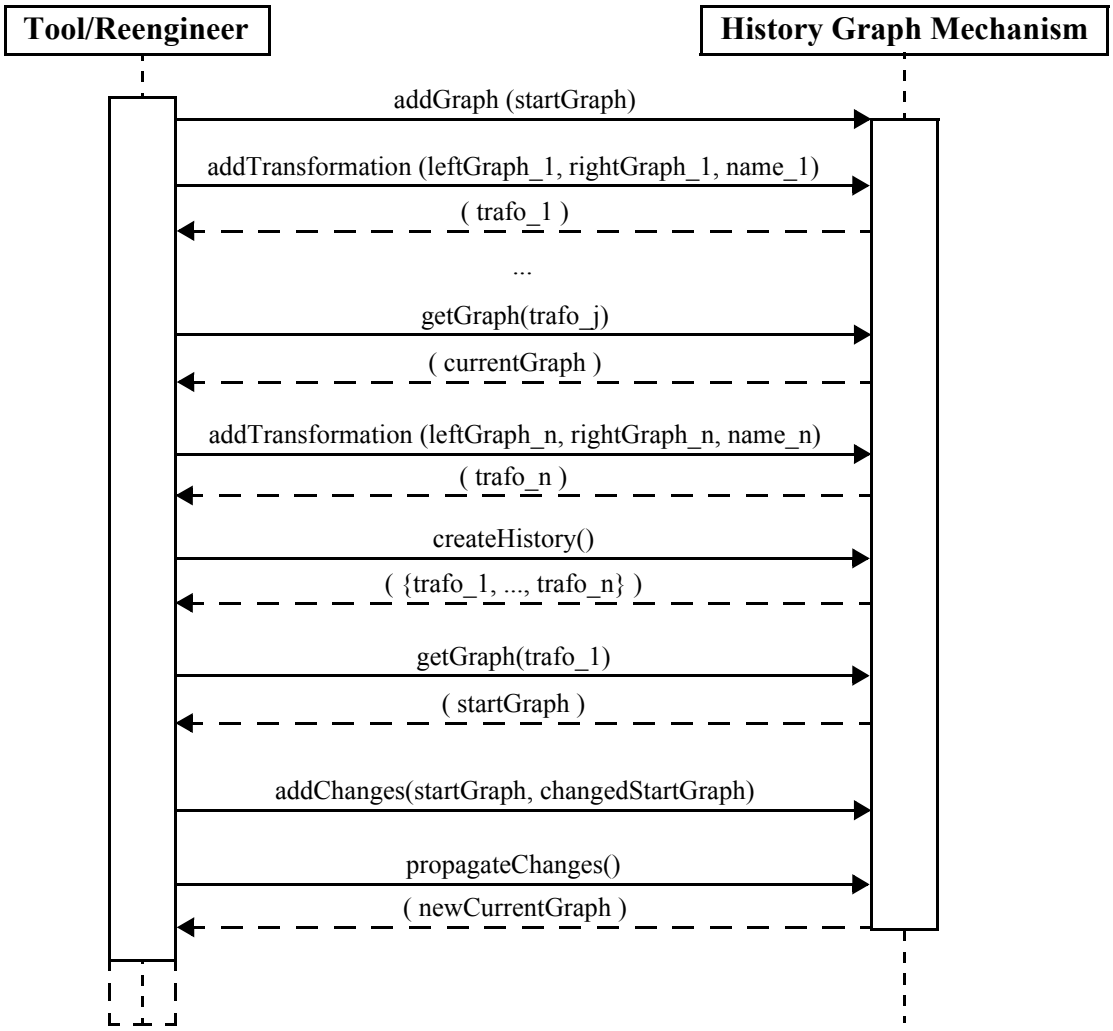


Figure 5.10: Interaction with the History Graph Mechanism

returned. Based on the transformation list, any intermediate graph can be restored by calling the getGraph(trafo_i) operation.

When an iteration of the reengineering process is required, the initial startGraph can be restored by invoking getGraph(trafo_1). Since trafo_1 is the first transformation, the valid graph before the execution of the first transformation is the initial startGraph. Subsequently, the user can perform all necessary additions and/or modifications to the initial graph of the legacy systems and submit the changes (addChanges(startGraph, changedStartGraph)). Likewise, the user may perform changes to any intermediate graph by restoring it, by editing it and finally by submitting the changes. Now the consistency can be re-established through an incremental undo of all transformations that depend on the changes to the start graph (propagateChange()). The result of this operation is the current valid graph newCurrentGraph.

The algorithm in Figure 5.11 describes the work performed by the History Graph Mechanism. At the beginning, it receives the start graph from the reengineering tool. This start graph (startGraph) becomes the initial history graph (HG) and further help graphs are initialised. Then, the History Graph Mechanism performs a Loop where iterations can take place. The History Graph Mechanism logs all invoked transformations in HG. If process iterations occur, the History Graph Mechanism recovers the requested graph (resultG) and returns it (line 9). Line 7 specifies that the requested graph resultG can be recovered by taking all graph elements from HG that:

- do not represent transformations (type(g)≠'Transformation'),
- do not participate in any transformation (g.-in->=∅) or
- are the output of transformations,
  - which have been executed before the given transformation (t1.timestamp < trafo_j.timestamp) and
  - which are not consumed by a previous transformation.

All isolated graph elements that are again valid in resultG are restored (line 8 - restoreElements(resultG)). How graph elements can be restored is described in Section 5.2.3.

Once the modifications to the graph are performed, they are sent back as modified graph changedG to the History Graph Mechanism (line 11). Then the affected transformation are calculated and returned as affectedT (line 12). The method calculateAffectedTransformation(HG, originalG, changedG) is presented in Figure 5.12.

The affected transformations (affectedT) and all graph elements that are exclusively output of the affected transformations ({ g∈HG | ∃ t∈affectedT, g := (t.-out-> ∧ ¬(t.<-in-)) }) are removed (line 14). Next, all isolated input graph elements that are valid again are restored; and all superfluous input graph elements of deleted affected transformations are removed (line 15). The resulting current graph updatedG (line 16) is processed in analogy

```
Algorithm Undo History Graph Mechanism
1.    Graph HG, resultG, updatedG, affectedT;
2.    addGraph (startGraph);
3.    HG := startGraph; // startGraph becomes the initial History Graph
      resultG := HG; // initialise the result graph
      updatedG := HG; // initialise the unpdated graph
      affectedT := ∅; // initialise the affected transformations
4.    Loop
5.       addTransformation (leftGraph_i, rightGraph_i, name_i);
         // log all transformations in HG
6.       if (getGraph(trafo_j) )// if the graph before trafo_j is reqeusted
7.       { resultG := { g∈HG | type(g)≠'Transformation' ∧
            ( (g.-in->=∅) ∨ ( ∃t1∈HG, t1:=(g.<-out-) | ∀t2∈HG, t2:=(g.-in->) :
                        type(t1)='Transformation' ∧type(t2)='Transformation' ∧
                        t1.timestamp < trafo_j.timestamp ∧
                        t1.timestamp >= t2.timestamp ) ) };
            // recover the current graph before trafo_j
8.          resultG := restoreElements(resultG);
9.          return resultG;
10.      };
11.      addChanges(originalG, changedG) // receive the changed graph
12.      if ( propagateChanges() )// if request to propagate changes is received
13.      { affectedT := calculateAffectedTransformation( HG, originalG, changedG );
            // determine all affected transformations
14.         HG := HG-(affectedT ∪ { g∈HG | ∃t∈affectedT, g := (t.-out-> ∧¬(t.<-in-)) } );
            // undo affected transformations
15.         HG := restoreElements(HG); // restore or remove isolated elements
16.         updatedG := { g∈HG | type(g)≠'Transformation' ∧
                        ( (g.-in->=∅) ∨ ( ∃t1∈HG, t1:=(g.<-out-) | ∀t2∈HG, t2:=(g.-in->) :
                        type(t1)='Transformation' ∧type(t2)='Transformation' ∧
                        t1.timestamp >= t2.timestamp ) ) };
                        // compute updated current graph
17.         return updatedG;
18.         displayUndoReport (affectedT);
19.      };
20.   EndLoop
End
```

Figure 5.11: Undo History Graph Mechanism

with resultG (line 7) and then retured (line 17). The only difference is that no given transformation has to be taken into account. Finally, a the list of undone transformation displayed (line 18).

Figure 5.12 shows how the input/output dependencies in the History Graph are used to detect all transformation applications which are affected by the modifications in the graph changedG compared to the graph resultG. It requires the calculation of the changeSet which represents the set of all graph elements that have been modified, cf. diff (originalG, changedG) in line 3. First, all transformations that have a changed graph element as input

```
Graph calculateAffectedTransformation( Graph HG, Graph originalG, Graph changedG )
{
1.    Graph changeSet, affectedT, affectedT';
2.    changeSet := ∅; // initialise the changed elements set
      affectedT := ∅; // initialise the affected transformations
      affectedT' := affectedT; initialise help graph
3.    changeSet := diff (originalG, changedG);
      // determine the changes between the two graphs resultG and changedG
4.    affectedT := { t∈HG | type(t) ='Transformation' ∧ ( ∀g∈changeSet, t = g.-in-> ) };
5.    Repeat // collect all following transformations
6.       affectedT' := affectedT;
7.       affectedT := { t∈HG | type(t) ='Transformation' ∧
            ( ∀t'∈affectedT, t = t'.-out->.-in-> ∧t.timestamp >= t'.timestamp ) };
8.    Until (affectedT == affectedT')
9.    return affectedT
}
```

Figure 5.12: Determine affected transformations

are added to the affected transformations set affectedT (line 4). Then, all subsequent transformations, i.e., transformations that are reachable from affected transformations and that have been executed after the affected transformations, are added to the affected transformation set affectedT (line 7). This is done as long as no new affected transformation is found (line 5, 6 and 8). Finally, the set of affected transformation is returned (line 9).

Figure 5.13 illustrates the History Graph Mechanism applied on the case study snapshot from Figure 5.9. We use the representation of Figure 5.8 for layout reasons.

We assume that the History Graph exists and start with the changePropagation() call (cf. line 11 in Figure 5.11). The lifted index ( )represents a changed graph element (Variant 'Patient' in Figure 5.9). Starting from this graph element all transformations depending on it are marked affected as described in Figure 5.12. This is shown by the hand with a pointing index ( ). We marked all affected graph elements with a circle ( ) to enhance the readers understanding. In Figure 5.13 all transformations except '2' are affected. Transformation '2' does not follow an affected transformation, i.e., it has no input graph element that is affected.

The removed graph elements are highlighted with a fist ( ) in Figure 5.14. Note that not all output graph elements of affected transformations are deleted. Output graph elements that are also input graph elements of an affected transformation are retained. Further, an output graph element of an affected transformation, which is also an output graph element of another not affected transformation, is retained.

Obviously, in both cases the graph elements are part of the currently valid graph. Furthermore not all consumed input graph elements are retained. Consumed input graph elements that are isolated, i.e., which have no link to any graph element, are deleted.

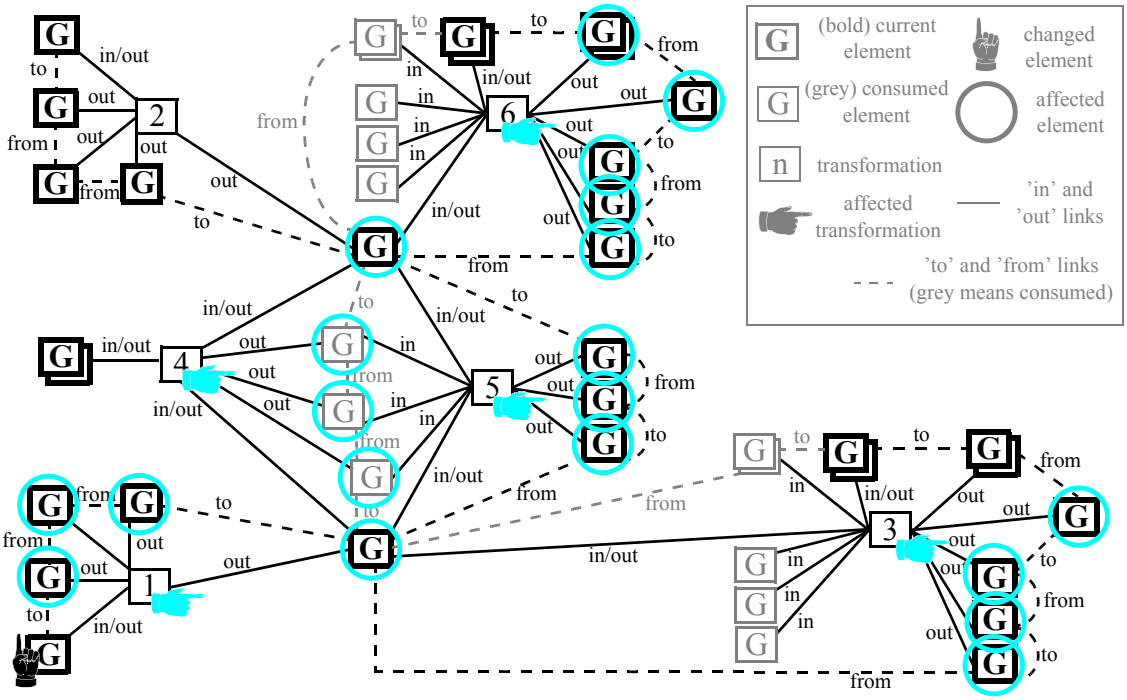


Figure 5.13: Affected History Graph: simple undo

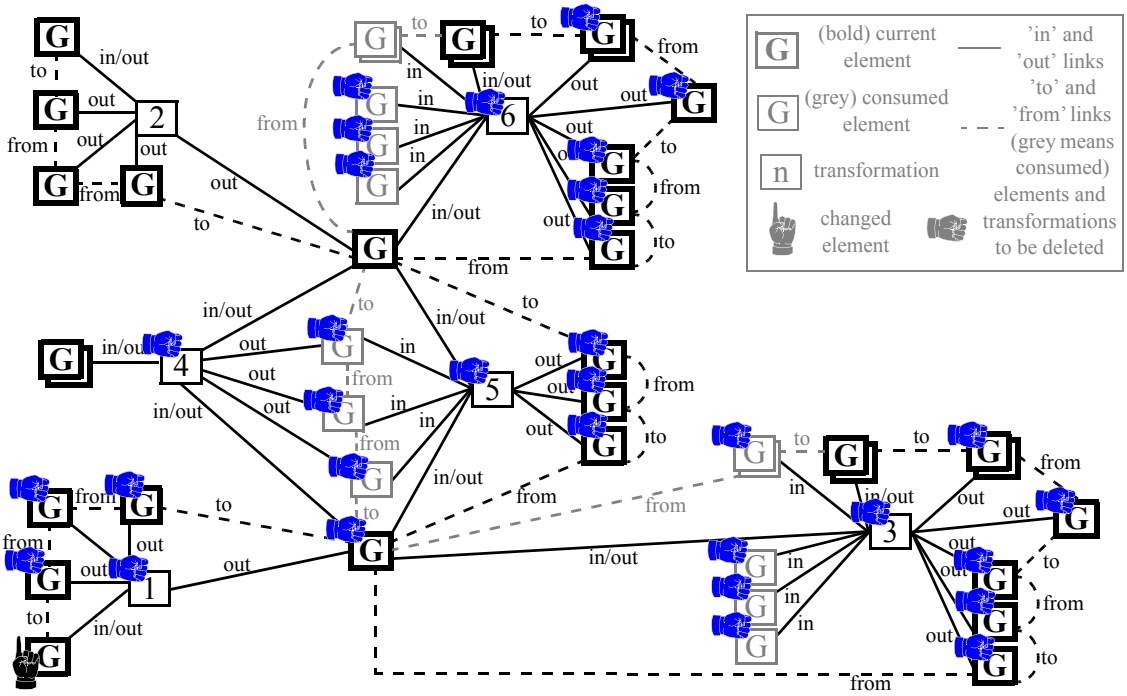


Figure 5.14: Updated History Graph: simple undo

The remaining graph elements (cf. Figure 5.9) are the Variant 'Deceased', the Class 'Deceased' and the corresponding mapping nodes (left, MapVariant, right). Additionally, the Transformation 'MapVariantToConcreteClass' remains. Further the CAttributes 'attrs1' inclusive the edges cattributes, the Preconditions, the Variant 'Patient' and the CAttributes 'attrs2' are not deleted.

5.2.3 Selective Undo History Graph Mechanism

The impact of one little change seems to be massive, but considering the whole palliative care information system only a small part is affected. Nevertheless, transformations were undone that not needed to. Obviously, the splitClass transformation applied to the Class 'Deceased' is still valid since it is not influenced by Variant 'Patient'. To reduce the loss of reengineers work further, we elaborate a selective undo History Graph Mechanism.

The main difference to the single undo History Graph Mechanism is that the affected transformations are not simply undone, i.e., removed, but they are reevaluated. The History Graph Mechanism exports the transformation to be reevaluated to the environment (tool/reengineer) and gets the successful or failed transformation back. The algorithm remains the same as the one of Figure 5.11 except to line 14; restoreElements(HG) can be discarded because transformations are reevaluated, i.e., the consumed input graph elements of transformations that will be valid again are restored before the transformations reevaluation. Further, calculateAffectedTransformation(HG, resultG, changedG) of Figure 5.12 is replaced by reevaluateTransformation(HG, resultG, changedG), cf. Figure 5.15.

Lines 1 to 10 are in analogy with the lines 1 to 9 in Figure 5.12. A changedSet is calculated and all depending transformations are collected in affectedT. These directly affected transformations have to be validated, i.e., are then reevaluated.

However, some of these transformations depend on input graph elements which were consumed or which values were changed by a transformation. These isolated graph elements have to be reproduced before the dependent transformation can be reevaluated. Reproducing these graph elements means to reevaluate all transformations that have been applied to produce them. Some of the transformations that have to be reevaluated might not have been marked because they are not directly affected by the changed graph elements. Hence, we need a further affected transformation collection, i.e., collecting the indirectly affected transformation. This is done in lines 11 to 14. Note that the reapplication of the transformations also reproduces the links between graph elements.

The selection of the transformations, which have to be reevaluated, for the graph elements that have to be restored in the simple undo case is done in analogy. After collecting all isolated input graph elements that are valid again, the indirectly affected transformations are collected and then reevaluated. In case that the graph element to be restored is an initial graph element, i.e., member of the start graph, we use the copy of the start graph. The

```
Graph reevaluateTransformation(Graph HG, Graph originalG, Graph changedG )
{
1.    Graph changeSet, affectedT, reevaluateT, dependendT, dependendT, affectedT';
2.    changeSet := ∅; // initialise the changed elements set
      affectedT := ∅; // initialise the affected transformations
      reevaluateT := ∅; // initialise the transformations to be reeavluate
      dependentT := ∅; // initialise the dependend transformations
      failedT := ∅; // initialise the affected transformations
      affectedT' := affectedT; initialise help graph
3.    Transformation oldT, newT;
4.    oldT := ∅;
      newT := ∅;
5.    changeSet := diff (originalG, changedG);
      // determin the changes between the two graphs resultG and changedG
6.    affectedT := { t∈HG | type(t) ='Transformation' ∧ ( ∀g∈changeSet, t = g.-in-> ) };
7.    Repeat // collect all directly affected transformations
8.       affectedT' := affectedT;
9.       affectedT := { t∈HG | type(t) ='Transformation' ∧
            ( ∀t'∈affectedT, t = t'.-out->.-in-> ∧t.timestamp >= t'.timestamp ) };
10.   Until (affectedT == affectedT')
11.   Repeat // collect all indirectly affected transformations
12.      affectedT' := affectedT;
13.      affectedT := { t∈HG | type(t) ='Transformation' ∧
                        ( ∀t'∈affectedT, (t'.<-out-)∉changeSet : t = t'.<-out-.<-in- };
14.   Until (affectedT == affectedT')
15.   Repeat
16.      reevaluateT := { t∈HG | type(t) ='Transformation' ∧
                        ( ∀t'∈HG, t' = t.<-out-.<-in- : t'∉affectedT) };
            // select all transformations that do not follow an affected one
17.      oldT := t∈reevaluateT; // select one transformation to be reevaluate
18.      newT := reevaluate (HG, oldT);
19.      if (newT ≠∅) // succesful reapplied transfomation oldT
20.      {  replace (oldT, newT); // replace oldT by newT
21.         affectedT := affectedT-{oldT}; // clean up affectedT
22.      }  else // failed reapplied transfomation oldT
23.      {  Repeat // collect all depending transformations
24.            affectedT' := dependentT;
25.            dependentT := { t∈affectedT | ∃g∈HG, g=oldT.-out-> ∧g≠oldT.<-in- :
                                 t = g.-in-> }
26.         Until (dependentT == affectedT')
27.         affectedT := affectedT-dependentT; // clean up affectedT
28.         failedT := failedT ∪{oldT} ∪dependentT;
29.      }
30.   Until (affectedT == ∅)
31.   return failedT;
32.}
```

Figure 5.15: Reevaluate transformations

unique identifiers of GXL enable an unambiguous matching and update of the graph elements in question.

Next, the transformations that do not depend on another affected transformation are chosen (line 16). Reevaluating transformations that do not have actual input graph elements is superflous. Out of the transformations with actual input, one transformation oldT is selected and reevaluated (lines 17 and 18). Reevaluating a transformation means to apply the corresponding transformation anew to the current (maybe changed) graph elements. The reevaluation itself is performed by the environment (tool or reengineer) in order to check the application constraints. The new transformation newT is returned. Note that for some reevaluations initial graph elements may be restored first by the unambiguous matching and update. Finally, when the reevaluated transformation is successfully reapplied ($newT \neq \emptyset$), it is replaced by the new transformation (line 20) and removed from the set of affected transformations (line 21).

In the case that the transformation has lost its applicability (line 22), all dependent transformations dependentT are collected (lines 23 to 26). The dependent transformations are also no longer applicable because their input is not revalidated. Subsequently, the failed transformation oldT and these dependent transformations dependentT are removed from the affected transformations (line 27) and added to the failed transformations failedT (line 28). The reevaluation finishes when all affected transformations were reevaluated (line 30). Instead of all transformations that are affected by the changes, only transformations that are not longer applicable are returned to the algorithm of Figure 5.11 (line 31).

To illustrate the History Graph Mechanism process Figure 5.16 to Figure 5.18 show the selective undo applied on the case study snapshot from Figure 5.9. Again we use the representation of Figure 5.8 for layout reasons.

Figure 5.16 shows the directly affected transformations (marked with ☞) and graph elements (marked with ◯) by the change of Variant 'Patient' (marked with ☝). Entity 'Patient' is not longer composed of the single Variant 'Patient' but of three Variants: 'Patient', 'Discharged' and 'Deceased', cf. Figure 3.8. Thus the Class 'Patient', which is the output of transformation with timestamp '1', no longer contains the same attributes prior to the change. Note that the directly affected transformations are the same as the affected transformations in the simple undo case. Therefore, Figure 5.13 and Figure 5.16 are identical.

Figure 5.17 shows all indirectly affected transformations(☞). The indirectly affected graph elements are marked in consequence (◯).

The marked transformations are then reevaluated in the predefined order of their input/output dependencies. Figure 5.18 shows the reevaluation. In this case, the first reevaluated transformation is either transformation '1' or transformation '2'. Transformation '3' depends only on transformation '1' and thus can precede transformation '2'. Transforma-

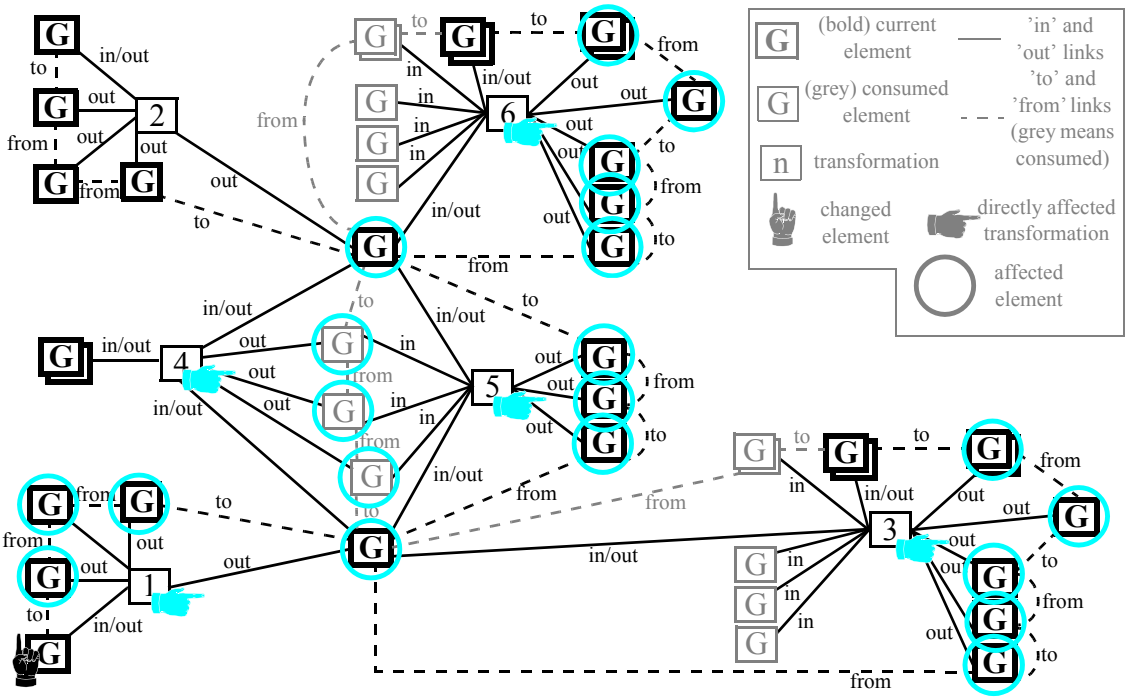


Figure 5.16: History Graph: directly affected transformations

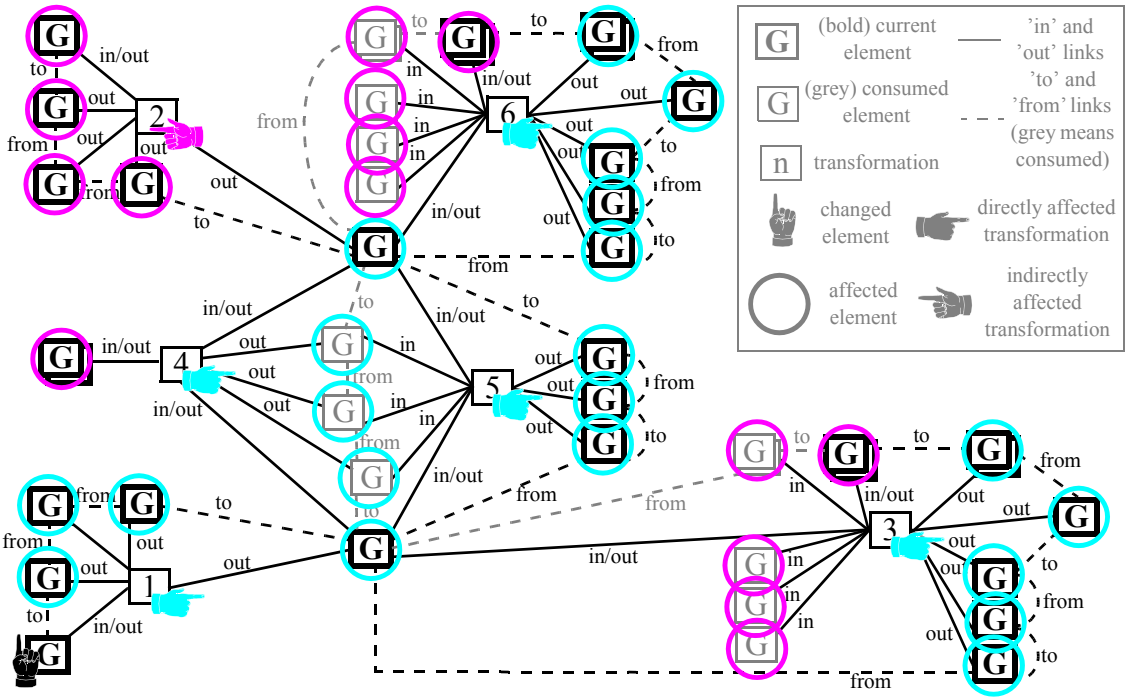


Figure 5.17: History Graph: indirectly affected transformations

tion ’3’ is successfully reevaluated because the involved attributes remained in Class ’Patient’. No other transformation is reevaluate prior these three transformations because all three remaining affected transformations depend on them. Next transformation ’4’ is reevaluated. Because of the changes (missing attributes) of the new Variant ’Patient’, the application of transformation ’4’ (createDuplication) fails. Subsequently transformation ’5’ (createAssociation) is no longer applicable because it depends on transformation ’4’. This is shown by the symbol. Transformations ’6’ is not removed from the affected transformation set although it depends on transformation ’4’. The reason is that the linking graph element is also input of transformation ’4’ and thus a reevaluation can be successful. Indeed, in our case splitClass (transformation ’6’) is still applicable. Transformation ’6’ is successfully reevaluated because Class ’Deceased’ did not changed. Finally, all transformations that are no longer applicable and exclusively their output graph elements are deleted (marked with).

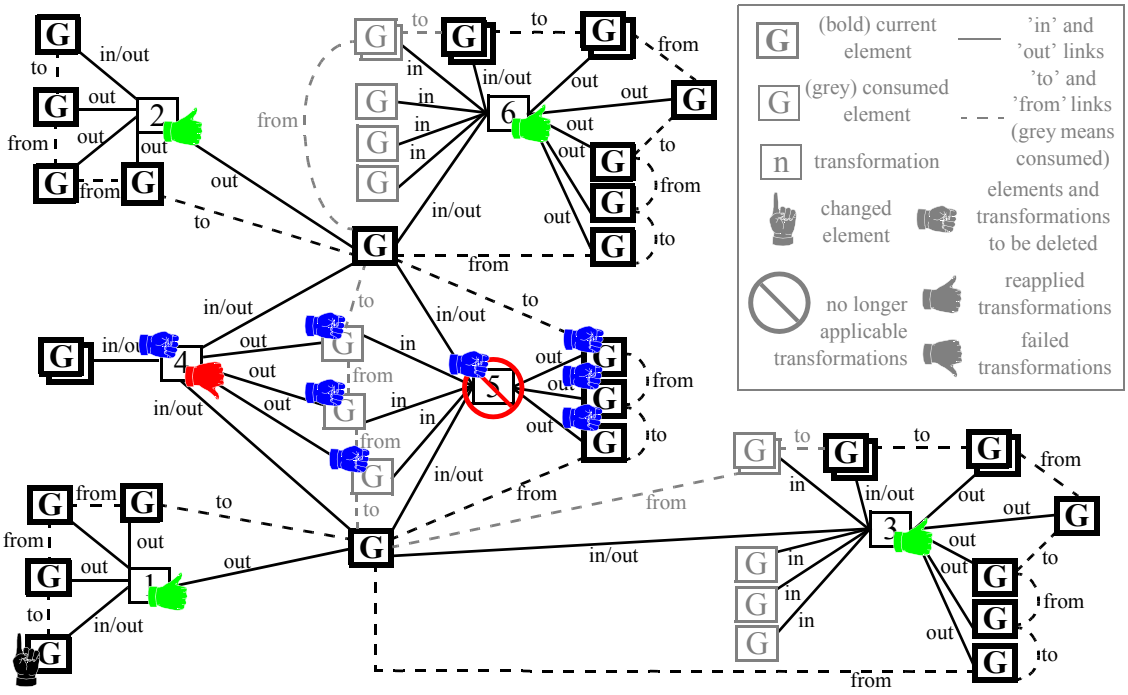


Figure 5.18: History Graph: reevaluated transformations

In a first time, the impact seems to be even more massive then for the simple undo case. Again, considering the whole palliative care information system only a small part is affected. The advantages of reevaluating the transformations are obvious comparing Figure 5.14 and Figure 5.18. In case of the simple undo all transformations are undone except transformation ’2’ whereas in the selective undo case only transformations ’3’ and ’5’ were undone.

5.2.4 Composed History Graph Mechanism

So far the History Graph Mechanism is based on one single graph which traces all the transformations. To further sustain iterative and explorative processes we introduce the *composed* History Graph Mechanism. We get following benefits from the composition of History Graphs:

- branches
  Try-outs during iteration are possible, i.e., from a common start graph several branches can be processed in parallel. To get the best possible result the same task can be performed by two distinct reengineers and later be compared. The merging of graphs resulting from different branches is not addressed for the moment.
- structured process
  Each process activity can be attributed an own History Graph (as far as it is possible). Thus, a History Graph is self contained for each process activity. This enables parallel work since the transformations are logged in different History Graphs, e.g., working in parallel on distinct graphs that are only united later. Working in parallel on graphs that depend on each other is only meaningful if the impact of changes is limited.
- impact reduction
  A consequence of several smaller History Graphs is that the number of transformations to be reevaluated is reduced. The reason is that inside a History Graph reevaluation no check is made if the output graph elements are „really“ affected. This could be achieved by investigating if a graph element was changed by the reevaluated transformation. This investigation automatically takes place for some transformations during the composition of History Graphs.
- prevention of scaling problems
  We have shown that our approach is scalable [JSWZ02]. Nevertheless, scaling problems may occur if a single History Graph stores all transformations for long-lasting processes. The division into several smaller History Graphs prevents such scaling problems.

Three composition kinds of History Graphs are provided by the composed History Graph Mechanism, i.e., *sequence*, *union* and *branch*. Figure 5.19 shows these kinds in the figure compartments B to D. Compartment A depicts a History Graph chronology: the start graph is transformed into the current graph; both are included in the History Graph (complete graph).

A sequence of History graphs is shown in compartment B. In the History Graph I, graph 1 is transformed into graph 2. In History Graph II, graph 2 is transformed into graph 3. The sequence is realised by copying the current graph of History Graph I (graph 2) and using it as start graph of History Graph II.

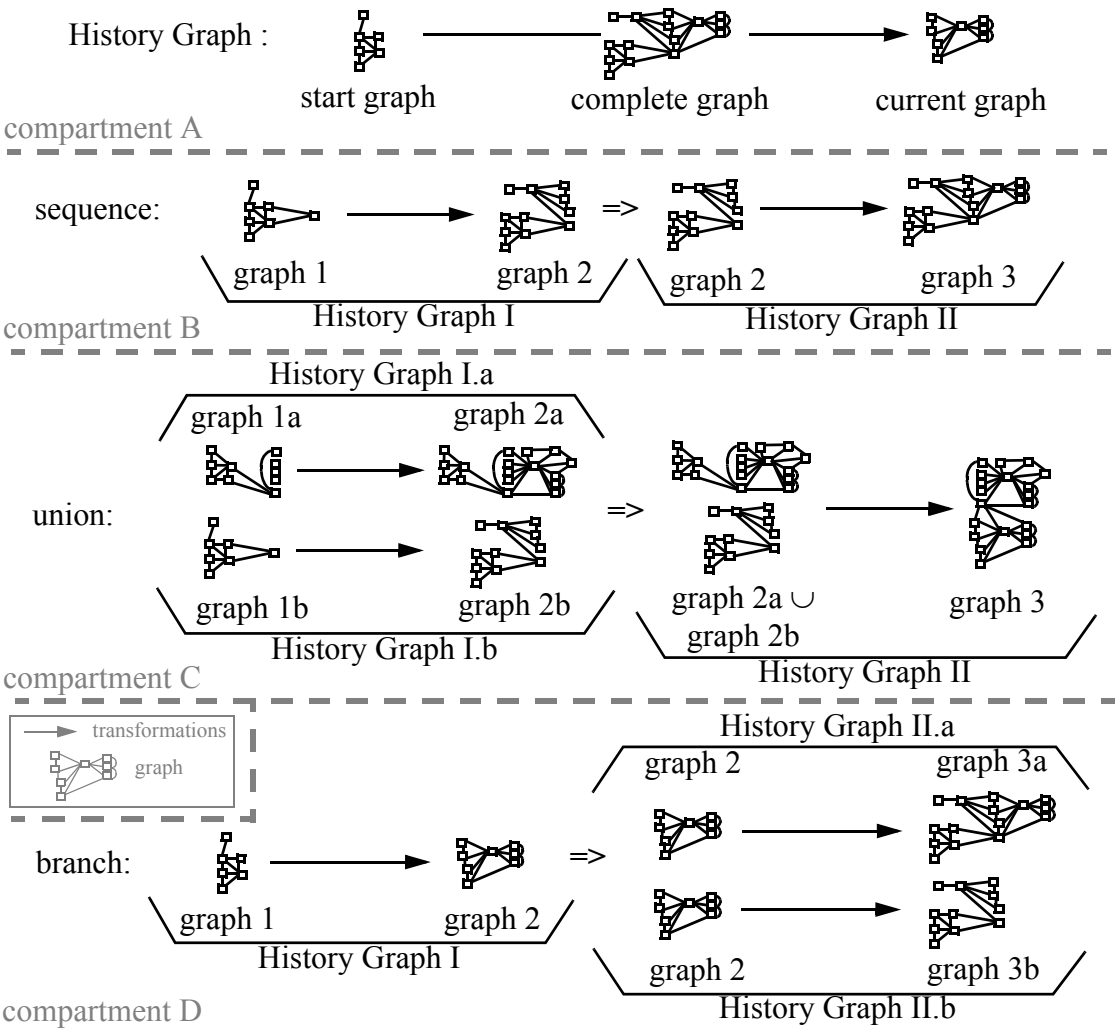


Figure 5.19: Composed History Graph: overview

In analogy, compartment C shows the union of History Graphs. The current graphs (graph 2a and graph 2b) of two or more parallel History Graphs (History Graph I.a and History Graph I.b) are united to one graph. This graph (graph 2a ∪ graph 2b) is the start graph of the subsequent History Graph (History Graph II).

Finally, branching of History Graphs is achieved by duplicating a current graph (graph 2 of History Graph I) and introducing it as start graph in two or more subsequent History Graphs (History Graph II.a and History Graph II.b), cf. compartment D.

The reevaluation in composed History Graphs works inside each History Graph like the reevaluation in single History Graphs. Only the transition, i.e., the correspondence of current and start graph, from one History Graph to another has to be determined. Figure 5.20

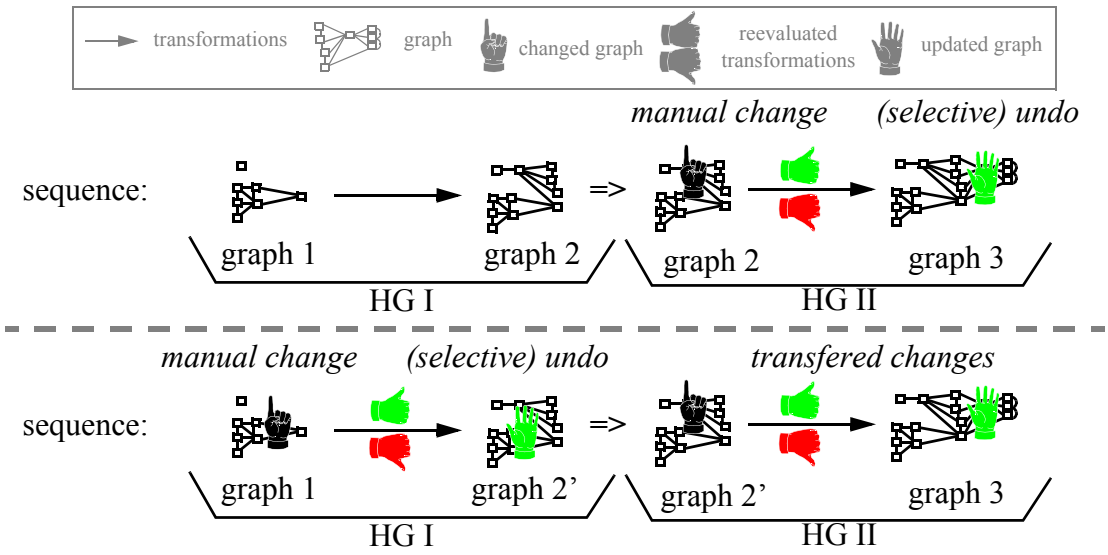


Figure 5.20: Composed History Graph: reevaluation

shows the reevaluation in case of the sequence of History Graphs. Two situations can occur. Firstly, the manual change takes place on graph 2. Then a (selective) undo is done inside the HG II and HG I is not affected. Secondly, the manual change takes place in graph 1. After reevaluation the graph 2 is updated to graph 2'. To transfer the changes, graph 2' is passed to the HG II and becomes the new start graph of HG II. This is done by calling addChanges(graph 2 , graph 2'); graph 2 exist as copy in HG II and graph 2' is the updated current graph of HG I. In term of changes:

changes in start graph (HG II) =
diff (old startGraph(HG II), updated currentGraph(HG I))

For the two remaining composition kinds, we only present the changes:

- union:
  - changes in start graph (HG II) =
    diff (old startGraph(HGII), updated currentGraph(HGI.a ∪ HGI.b))
- branch:
  - changes in start graph (HG II.a) =
    diff (old startGraph(HGII.a), updated currentGraph(HGI))
  - changes in start graph (HG II.b) =
    diff (old startGraph(HGII.b), updated currentGraph(HGI))

Figure 5.21 shows an example of History Graph composition in the context of our case study. The reengineering process starts with the three databases hospdata, bvmtdata and outcomes_be. To trace and maintain all applied transformations, fourteen history graphs are involved.

Each database is parsed and represented as an initial physical schema. In Figure 5.21 we only quote the qualifier of the schemas, e.g., physical schema is quoted with 'physical'. Each physical schema is retrieved into a relational schema (HG I.a, HG I.b and HG I.c). Then each relational schema is mapped to a conceptual schema (HG II.a, HG II.b and HG II.c) that is refactored (HG III.a, HG III.b and HG III.c). Those transformations can take place in parallel and thus three initial physical schemas are transformed in three conceptual refactored schemas.

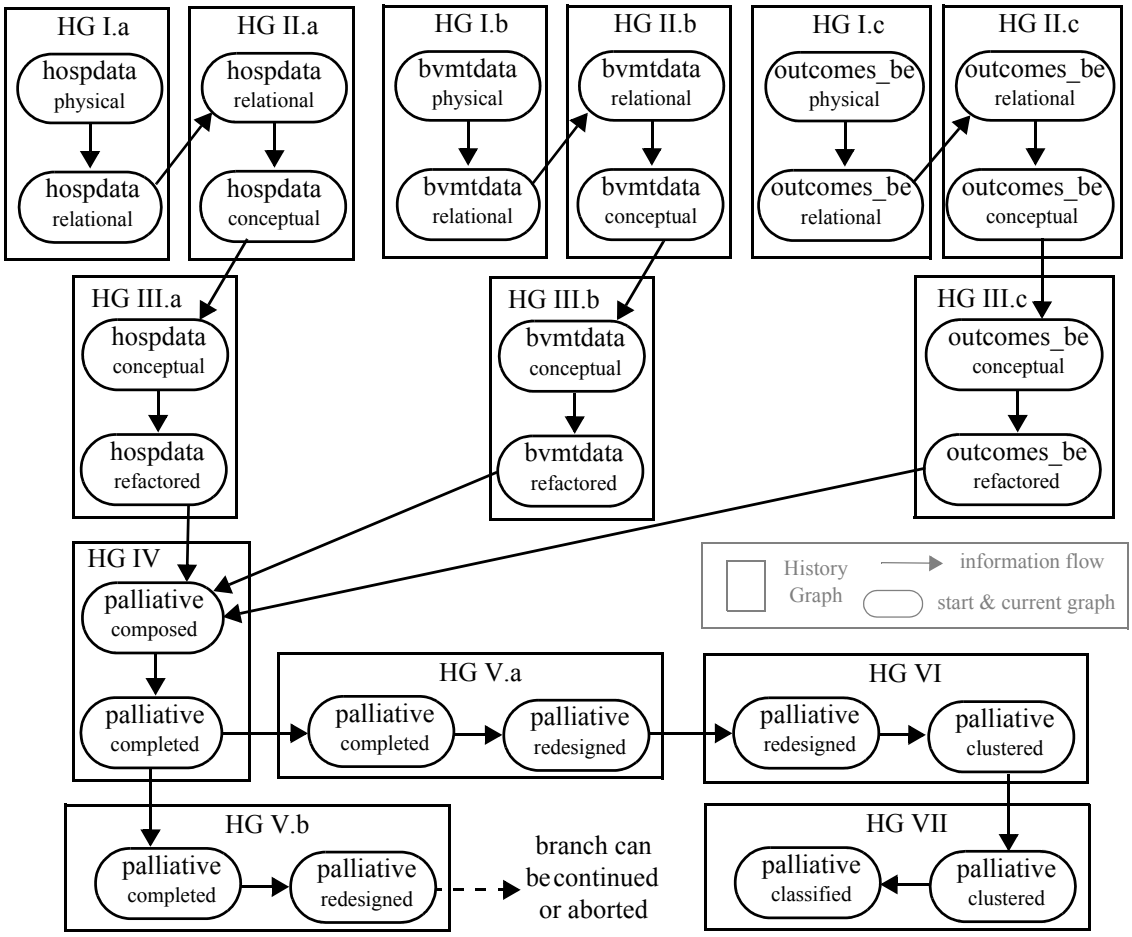


Figure 5.21: History Graph Sequence Example

The three refactored schemas are united into one composed schema. Since these refactored schemas are disjoint, i.e., they do not have any inter-schema relationships this can easily be done. The inter-schema relationships are investigated next. The composed schema is transformed into a completed schema (HG IV). At this point a single user or more than one user may try out some redesign transformations on the same completed schema and decide later which branch will persist. In the scenario of Figure 5.21 two branches are created. The corresponding (redesign) transformations are logged in HG V.a and HG V.b,

respectively. We assume that the branch logged by HG V.a is chosen. HG VI maintains the transformation related to clustering. Finally, the classification operations are retained in HG VII.

One benefit of the composed History Graph Mechanism is the impact reduction. This is shown in Figure 5.22 to Figure 5.24 where we replay the reevaluation of Section 5.2.3.

We start with the changed Variant 'Patient' in HG II.a, cf. Figure 5.22. The only affected transformation is transformation '1' because all other transformations, i.e., '2' to '6' are located in other History Graphs. Transformation 'MapVariantToConcreteClass' (transformation '1') is successfully reapplied.

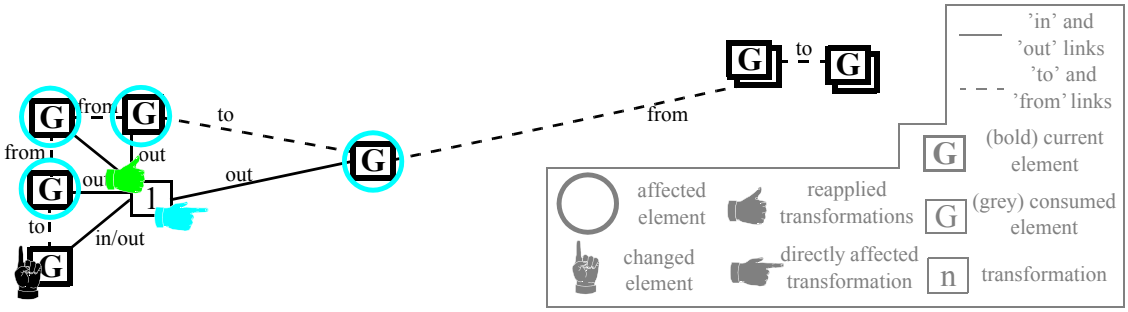


Figure 5.22: Reevaluation of HG II.a

During refactoring, Class 'Patient' was split into Class 'Patient' and Class 'Address'. Since transformation '1' was reapplied, changes occurred, but Class 'Patient' itself was not changed. Thus the transfer of changes from HG II.a to HG III.a do not affect any input graph element of transformation '3', i.e., splitClass, cf. Figure 5.23.

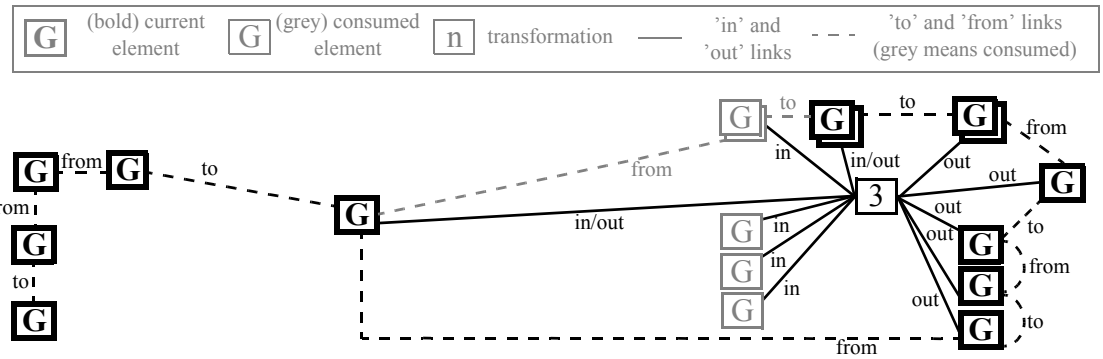


Figure 5.23: Reevaluation of HG III.a

Even if in HG III.a no transformation was affected, the transfer of changes is pursued. Indeed, the updated graph elements of HG II.a have an impact on HG IV. The mapping of the changed Variant 'Patient' engender changes in the Preconditions of transformation '4', cf. Figure 5.24. Like in Figure 5.18, the application of transformation '4' (createDuplica-

tion) fails and transformation ’5’ (createAssociation) is no longer applicable. Thus the concerned graph elements are deleted.

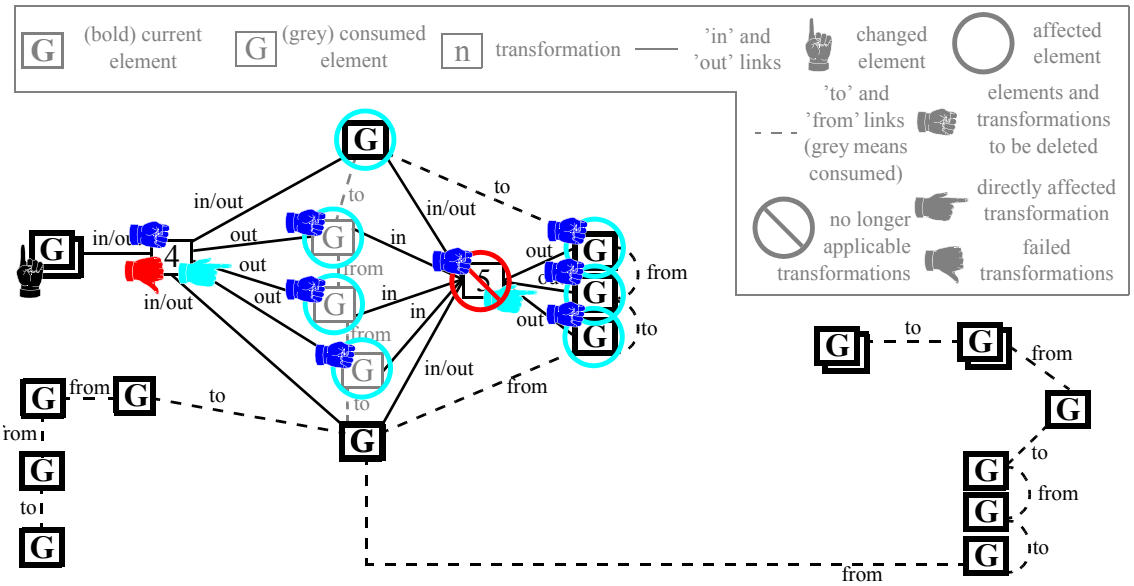


Figure 5.24: Reevaluation of HG IV

Class ’Deceased’ is affected, but not changed. In analogy with HG III.a, no impact on HG V.a and HG V.b is transferred. Note that the changes still have to be transferred further. In our case the deletion of the duplication Association ’deathdate’ affects the clustering and consequently HG VI. We refer to Section 4.2.1 for details of the clustering.

Comparing the reevaluation between the single History Graph Mechanism and the composed History Graph Mechanism in our example, only 2 instead of 5 transformations are reevaluated. The reason therefore is the duplication of graph elements in the different History Graphs. On the one hand, a check if the graph element has really changed is done by transferring the changes to the subsequent History Graph. On the other hand, graph elements do not have to be restored because they exist in their current state in the corresponding History Graph.

5.3 Tool Support

The GXL-based History Graph Mechanism tool support is joint work with the netlab Group located at the University of Victoria, Canada. In our context it is located in the REDDMOM project. REDDMOM is mainly based on the FUJABATS and especially on parts of the FUJABARE. The History Graph Mechanism is independent from the tool support for the undestading phase and the adaption phase. It is accessed via an API. Figure 5.25 shows an overview of the History Graph Mechanism tool support for model consistency management. The filled tool parts are currently under development.

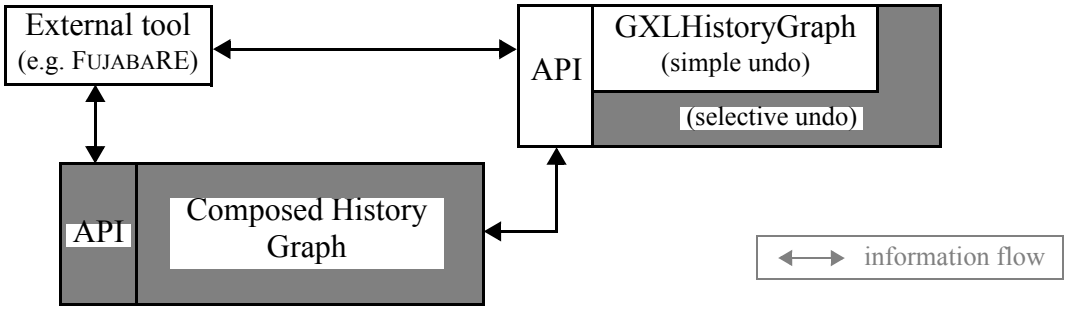


Figure 5.25: Architecture of the History Graph Mechanism tool support

External tool (prerequisites):

A tool has to meet certain requirements to be able to interface with our History Graph Mechanism. We have minimised these requirements as much as possible to enable the integration with many different environments. Basically, a tool has to be able to

- import and export its internal abstract syntax graph structure (preferably in GXL),
- report for each transformation on the abstract syntax graph
  - the graph elements that represent the input of the transformation (in GXL)
  - the graph elements that represent the output of the transformation (in GXL), and
- provide an API that enables the reevaluation of a transformation specified by its name and with given input (in GXL).

GXLHISTORYGRAPH:

The API of the GXLHISTORYGRAPH comprises following operations:

- addGraph(graph)
  History Graph initialisation, makes a graph copy.
- addTransformation(leftGraph_i, rightGraph_i, name_i)
  build the History Graph, returns transformation 'trafo_i'.
- createHistory()
  returns the list of all applied transfomations {trafo_1, ..., trafo_n}.
- getGraph(trafo_j)
  calculates the valid graph before the application of transformation trafo_j.
- addChanges(graph1, graph2)
  transfers the changes between graph1 and graph2 into the History Graph.
- propagateChange()
  determines the impact of the changes and returns a „change-consistent" graph.

All graphs passed to GXLHISTORYGRAPH must be in GXL format.

The simple undo determine the affected transformations and undo them. This is done by deleting the affected transformations and the corresponding graph elements

The selective undo only undo the transformations that cannot be successfully reevaluated.

Composed History Graph:

The API of the Composed History Graph comprises following operations:

- createHistoryGraph()
  creates a new History Graph and returns it. All History Graph (in our case GXLHistoryGraph) API operations are available.
- createHistory()
  returns the list of all History Graphs {HG I, ..., HG N}.
- getHistoryGraph(HG x)
  returns the History Graph HG x.
- propagateChange()
  transfers the changes to all History Graphs.

All History Graph API operations are passed to the corresponding GXLHISTORYGRAPH operations. The GXLHISTORYGRAPH can be replaced by other History Graph implementations. If an external tool uses a format different from GXL, two solutions are possible. Firstly, a new History Graph that understands this format has to be implemented. Secondly, a converter from this format into GXL and vice versa is needed.

5.4 Related work

The described research is related to various existing approaches maintaining inter-document consistency in the software (forward) engineering process. Perhaps the most prominent technology in this regard are mechanisms for version and configuration management like, e.g., the concurrent versions system [CVS]. In principle, mechanisms like the concurrent versions system can be used to create version histories of recovered design documents. However, such version histories do not consider the actual structural dependencies among the transformations performed in the abstraction process. Instead, the version history merely reflects a time-oriented view on intermediate stages reached in the (last iteration of the) recovery process. Consequently, traditional version management systems can only provide limited support for iteration.

Lefering and Schürr suggest a dedicated formalism called Triple Graph Grammars for specifying structure dependencies among abstract syntax graphs [SL96]. This formalism is used in the IPSEN project for generating integration tools that maintain consistency and propagate changes from software design documents to the implementation code and vice

versa [Nag96]. Nowadays, most commercially available development environments provide (limited) support for change propagating between software documents on different levels of abstraction.

Unfortunately, very few reengineering tools provide similar support for propagating changes of the knowledge about a legacy system. In most current tools, the legacy system analysis and information extraction, the information transformation and abstraction, and the system and information extension (integration) cannot be iterated without losing the interactive work performed. They impose a strictly phase-oriented, waterfall-like reengineering process, without the support for iteration. This is an important limitation in practice, as iterations between reverse engineering and integration steps occur frequently: when a reengineer learns more about the abstract design of a legacy system, (s)he often refutes some initial assumptions and does further investigations. Moreover, reengineering projects might have durations from several months up to years. Urgent (on-the-fly) modifications of the original system during this period have to be reflected in the target system, which, of course, leads to the demand for iteration in the reengineering process. This problem of consistency management in process iteration is only addressed in DB-MAIN [HHE+99] and VARLET [Jah99].

DB-MAIN logs the history of all transformations during the reengineer's interaction. Each performed transformation is recorded precisely and completely to make its inversion possible. The history has to be monotone and linear. This makes it possible that changes can be propagated from one schema level to another. In case that a transformation recorded in the history becomes invalid the reengineer has to resolve the inconsistencies manually. DB-MAIN does not allow the reengineer to get back (and modify) the initial state of the legacy system. Exploration (branches) and concurrent work are not supported.

The VARLET environment supports iterative data analysis and abstraction processes [JW99]. VARLET logs all schema transformations invoked by the user in an automated logbook (including their pre- and post-conditions and interdependencies). Moreover, the initial, low-level representation of the legacy schema remains available to the user. Whenever the initial representation of the legacy system is changed, VARLET uses the logged transformation dependencies to determine which interactive operations are affected by the modifications. The pre-conditions of these transformations are then reevaluated and only those transformations that fail this test are undone. The consistency of the extracted information on the legacy system and its interactively created abstraction has been re-established. This consistency mechanism is encoded in VARLET and thus cannot be reused by other tools.

Two related approaches that address change propagation are the *Evolvable View Environment* [NR99] and *ArchDiff* [vv02]. Nica and Rudensteiner [NR99] present an approach that updates view extents after view synchronisation driven by schema changes. Similar to our approach, they exploit the knowledge: how the view definition was synchronised

and which changes were performed on the view definition after synchronisation. Van der Westhuizen and van der Hoek [vv02] present algorithms that can be used for understanding architectural changes and propagating those changes among individual architectures in the product line. The strength of the algorithms lies in their use of a simple, XML-based representation for capturing architectural changes. Different algorithms build upon this representation to determine not only those architectural elements that have been added or removed, but also those sets of elements that represent replacements within the architecture.

5.5 Summary and Future Work

This chapter started with an overview of our model consistency management. The presented History Graph Mechanism enables to restore consistency between models that represent web information systems. We provided the background information for our graph-based approach. Next we explained how model changes are propagated and discussed successively simple undo and selective undo to restore consistency. Finally, we presented how History graphs can be combined and the consequent tool support.

We allow composition and duplication (results in branches) of history graphs. This provides flexibility and sustains iterations, e.g. a parallel try in multiple branches followed by choosing one branch. The choosing can be done after change propagation to all branches. We recommend that all branches are erased and are not longer used to avoid confusions and runtime overhead. The composition can be refined by passing subgraphs, but this requires an exact definition of filter (user view). Future improvements would be the splitting and merging of graphs. This would provide more flexibility, e.g., by merging branches. Therefore transformations covering several graphs may be needed, what makes a consistency management hard. Moreover, possibilities to suppress, add or replace a transformation would increase the usability and support of iteration.

CHAPTER 6: CONCLUSIONS

Fundamental progress has to do with the reinterpretation of basic ideas.

-- ALFRED NORTH WHITEHEAD
British mathematician, logician and philosopher (1861-1947)

6.1 Summary

The costs spent for reengineering are no garantee for successful reengineering. In [BSS+99] ten reasons that reengineering efforts fail are presented. Seven reasons tackle project management and technical issues. The remaining three reasons are matched by our data-oriented reengineering process:

- „The organization does not have its legacy system under control.“ (reason 4)

  We provide understanding of a web information system by data-oriented reverse engineering and use clustering to partition the web information system.
- „Software architecture is not a primary reengineering consideration.“ (reason 6)

  We consider relationships between data components and proposed architectural patterns to (re)restructure the web information system.
- „There is no notion of a separate and distinct „reengineering process“. “ (reason 7)

  We propose our data-oriented reengineering process that includes the management of the models representing the web information system.

Our data-oriented process is composed of two phases: the understanding phase and the adapting phase. Figure 6.1 shows the flexible tool support, which sustains iterations and explorations, that we provide for these two phases. Basic information about the data is regained by parsing the web information system and represented in Data Component Models. These Data Component Models are interactively completed, abstracted and transformed by the reengineer. The web information system can then be extended with further models. From the regained and new models an access layers and new applications are generated.

One of the major contributions of this dissertation is the classification and retrieval of relationships in web information systems. We combined existing approaches and tools to recover the data structure of a web information system particularly the data dependencies between the data components. During this reverse engineering process uncertainty is expressed with fuzzy logic and inconsistencies are tolerated.

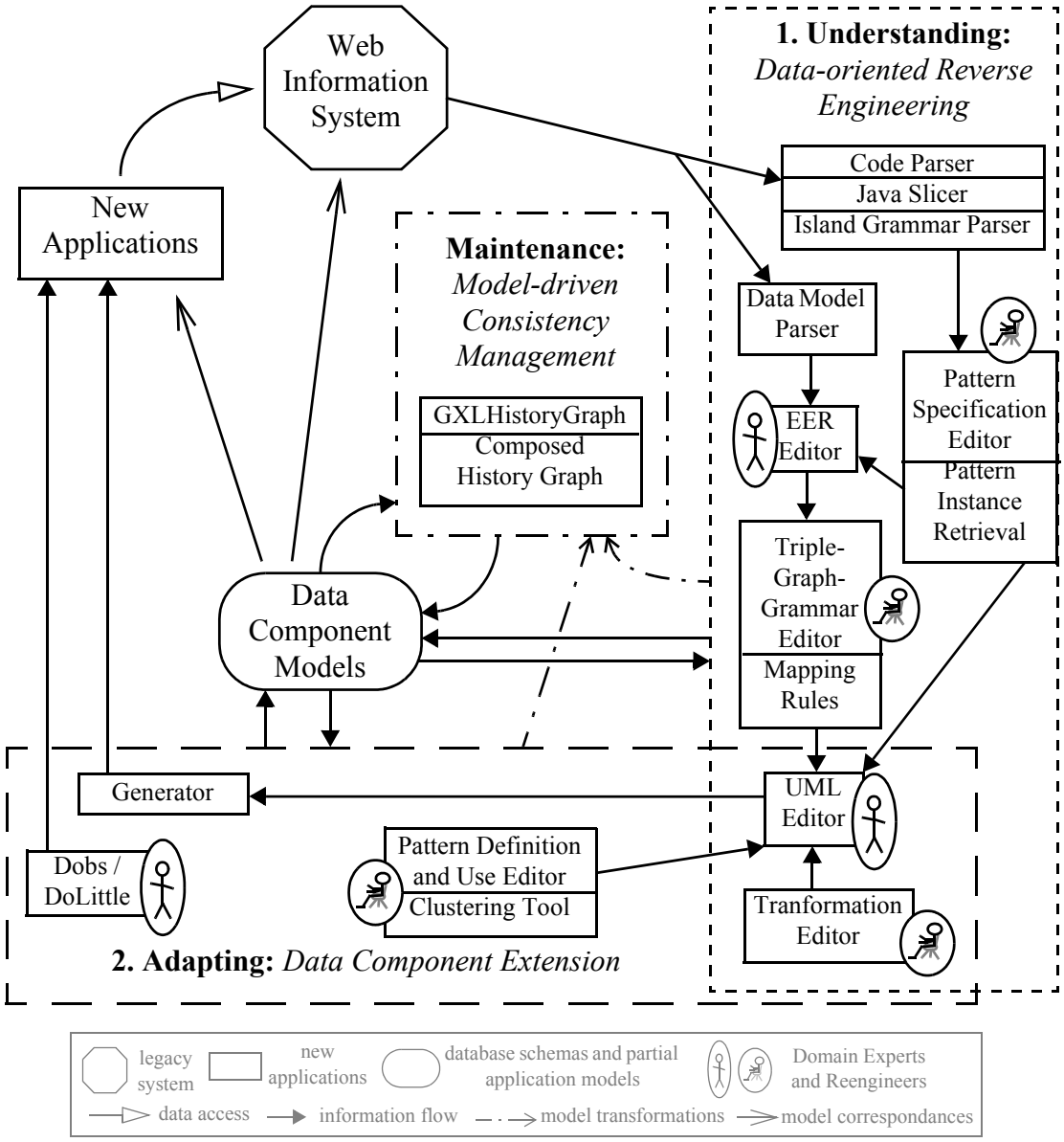


Figure 6.1 Tool Support for the Data-oriented Reengineering process

Once models of the data components are reverse engineered, these retrieved models are redesigned and new functionality, i.e., schemas and/or applications, is modeled. To this end, we provide clustering of the Data Component Models followed by classification of the data components. We presented clustering strategies depending on the chosen integration strategy. To (re)structure the data components we elaborate four architectural patterns.

After adapting and extending the web information, we provide code generation facilities. Firstly, based on the maintained dependencies between the physical schemas and conceptual, object-oriented schemas, we generated transactional object-oriented access layers for the legacy databases. Secondly, for newly modeled data components we generated Java code that accesses the legacy system via the transactional access layer. The aim is to enable the construction of data component prototypes by model execution.

We maintain the knowledge of the system by model consistency management. We make use of graph transformations for the model-based reverse engineering and the model-driven development. We developed an incremental graph-based model change propagation mechanism, i.e., the History Graph Mechanism, which preserve consistency between models. This mechanism enables the selective undo of (graph) transformations that become invalid because of changes performed after the transformations' application.

Finally, we evaluated our data-oriented reengineering process and tool support with a case study for proof of concepts. This case study deals with the reengineering of a web information system in the health care domain.

6.2 Transferability of Results

Most of our results are not limited to the reverse engineering of legacy web information systems like presented in the case study, i.e., composed of databases and object-oriented application code. Our data-oriented reverse engineering process can be applied to other data sources, e.g., file based, and non object-oriented application code, e.g., COBOL.

Similar challenges exist in architectural reverse engineering. As our approach is based on a combination of data reverse engineering and design recovery, especially design pattern instances recovery, it can be extended to architectural aspects: (1) either the extension can be done by adding or exchanging tools for recover architectural models and dependencies; or (2) the existing parsers and pattern definitions can be adapted or extended.

The relationship retrieval is based on a data related pattern instance recovery approach. Business rules also depend on the information systems' data. Defining business rules with patterns anchored on the systems' data structure enable their detection with our approach.

The generation of the transactional object-oriented access layer and the model consistency management enables incremental web information system migration [BS95]. The access layer builds an interface between the data and the applications. First, the applications' data access is shifted to exclusive data access via the generated access layer. Then, the physical schemas can be changed or migrated in a controlled way and with limited impact into the generated access layer. Last, data transfer can be automated due to the explicit data component model maintenance. The model consistency management enables a long-lasting maintenance strategy by keeping the models consistent during parallel system evolution.

Different engineering tools have different application domains (e.g., software product line development, architectural recovery or program understanding) but the vast majority of

tools have in common that they are graph-based, i.e., they store models in abstract syntax graphs. Most of the engineering processes supported by these tools are iterative. Our History Graph Mechanism can propagate changes occurring during iteration through any graph structure. Therefore, every graph-based tool providing the simple History Graph Mechanism API can benefit from incremental change propagation.

6.3 Open Problems

The case study revealed open problems that need further investigation. The definitions of the usage and constraint relationship should be subdivided to enable more detailed analysis and clustering. The kinds of data dependencies used, e.g., replication or duplication, would provide additional helpful knowledge to the reengineer. Another useful information regarding the extension would be how the data dependency is used, e.g., encapsulated in a method call or directly by an embedded SQL statement.

A related open problem is the clustering. First results during evaluation have demonstrated the clustering usefulness. The existing clustering approaches and tools are generally focused on source code re-modularisation or schema abstraction. A deeper investigation of data component clustering focusing on data distribution is desirable to reach more accurate and reliable results.

Our approach is limited to the generation of transactional object-oriented access layers to existing databases and of prototypes. Many practical scenarios exist that require platform-specific application design and thus specific generation. A major improvement would be the code generation for middleware technologies. Common solution for information system integration middleware within organisations is distributed transaction processing [XA94] as provided by transaction monitors [Hud94, Hal96] and middleware transaction services [OTS98, JTS99]. A more scalable solution is reliable messaging [Hou98, Lew99] which results in a reliable asynchronous processing scenario. Liebig and Tai propose an integration of message-oriented transactions and distributed object transaction to middleware mediated transactions [LT01].

Existing tools can be integrated by XML file exchange. Nevertheless, tighter tool integration, e.g., at the meta-model level like realised in the FUJABATS [BGN+03], is desirable. Further, layout information is often lost during iteration and information transfer between tools.

6.4 Future Directions

More possible information sources for the data-oriented reverse engineering process can be found, like sketched in [HHH+00]. One valuable source of information would be to record the user interactions on the system to find out and increase certainty of data dependencies. Depending on the existent application the databases are queried in various forms. Approaches that traces user interactions for system understanding are presented in

[MC01, ESS02]. We implemented a proprietary approach that propose optimisation measures depending on the kind of query and the duration a query needs in the different system's tiers [Rot03, Wad03]. This result can be transferred and used for web information system understanding.

The retrieval can be further improved by adding and combining reverse engineering techniques to our approach [MBPRR01]. In [DRT98] three reverse engineering patterns are presented: (1) code duplication detection, (2) architectural extraction using prototyping and (3) inferring hot spots (a variation in the application domain) from overridden methods. The code duplication detection can be used to weight the code fragments. Since we provide prototyping facilities, a further iteration can take place applying this pattern. Inferring hot pots can help to recover hidden dependencies. „A hidden dependency is a relationship between two seemingly independent components and it is caused by a data flow inside a third software component.“ -- [YR01] Further the automatic extraction of interfaces like presented in [WML02] would help the reengineer when (re)structuring the web information system.

The current information technology trends require the extension of legacy web information systems in two directions. Firstly, integrating mobile devices is needed to meet the ubiquitous computing requirements. Mascolo et al. present a data-sharing middleware for mobile computing named XMIDDLE [MCZE02]. The sharing of XML documents across heterogeneous mobile hosts is provided by XMIDDLE, allowing on-line and offline access to data. Replication transparency is abandoned by XMIDDLE to achieve an acceptable performance and scalability. Secondly, a seamless integration to the internet is still a challenge. Many approaches exist in this direction, e.g., [MG00, KK01, MMK02, SL02].

REFERENCES

Chapter 1: Introduction

[Aik96] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1996.

[Bas90] V.R. Basili. *Viewing Maintainance as Reuse-Oriented Software Development*. IEEE Software, 7(1):19–25, September 1990. IEEE Computer Society Press.

[BGN+03] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*. In Proceedings of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), pages 51–56, September 2003. online at http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.p%df.

[BH99] I.T. Bowman and R.C. Holt. *Reconstructing Ownership Architectures To Help Understand Software Systems*. In Proceedings of the 7th International Workshop on Program Comprehension, Pittsburgh, USA, pages 28–37. IEEE Computer Society Press, May 1999.

[CC90] Elliot J. Chikofsky and James H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, 7(1):13–17, January 1990. IEEE Computer Society Press.

[EH99] V. Englebert and J.-L. Hainaut. *DB-MAIN: A Next Generation Meta-CASE*. Journal of Information Systems - Special Issue on Meta-CASEs, 24(2):99–112, June 1999. Elsevier Science Publishers B.V (North-Holland).

[GAK99] G.Y. Guo, J.M. Atlee, and R. Kazman. *A Software Architecture Reconstruction Method*. In Proceedings of the 1st Working IFIP Conference on Software Architecture, San Antonio, USA, pages 22–24. Kluwer Academic Publishers, February 1999.

[GCBM96] W.G. Grisworld, M.I. Chen, R.W. Bowdidge, and J.D. Margenthaler. *Tool Support for Planning the Restructuring of Data Abstractions in Large Systems*. In Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, USA, pages 33–45. ACM Press, October 1996.

[Gro01] Object Management Group. *Model Driven Architecture (MDA) Edited by Joaquin Miller and Jishnu Mukerji*. online at http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 492 Old Connecticut Path, Framingham, MA 01701, USA, 2001.

[GW01] H. Giese and J.P. Wadsack. *Reengineering for Evolution of Distributed Information Systems*. In Proceedings of the 3rd International Workshop on Net-Centric Computing: Migrating to the Web, Toronto, Canada, pages 36–39. ACM Press, May 2001.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JGW02] J.H. Jahnke, D.M. German, and J.P. Wadsack. *Architecturals Patterns for Data Mediation in Web-centric Information Systems*. In Proceedings of of the 3rd ICSE Workshop on Web Engineering, Orlando, USA, May 2002.

[JNW00] J.H. Jahnke, J. Niere, and J.P. Wadsack. *Automated Quality Analysis of Component Software for Embedded Systems*. In Proceedings of the 8th International Workshop on Program Comprehension, Limerick, Irland, pages 18–26. IEEE Computer Society Press, June 2000.

[JSWZ02] J.H. Jahnke, W. Schäfer, J.P. Wadsack, and A. Zündorf. *Supporting Iterations in Exploratory Database Reengineering Processes*. Journal of Science of Computer Programming, 45(2-3):99–136, November 2002. Elsevier Science Publishers B.V (North-Holland), (Special Issue on Software Maintenance and Reengineering).

[JW99a] J.H. Jahnke and J.P. Wadsack. *Human-centered Reverse Engineering Environments should Support Human Reasoning*. In Proceedings of the 1st Workshop on Soft Computing Applied to Software Engineering, Limerick, Ireland, pages 77–84. Limerick University Press, April 1999.

[JW99b] J.H. Jahnke and J.P. Wadsack. *Integration of analysis and redesign activities in information system reengineering*. In P. Nesi and C. Vernoef, editors, Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, Amsterdam, The Nederlands, pages 160–168. IEEE Computer Society Press, March 1999.

[JW99c] J.H. Jahnke and J.P. Wadsack. *Varlet: Human-Centered Tool Support for Database Reengineering*. In J. Ebert, B. Kullbach, and F. Lehner, editors, Proceedings 1st of the Workshop Software Reengineering. Bad Honnef, Germany, pages 149–156. Fachberichte Informatik, Universität Koblenz-Landau, July 1999.

[JW00a] J.H. Jahnke and J.P. Wadsack. *The Varlet Analyst: Employing Imperfect Knowledge in Database Reverse Engineering Tools*. In In Proceedings of the 3rd International Workshop on Intelligent Software Engineering, Limerick, Ireland, pages 59–69. IEEE Computer Society Press, June 2000.

[JW00b] J.H. Jahnke and A. Walenstein. *Reverse Engineering Tools as Media for Imperfect Knowledge*. In Proceedings of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, pages 22–31. IEEE Computer Society Press, November 2000.

[JWZ02] J.H. Jahnke, J.P. Wadsack, and A. Zündorf. *A History Concept for Design Recovery Tools*. In Proceedings of the 4th European Conference on Software Maintenance and Reengineering, Budapest, Hungary, pages 59–69. IEEE Computer Society Press, March 2002.

[KSRL01] R.K. Keller, R. Schauer, S. Robitaille, and B. Lagu. *The SPOOL Approach to Pattern-Based Recovery of Design Components*. In H. Erdogmus and O. Tanir, editors, Advances in Software Engineering Topics in Evolution, Comprehension, and Evaluation. Springer Verlag, 2001.

[Mil98] H.W. Miller. *Reengineering Legacy Software Systems*. Digital Press, 1998.

[MTW93] H. Müller, S. Tilley, and K. Wong. *Understanding Software Systems using Reverse Engineering Technology: Perspectives from the Rigi Project*. In Proceedings of the 1993 IBM/NRC CAS Conference, Toronto, Canada, pages 217–226. IBM, October 1993.

[Nie03] J. Niere. *Inkremetelle Mustererkennung*. PhD thesis, University of Paderborn, Paderborn, Germany, December 2003.

[NNWZ00] U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. *Roundtrip Engineering with FUJABA*. In J. Ebert, B. Kullbach, and F. Lehner, editors, Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany, pages 31–34. Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. *Towards Pattern-Based Design Recovery*. In Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, pages 338–348. ACM Press, May 2002.

[NWW01] J. Niere, J.P. Wadsack, and L. Wendehals. *Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic*. Technical Report tr-ri-01-222, University of Paderborn, Paderborn, Germany, March 2001.

[NWW03] J. Niere, J.P. Wadsack, and L. Wendehals. *Handling Large Search Space in Pattern-Based Reverse Engineering*. In Proceedings of the 11th International Workshop on Program Comprehension, Portland, USA, pages 274–280. IEEE Computer Society Press, May 2003.

[NWZ01] J. Niere, J.P. Wadsack, and A. Zündorf. *Recovering UML Diagrams from Java Code using Patterns*. In J.H. Jahnke and C. Ryan, editors, Proceedings of the 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands. Centre for Telematics and Information Technology, University of Twende, The Netherlands, February 2001. online at http://trese.cs.utwente.nl/scase/scase-2/Proceedings.pdf.

[OLHL02] A. Orso, D. Liang, M.J. Harrold, and R. Lipton. *Gamma System: Continuous Evolution of Software after Deployment*. In Proceedings of the 2002 International Symposium on Software Testing and Analysis, pages 65–69. ACM Press, July 2002.

[ORH02] A. Orso, A. Rao, and M. J. Harrold. *A Technique for Dynamic Updating of Java Software*. In Proceedings of the 2002 IEEE International Conference on Software Maintenance, Montréal, Canada, pages 649–658. IEEE Computer Society Press, October 2002.

[PG02] M. Pinzger and H. Gall. *Pattern-Supported Architecture Recovery*. In Proceedings of the 10th International Workshop on Program Comprehension, Paris, France, pages 53–63. IEEE Computer Society Press, June 2002.

[SK01] K. Sartipi and K.Kontogiannis. *A Graph Pattern Matching Approach to Software Architecture Recovery.* In Proceedings of the 2001 IEEE International Conference on Software Maintenance, Florence, Italy, pages 408–418. IEEE Computer Society Press, November 2001.

[SWJ04] W. Schäfer, J.P. Wadsack, and J.H. Jahnke. *Software Reengineering - Die Suche nach verlorener Information*. ForschungsForum Paderborn, 7-2004, January 2004. Paderborner Universitätsmagazin, (to appear).

[Wad03] J.P. Wadsack. *Architectural Issues in Data Reengineering*. In Report of the Dagstuhl-Seminar 03061 on Software Architecture Recovery and Modeling. Schloss Dagstuhl, Germany, February 2003.

[War99] I. Warren. *The Renaissance of Legacy Systems - Method Support fpr Softwrae-System Evolution*. PRACTIONER SERIES. Springer Verlag, 1999.

[WJ02] J.P. Wadsack and J.H. Jahnke. *Towards Model-Driven Middleware Maintenance*. In Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the context of Model-Driven Architecture, Seattle, USA, November 2002. online at http://www.softmetaware.com/oopsla2002/mda-workshop.html.

[WJ03] J.P. Wadsack and J.H. Jahnke. *A History Concept for Recovery and Design Tools*. In Report of the Dagstuhl-Seminar 03061 on Software Architecture Recovery and Modeling. Schloss Dagstuhl, Germany, February 2003.

[WNGJ02] J.P. Wadsack, J. Niere, H. Giese, and J.H. Jahnke. *Towards Data Dependency Detection in Web Information Systems*. In Proceedings of the ICSM 2002 Database Maintenance and Reengineering Workshop, Montréal, Canada, pages 47–64. IEEE Computer Society Press, October 2002.

[ZSG79] M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.

[ZWR01] A. Zündorf, J.P. Wadsack, and I. Rockel. *Merging graph-like object structures*. In Proceedings of the 10th International Workshop on Software Configuration Management, Toronto, Canada, May 2001.

Chapter 2: Data-oriented Reengineering: A Case Study

[AOS+99] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan. *The Jini(TM) Specification*. Addison-Wesley, June 1999.

[BBD+03] I. Bilykh, Y. Bychkov, D. Dahlem, J.H. Jahnke, G. McCallum, C. Obry, A. Onabajo, and C. Kuziemsky. *Can GRID Services Provide Answers to the Challenges of National Health Information Sharing?* In D.A. Stewart, editor, Cascon 2003: Meeting of Minds, Toronto, Canada. IBM, October 2003.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1st edition, 1999.

[CC90] Elliot J. Chikofsky and James H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, 7(1):13–17, January 1990. IEEE Computer Society Press.

[Cim00] J. J. Cimino. *From Data to Knowledge through Concept-oriented Terminologies*. American Medical Informatics Association, 7(3):288–297, 2000. Hanley & Belfus, Inc. Medical Publishers.

[CVS] CVS. *Concurrent Versions System - The open standard for version control*. http://www.cvshome.org/.

[Dat89] C.J. Date. *A Guide to the SQL standard*. Addison-Wesley, 1989.

[GGT01] M. Gehrke, H. Giese, and M. Tichy. *A Jini-supported Distributed Version and Configuration Management System*. In Proceedings of the 2001 International Symposium on Convergence of IT and communications, Denver, USA, August 2001.

[HA00] K. Hogshead Davis and P. Aiken. *Data Reverse Engineering: A Historical Survey.* In Proceedings of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, pages 70–78. IEEE Computer Society Press, November 2000.

[HEH+95] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. *Requirements for information system reverse engineering support*. In Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, pages 136–145. IEEE Computer Society Press, July 1995.

[HHH+00] J-L. Hainaut, J. Henrard, J-M. Hick, J. Englebert, and D. Roland. *The Nature of Data Reverse Engineering*. In Proceedings of the Data Reverse Engineering Workshop EuroRef, 7th Reengineering Forum, Reengineering Week 2002. Zurich, Switzerland, March 2000.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[Moo01] L. Moonen. *Generating Robust Parsers using Island Grammars*. In Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, pages 13–22. IEEE Computer Society Press, October 2001.

[MRF+03] M. A. Munoz, M. Rodrigez, J. Favela, A.I. Martinez-Garcia, and V.M. Gonz lez. *Context-Aware Mobile Communication in Hospitals*. IEEE Computer, 36(9):38–46, September 2003. IEEE Computer Society Press.

[Nie03] J. Niere. *Inkremetelle Mustererkennung*. PhD thesis, University of Paderborn, Paderborn, Germany, December 2003.

[OBJ03] A. Onabajo, I. Bilykh, and J.H. Jahnke. *Wrapping Legacy Medical Systems for Integrated Health Network*. In NET.ObjectDAYS 2003, Erfurt, Germany. Springer Verlag, September 2003.

[Ona03] A. Onabajo. *GRID FEDERATION ENVELOPE (GFE): Federating Medical Information Systems*. Master's thesis, Department of Computer Science, University of Victoria, Victoria, Canada, September 2003.

[RT02] W. Raghupathi and J. Tan. *Strategic IT Applications oin Health Care*. Communications of the ACM, 45(12):56–61, December 2002. ACM Press.

[SL96] A. Schürr and M. Lefering. *Specification of Integration Tools*. In M. Nagl, editor, Building Tightly Integrated Software Development Environments: The IPSEN Approach, volume 1170 of Lecture Notes in Computer Science, pages 324–334. Springer Verlag, 1996.

[SPPB02] P. Sousa, M.L. Pedro-de-Jesus, G. Pereira, and F. Brito e Abreu. *Clustering Relations into abstract ER Schemas for database reverse engineering*. Journal of Science of Computer Programming, 45(2-3):137–153, November 2002. Elsevier Science Publishers B.V (North-Holland), (Special Issue on Software Maintenance and Reengineering).

[TS95] S.R. Tilley and D.B. Smith. *Perspectives on Legacy Systems Reengineering (draft)*. Reengineering Center, Softwrae Engineering Institute, Carnegie Mellon University, 1995. (available online at http://www.sei.cmu.edu/reengineering/lsysree.pdf).

[TWBK89] T. Teorey, G. Wei, D. Bolton, and J. Koenig. *ER Model Clustering as an Aid for User Communication and Documentation in Database Design*. Communications of the ACM, 32(8):975–987, August 1989. ACM Press.

[vDK99] A. van Dreusen and T. Kuipers. *Building Documentation Generators*. In Proceedings of the 9th International Conference on Software Maintenance, Oxford, UK, pages 40–49. IEEE Computer Society Press, September 1999.

[Wei84] M. Weiser. *Program slicing*. IEEE Transactions on Software Engineering, 10(4):352–357, July 1984. IEEE Computer Society Press.

[Wei02] G. Weiss. *Welcome To The (Almost) Digital Hospital*. IEEE Spectrum, 39(3):44–49, March 2002. IEEE Computer Society Press.

[Wil03] E.V. Wilson. *Asychronous Health Care Communication*. Communications of the ACM, 46(6):79–84, June 2003. ACM Press.

[WNGJ02] J.P. Wadsack, J. Niere, H. Giese, and J.H. Jahnke. *Towards Data Dependency Detection in Web Information Systems*. In Proceedings of the ICSM 2002 Database Maintenance and Reengineering Workshop, Montréal, Canada, pages 47–64. IEEE Computer Society Press, October 2002.

Chapter 3: Data-Oriented Reverse Engineering

[AFC98] G. Antoniol, R. Fiutem, and L. Christoforetti. *Design pattern recovery in object-oriented software*. In Proceedings of the 6th International Workshop on Program Comprehension, Ischia, Italy, pages 153–160. IEEE Computer Society Press, June 1998.

[AGG] Technical University of Berlin. *AGG, the Attributed Graph Grammar system*. Online at http://www.tfs.cs.tu-berlin.de/agg.

[Aik96] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1996.

[Amb02] S. Ambler. *Agile Database Techniques : Effective Strategies for the Agile Software Developer*, chapter 12: Database Refactoring. John Wiley and Sons, Inc., October 2002.

[And94] M. Andersson. *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering*. In Proceedings of the 13th International Conference of the Entity Relationship Approach, Manchester, volume 881 of Lecture Notes in Computer Science, pages 403–419. Springer Verlag, December 1994.

[AOS+99] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan. *The Jini(TM) Specification*. Addison-Wesley, June 1999.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database design*. Benjamin/Cummings, 1992.

[BDH+87] H. Briand, C. Ducateau, Y. Hebrail, D. Herin-Aime, and J. Kouloumdjian. *From Minimal Cover to Entity-Relationship Diagram*. In Proceedings of the 6th International Conference of the Entity-Relationship Approach, New York, USA, pages 287–304. North-Holland, November 1987.

[Bew98] B. Bewermeyer. *Cliche-Erkennung in relationalen Datenbankanwendungen*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, September 1998.

[BGD97] A. Behm, A. Geppert, and K. R. Dittrich. *On the Migration of Relational Schemas and Data to Object-Oriented Database Systems*. In Proceedings 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria, pages 13–33. Österreichische Computer Gesellschaft, December 1997.

[BGN+03] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*. In Proceedings of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), pages 51–56, September 2003. online at http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.p%df.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. *A Comparative Analysis of Methodologies for Database Schema Integration*. ACM Computing Surveys, 18(2):323–364, 1986. ACM Press.

[BP98] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[BR97] H. Blockeel and L. De Raedt. *Relational knowledge discovery in databases*. In S. Muggleton, editor, Proceedings of the 6th International Workshop on Inductive Logic Programming, Stockholm, Sweden, volume 1314 of Lecture Notes in Artificial Intelligence, pages 199–211, Berlin, August 1997. Springer Verlag.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1st edition, 1999.

[CBB+00] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O.Schadow, T. Staniendam, and F. Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, San Francisco (CA), USA, 2000.

[Cha96] D. Chappell. *Understanding ActiveX and OLE - A Guide for Developers and Managers*. Microsoft Press, 1996.

[Che76] P.P. Chen. *The Entity-Relationship Model – Toward a unified view of data*. ACM Transactions on Database Systems, 1(1):9–36, 1976. ACM Press.

[Chu04] D. Church. *Using mildly context-sensitive island grammars for semi-structured data extraction*. Master's thesis, Department of Computer Science, University of Victoria, Victoria, Canada, forthcoming in 2004.

[CL93] T. Catarci and M. Lenzerini. *Representing and Using Interschema Knowledge in a CooperativeInformation Systems*. International Journal of Intelligent and Cooperative Information Systems, 2(4):375–398, 1993. IEEE Computer Society Press.

[Cod70] E.F. Codd. *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13(6):377–387, June 1970. ACM Press.

[COR99] CORBA-2.3.1. *The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 2.3.1 Specification*. Object Managment Group, October 1999. Revision 2.3.1: OMG Technical Document formal/99-10-07.

[COS98] COSS-2.3. *CORBAservices: Common Object Services Specification*. Object Managment Group, December 1998. Revision 2.3, OMG technical document 98-12-09.

[Dat00] C.J. Date. *An introduction to database systems*. Addison-Wesley, 7th edition, 2000.

[EH99] V. Englebert and J.-L. Hainaut. *DB-MAIN: A Next Generation Meta-CASE*. Journal of Information Systems - Special Issue on Meta-CASEs, 24(2):99–112, June 1999. Elsevier Science Publishers B.V (North-Holland).

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In G. Engels and G. Rozenberg, editors, Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation, Paderborn, Germany, volume 1764 of Lecture Notes in Computer Science, pages 296–309. Springer Verlag, November 1998.

[Fon97] J. Fong. *Converting Relational to Object-Oriented Databases*. ACM SIGMOD Record, 26(1):53–58, March 1997. ACM Press.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FV95] C. Fahrner and G. Vossen. *Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93*. In Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases, Singapur, volume 1013 of Lecture Notes in Computer Science, pages 429–446. Springer Verlag, 1995.

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

[GO00] C.L. Gittens and S.L. Osborn. *Database Integration Using Graph Transformation*. In Joint 2000 APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, Berlin, Germany, pages 96–105. online at http://tfs.cs.tu-berlin.de/gratra2000, March 2000.

[Hai89] J.-L. Hainaut. *A Generic Entity-Relationship Model*. In Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis, Namur, Belgium. North-Holland, 1989.

[HCTJ93] J-L. Hainaut, M. Chandelon, C. Tonneau, and M. Joris. *Contribution to a Theory of Database Reverse Engineering*. In Proceedings of the Working Conference on Reverse Engineering, Baltimore, USA, pages 161–170. IEEE Computer Society Press, May 1993.

[HDZ00] J. Hatcli, M.B. Dwyer, and H. Zheng. *Slicing software for model construction*. Higher-Order and Symbolic Computation, 3(4):315–253, 2000. Kluwer Academic Publishers.

[HEH+95] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. *Requirements for information system reverse engineering support*. In Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, pages 136–145. IEEE Computer Society Press, July 1995.

[HG98] M. Harman and K.B. Gallagher. *Program Slicing - Introduction to the Special Issue on Program Slicing*. Information and Software Technology, 40(11):577–581, November/December 1998. Elsevier Science Publishers B.V (North-Holland).

[HH01] J.-L. Hainaut and J. Henrard. *Data dependency elicitation in database reverse engineering*. In Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, pages 11–19. IEEE Computer Society Press, 2001.

[HHEH96] J.-L. Hainaut, J.-M. Hick, V. Englebert, and J. Henrard. *Understanding Implementation of Is-A Relations*. In Proceedings of the 15th International Conference on the Entity-Relationship Approach, Cottbus, Germany, volume 1157, pages 42–50. Springer Verlag, 1996.

[HHH+96] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, and V. Englebert. *Database Design Recovery.* In Proceedings of the 8th Conference on Advance Information Systems Engineering, Crete, Greece, volume 1080 of Lecture Notes in Computer Science, pages 272–300. Springer Verlag, 1996.

[HHH+99] J. Henrard, J-L. Hainaut, J-M. Hick, D. Roland, and J. Englebert. *Data structure extraction in database reverse engineering*. In Proceedings of the 1st International Workshop on Reverse Engineering in Information Systems, Paris, France, pages 149–160, November 1999.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF - Reference Manual* -. SIGPLAN Notices, 24(11):43–75, 1989. ACM Press.

[HN90] M. T. Harandi and J. Q. Ning. *Knowledge Based Program Analysis*. IEEE Transactions on Software Engineering, 7(1):74–81, 1990. IEEE Computer Society Press.

[HTJC94] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. *Transformation-Based Database Reverse Engineering*. In Proceedings of the 12th International Conference on the Entity-Relationship Approach, Dallas, USA, volume 823, page 364. Springer Verlag, 1994.

[Ib96] CORBA IDL-binding. *Information technology – Information Resource Dictionary System (IRDS) Services Interface*. ISO/IEC, 1996. Amendment 3:1996 to ISO/IEC 10728:1993 CORBA IDL binding.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JCC] WebGain, Inc. *JavaCC, the Java Parser Generator. Online at http://www.experimentalstuff.com/Technologies/JavaCC/ (last visited June 2003)*.

[JJ94] U.A. Johnen and M.A. Jeusfeld. *An Executable Meta Model for Re-Engineering of Database Schemas*. In Proceedings of the 13th International Conference on the Entity-Relationship Approach, Manchester, UK, number 885 in Lecture Notes in Computer Science, pages 533–547. Springer Verlag, March 1994.

[JK89] P. Johannesson and K. Kalman. *A Method for Translating Relational Schemas into Conceptual Schemas*. In F.H. Lochovsky, editor, Proceedings of the 8th International Conference on Entity-Relationship Approach, Toronto, Canada, pages 271–285. North-Holland, October 1989.

[JSZ96] J. Jahnke, W. Schäfer, and A. Zündorf. *A Design Environment for Migrating Relational to Object-Oriented Database Systems*. In Proceedings of the 6th International Conference on Software Maintenance, Monterrey, USA, pages 163–170, November 1996.

[JSZ97] J.H. Jahnke, W. Schäfer, and A. Zündorf. *Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications*. In Proceedings of the 6th European Software Engineering Conference, volume 1302 of Lecture Notes in Computer Science, pages 193–210. Springer Verlag, September 1997.

[JZ99] J.H. Jahnke and A. Zündorf. *Applying Graph Transformations To Database Re-Engineering*. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation, volume 2 - Application, Languages and tools., pages 267–284. World Scientific, Singapore, 1999.

[KM96] A. Konar and A. K. Mandal. *Uncertainty Management in Expert Systems Using Fuzzy Petri Nets*. IEEE Transactions on Knowledge and Data Engineering, 8(1):96–105, February 1996. Academic Press, London.

[KP96] C. Krämer and L. Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software*. In Proceedings of the 3rd Working Conference on Reverse Engineerinung, Monterey, CA, pages 208–215. IEEE Computer Society Press, November 1996.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. *Pattern-Based Reverse-Engineering of Design Components*. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE Computer Society Press, May 1999.

[LCBO03] P.K. Linos, Z. Chen, S. Berrier, and B. O'Rourke. *A Tool for Understanding Multi-Language Program Dependencies*. In Proceedings of the 11th International Workshop on Program Comprehension (IWPC), Portland, USA, pages 64–72. IEEE Computer Society Press, May 2003.

[Lef95] Martin Lefering. *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Informatik. Verlag Shaker, 1995.

[LNE89] J.A. Larson, S.B. Navathe, and R. Elmasri. *A Theory of Attribute Equivalence in Databases with Application to Schema Integration*. IEEE Transactions on Software Engineering, 15(4):449–463, April 1989. IEEE Computer Society Press.

[MCAH95] P. Martin, J.R. Cordy, and R. Abu-Hamdeh. *Information Capacity Preserving of Relational Schemas Using Structural Transformation*. Technical report, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, November 1995.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1st edition, 1984.

[MJS+00] H.A. Müller, J.H. Jahnke, D.B. Smith, M.A. Storey, and K. Wong. *Reverse Engineering: a roadmap*. In A. Finkelstein, editor, Future of Software Engineering. International Conference on Software Engineering, Limerick, Ireland, pages 47–60. ACM Press, June 2000.

[MM90] R.W. Mathews and W.C. McGee. *Data modeling for software develpment*. IBM Systems Journal, 29(2):228–235, 1990. IBM.

[Moo01] L. Moonen. *Generating Robust Parsers using Island Grammars*. In Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, pages 13–22. IEEE Computer Society Press, October 2001.

[NA87] S.B. Navathe and A.M. Awong. *Abstracting Relational and Hierarchical Data with a Semantic Data Model*. In Proceedings of the 6th International Conference of the Entity-Relationship Approach, New York, USA, pages 305–333. North-Holland, November 1987.

[Nie03] J. Niere. *Inkremetelle Mustererkennung*. PhD thesis, University of Paderborn, Paderborn, Germany, December 2003.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. *Towards Pattern-Based Design Recovery.* In Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, pages 338–348. ACM Press, May 2002.

[NWW01] J. Niere, J.P. Wadsack, and L. Wendehals. *Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic*. Technical Report tr-ri-01-222, University of Paderborn, Paderborn, Germany, March 2001.

[NWW03] J. Niere, J.P. Wadsack, and L. Wendehals. *Handling Large Search Space in Pattern-Based Reverse Engineering*. In Proceedings of the 11th International Workshop on Program Comprehension, Portland, USA, pages 274–280. IEEE Computer Society Press, May 2003.

[OTS98] OTS-1.1. *Transaction Service Specification*. Object Management Group, February 1998. The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 1.1 Specification, Revision 1.1: OMG Technical Document formal/97-12-17.

[Pal01] M. Palasdies. *Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, May 2001.

[PB94] W.J. Premerlani and M.R. Blaha. *An Approach for Reverse Engineering of Relational Databases*. Communications of the ACM, 37(5):42–49, May 1994. ACM Press.

[PKBT94] J-M. Petit, J. Kouloumdjian, J-F. Boulicaut, and F. Toumani. *Using Queries to Improve Database Reverse Engineering*. In Proceedings of 13th International Conference of the Entity-Relationship Approach, Manchester, UK, number 885 in Lecture Notes in Computer Science, pages 369–386. Springer Verlag, March 1994.

[PP94] S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns*. IEEE Transactions on Software Engineering, 20(6):463–475, June 1994. IEEE Computer Society Press.

[PTBK96] J-M. Petit, F. Toumani, J. Boulicaut, and J. Kouloumdjian. *Towards the reverse engineering of denormalized relational databases*. In Proceedings 12th International Conference on Data Engineering, pages 218–227, New Orleans, February 1996. IEEE Computer Society Press.

[Qui94] A. Quilici. *A Memory-Based Approach to Recognizing Programming Plans*. Communications of the ACM, 37(5):84–93, May 1994. ACM Press.

[Rad99] A. Radermacher. *Support for Design Patterns through Graph Transformation Tools*. In Proceedings of 1999 International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance, Kerkrade, The Netherlands, volume 1779 of Lecture Notes in Computer Science, pages 111–126. Springer Verlag, 1999.

[Rec01] C. Reckord. *Entwurf eines generischen Sichtenkonzeptes für die Entwicklungsumgebung Fujaba*. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, February 2001.

[RH96] S. Ramanathan and J. Hodges. *Reverse Engineering Relational Schemas to Object-Oriented Schemas*. Technical Report MSU-960701, Department of Computer Science, Mississippi State University, USA, July 1996.

[RH97] S. Ramanathan and J. Hodges. *Extraction of Object-Oriented Structures from Existing Relational Databases*. ACM SIGMOD Record, 26(1), March 1997. ACM Press.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[RSK91] M. Rusinkiewisz, A. Sheth, and G. Karabatis. *Specifying Interdatabase Dependencies in a Multidatabase Environment*. IEEE Computer, 24(12):46–53, December 1991. IEEE Computer Society Press.

[SA03] M. M. Sufyan Beg and N. Ahmad. *Soft Computing Techniques for Rank Aggregation on the World Wide Web*. World Wide Web: Internet and Web Information Systems, 6(1):5–22, March 2003. Kluwer Academic Publishers.

[SCC+93] Y.-P. Shan, T. Cargill, B. Cox, W. Cook, M. Loomis, and A. Snyder. *Is Multiple Inheritance Essential to OOP?* SIGPLAN Notices - Proccedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, 28(10):360–363, October 1993. ACM Press.

[Sch94] Andy Schürr. *Specification of Graph Translators with Triple Graph Grammars*. In Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, pages 151–163, Herrschin, Germany, June 1994. Spinger Verlag.

[Sch01] M.A. Schwarz. *Integration eines inkrementellen Parsing - Algorithmus in Fujaba*. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, September 2001.

[SK90] F.N. Springsteel and C. Kou. *Reverse Data Engineering of E-R Designed Relational Schemas*. In Proceedings of the 1st Databases, Parallel Architectures and their Applications, Miami Beach, USA, pages 438–440. Springer Verlag, March 1990.

[SL96] A. Schürr and M. Lefering. *Specification of Integration Tools*. In M. Nagl, editor, Building Tightly Integrated Software Development Environments: The IPSEN Approach, volume 1170 of Lecture Notes in Computer Science, pages 324–334. Springer Verlag, 1996.

[SLGC94] O. Signore, M. Loffredo, M. Gregori, and M. Cima. *Reconstruction of ER Schema from Database Applications: a Cognitive Approach*. In Proceedings of 13th International Conference on the Entity-Relationship Approach, Manchester, UK, number 885 in Lecture Notes in Computer Science, pages 387–402. Springer Verlag, March 1994.

[Sou98a] C. Soutou. *Inference of Aggregate Relationships through Database Reverse Engineering*. In Proceedings of 17th International Conference on Conceptual Modeling, Singapore, volume 1507 of Lecture Notes in Computer Science, pages 135–149. Springer Verlag, November 1998.

[Sou98b] C. Soutou. *Relational Database Reverse Engineering: Extraction of Cardinality Constraints*. Data and Knowledge Engineering, 28(2):161–207, November 1998. Elsevier Science Publishers B.V (North-Holland).

[SvG98] J. Seemann and J.W. von Gudenberg. *Pattern-Based Design Recovery of Java Software*. ACM SIGSOFT Software Engineering Notes, 23(6):10–16, November 1998. ACM Press.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. *Graph Grammar Engineering with PROGRES*. In W. Schäfer and P. Botella, editors, Proceedings of 5th European Software Engineering Conference, Barcelona, Spain, volume 989 of Lecture Notes in Computer Science, pages 219–234. Springer Verlag, September 1995.

[Szy99] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[TA99] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference*. In Proceedings of the 9th International Conference on Software Maintenance, Oxford, UK, pages 230–238. IEEE Computer Society Press, September 1999.

[TGF00] A-R. H. Tawil, W. A. Gray, and N. J. Fiddian. *Discovering and Representing InterSchema Semantic Knowledge in a Cooperative Multi-Information Server Environment*. In M. T. Ibrahim, J. Küng, and N. Revell, editors, Proceedings of the 11th International Conference on Database and Expert Systems Applications, London, UK, volume 1873 of Lecture Notes in Computer Science, pages 548–562. Springer Verlag, September 2000.

[Tip95] F. Tip. *A Survey of Program Slicing Techniques*. Journal of programming languages, 3(3):121–189, September 1995. Chapman & Hall.

[vDK99] A. van Dreusen and T. Kuipers. *Building Documentation Generators*. In Proceedings of the 9th International Conference on Software Maintenance, Oxford, UK, pages 40–49. IEEE Computer Society Press, September 1999.

[Wad98] J.P. Wadsack. *Inkrementell Konsistenzerhaltung in der transformationsbasierten Datenbankmigration*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1998.

[Wei84] M. Weiser. *Program slicing*. IEEE Transactions on Software Engineering, 10(4):352–357, July 1984. IEEE Computer Society Press.

[Wen01] L. Wendehals. *Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, October 2001.

[Wen03] L. Wendehals. *Improving Design Pattern Instance Recognition by Dynamic Analysis*. In Proceedings of the ICSE 2003 Workshop on Dynamic Analysis, Portland, USA, pages 29–32, May 2003. online at http://www.cs.nmsu.edu/ jcook/woda2003/woda2003.pdf.

[Wil96] L.M. Wills. *Using Attributed Flow Graph Parsing to Recognize Programs*. In J.E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, Proceedings of 5th International Workshop on Graph Grammars and Their Application to Computer Science, Williamsburg, USA, volume 1073 of Lecture Notes in Computer Science, pages 170–184, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

[WNGJ02] J.P. Wadsack, J. Niere, H. Giese, and J.H. Jahnke. *Towards Data Dependency Detection in Web Information Systems*. In Proceedings of the ICSM 2002 Database Maintenance and Reengineering Workshop, Montréal, Canada, pages 47–64. IEEE Computer Society Press, October 2002.

[Zad65] L.A. Zadeh. *Fuzzy Sets*. Information and Control, 8:338–353, 1965. Elsevier Science Publishers B.V (North-Holland).

[Zün95] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, RWTH Aachen, 1995.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. (draft available online: http://www.uni-paderborn.de/fachbereich/AG/schaefer/Personen/Ehemalige/Zuend%orf/AZRigSoftDraft_0_2.pdf).

Chapter 4: Data Component Extension

[ABM96] A. Aarsten, D. Brugali, and G. Menga. *Patterns for Three-Tier Client/Server Applications*. In Proceedings of the 3rd Conference on the Pattern Languages of Programs, Urbana-Champaign, USA. Washington University, Technical Report# WUCS-97-07, September 1996.

[ADD+91] R. Ahmed, P. De Smedt, W. Du, W. Kent, M.A. Ketabchi, W. Litwin, A. Rafii, and M.-C. Shan. *The Pegasus Heterogeneous Multidatabase System*. IEEE Computer, 24(12), December 1991. IEEE Computer Society Press.

[AL99] N. Anquetil and T. Lethbridge. *Recovering software architecture from the names of source files*. In Proceedings of the 6th Working Conference on Reverse Engineering, Atlanta, USA, pages 235–255. IEEE Computer Society Press, October 1999.

[Amb02] S.W. Ambler. *Deriving Web services from UML models*. developerWorks newsletter: technology edition, March 2002. online at http://www-106.ibm.com/developerworks/webservices/library/ws-uml1.

[Anq00] N. Anquetil. *A comparison of graphs of concept for reverse engineering*. In Proceedings of the 8th International Workshop on Program Comprehension, Limerick, Irland, pages 231–240. IEEE Computer Society Press, June 2000.

[BGD97] A. Behm, A. Geppert, and K. R. Dittrich. *On the Migration of Relational Schemas and Data to Object-Oriented Database Systems*. In Proceedings 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria, pages 13–33. Österreichische Computer Gesellschaft, December 1997.

[BGN+03] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*. In Proceedings of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), pages 51–56, September 2003. online at http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.p%df.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Somerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, Inc., 1st edition, 1996.

[CER99] CERMICS Database Team, 2004 route des lucioles, BP93, 06902 Sophia Antipolis Cedex, France. *ObjectDRIVER V1.1 User Manual*, 1999.

[CHK+91] T. Connors, W. Hasan, C. Kolovson, M.A. Neimat, D. Schneider, and K. Wilkinson. *The Papyrus Integrated Data Server*. In Proceddings of the 1st International Parallel and Distributed Information Systems Conference, Miami Beach, USA, page 139. IEEE Computer Society Press, December 1991.

[CHS+95] M-J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. *Towards Heterogeneous Multimedia Information Systems: The Garlic Approach*. In M.T. Ozsu and M.C. Shan, editors, 5th International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, pages 124–131. IEEE Computer Society Press, March 1995.

[CL00] J.L. Cybulski and T. Linden. *Composing Multimedia Artifacts for Reuse*. In N. Harrison, B. Foote, and H. Rohnert, editors, Pattern Languages of Program Design 4, pages 461–488. Addison-Wesley, 2000.

[Coc96] A. Cockburn. *Prioritizing Forces in Software Design*. In J.M. Vlissides, J.O. Coplien, and N.L. Kerth, editors, Pattern Languages of Program Design 2, pages 317–333. Addison-Wesley, 1996.

[CS90] S. Choi and W. Scacchi. *Extracting and restructuring the design of large systems*. IEEE Software, 7(1):66–71, January 1990. IEEE Computer Society Press.

[CWMY02] F. Chen, Q. Wang, H. Mei, and F. Yang. *An Architecture-Based Approach for Component-Oriented Development*. In In Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, pages 450–455. IEEE Computer Society Press, August 2002.

[DB00] J. Davey and E. Burd. *Evaluating the Suitability of Data Clustering for Software Remodularisation*. In Proceedings of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, pages 268–277. IEEE Computer Society Press, November 2000.

[ES02] G. Engels and S. Sauer. *Object-oriented Modeling of Multimedia Applications*. In S.K. Chang, editor, Handbook of Software Engineering and Knowledge Engineering, volume 2, pages 21–53. World Scientific, Singapore, 2002.

[FF99] E.B. Fernandez and R. Flanders. *Data Filter Architecture Pattern*. In Proceedings of the 6th Conference on the Pattern Languages of Programs, Urbana-Champaign, USA, August 1999. online at http://jerry.cs.uiuc.edu/plop/plop99/proceedings.

[FGS93] D.D. Fang, S. Ghandeharizadeh, and A. Si. *The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database Systems*. In Proceedings of the 9th International Conference on Data Engineering, Vienna, Austria, pages 467–475. IEEE Computer Society Press, April 1993.

[FM86] P. Feldman and D. Miller. *Entity Model Clustering: Structuring A Data Model By Abstraction*. The Computer Journal, 29(4):348–360, 1986. Oxford University Press.

[Fon97] J. Fong. *Converting Relational to Object-Oriented Databases*. ACM SIGMOD Record, 26(1):53–58, March 1997. ACM Press.

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

[GMS01] H. Gomaa, D.A. Menasc, and M. E. Shin. *Reusable component interconnection patterns for distributed software architectures*. In In Proceedings of the 2001 Symposium on Software Reusability: putting software reuse in context, Toronto, Canada, pages 69–77. ACM Press, May 2001.

[Gra99] M. Grand. *Transaction Patterns: A Collection of Four Transaction Related Patterns*. In Proceedings of the 6th Conference on the Pattern Languages of Programs, Urbana-Champaign, USA, August 1999. online at http://jerry.cs.uiuc.edu/ plop/plop99/proceedings.

[GSNW02] A. Gokhale, D.C. Schmidt, B. Natarajan, and N. Wang. *Applying Model-Integrating Computing to Component Middleware and Enterprise Application*. Communications of the ACM, 45(10):65–69, October 2002. ACM Press.

[GW01] H. Giese and J.P. Wadsack. *Reengineering for Evolution of Distributed Information Systems*. In Proceedings of the 3rd International Workshop on Net-Centric Computing: Migrating to the Web, Toronto, Canada, pages 36–39. ACM Press, May 2001.

[GZ02] L. Geiger and A. Zündorf. *Graph Based Debugging with Fujaba*. Electronic Notes in Theoretical Computer Science, 72(2), November 2002. Elsevier Science Publishers B.V (North-Holland).

[HC01] G.T. Heinemann and W.T. Councill. *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley, 2001.

[HK02] I. Hammouda and K. Koskimies. *Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans*. In In Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, pages 856–866. IEEE Computer Society Press, August 2002.

[HL02] P. Hoven and M. Liebrecht. *Entwurf und Implementierung einer Import/ Export Funktionalität für die Entwicklungsumgebung Fujaba*. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, August 2002.

[HMN+99] L.M. Haas, R.J. Miller, B. Niswonger, M. Tork Roth, P.M. Schwarz, and E.L. Wimmers. *Transforming Heterogeneous Data with Database Middleware: Beyond Integration*. Bulletin of the IEEE Computer Socisty Technical Committee on Data Engineering, 2(1):31–36, March 1999. IEEE Computer Society Press.

[HWSS00] R.C. Holt, A. Winter, A. Schürr, and S. Sim. *GXL: Towards a Standard Exchange Format*. In Proceedings of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, pages 162–171, Brisbane, Australia, November 2000. IEEE Computer Society Press.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JOS93] P. Jaeschke, A. Oberweis, and W. Stucky. *Extending ER Model Clustering by Relationship Clustering*. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, Proceedings of the 12th International Conference on Entity-Relationship Approach, Arlington, Texas, USA, volume 823 of Lecture Notes in Computer Science, pages 451–462. Springer Verlag, December 1993.

[Kam03] M.L. Modjo Kamneng. *Entwurfsunstresttzung Web-basierter Schnittstellen aud Basis der UML*. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, April 2003.

[Köl98] U. Kölsch. *Object-Oriented Re-Engineering of Information Systems in a Heterogeneous Distributed Environment*. In Proceedings of the 5th Working Conference on Reverse Engineering, Hawaii, USA, pages 104–114. IEEE Computer Society Press, October 1998.

[LB03] J.A. Landay and G. Borriello. *Design Patterns for Ubiquitous Computing*. IEEE Computer, 36(8):93–95, August 2003. IEEE Computer Society Press.

[Lin99] D. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.

[LS97] C. Lindig and G. Snelting. *Assessing modular structure of legacy code based on mathematical concept analysis*. In Proceedings of the 19th International Conference on Software Engineering, Boston, USA, pages 349–359. ACM Press, May 1997.

[Lun98] C.-H. Lung. *Software architecture recovery and restructuring through clustering techniques*. In Proceedings of the 3rd international workshop on Software architecture, Orlando, USA, pages 101–104. ACM Press, November 1998.

[Man99] D.-A. Manolescu. *Feature Extraction: A Pattern for Information Retrieval*. In N. Harrison, B. Foote, and H. Rohnert, editors, Pattern Languages of Program Design 4, pages 391–412. Addison-Wesley, December 1999.

[Meu95] R. Meunier. *The Pipes and Filters Architecture*. In J.O. Coplien and D.C. Schmidt, editors, Pattern Languages of Program Design 1, pages 427–440. Addison-Wesley, October 1995.

[MHH+01] R.J. Miller, M.A. Hernandez, L.M. Haas, L. Yan, C.T.H. Ho, and R. Fagin andL. Popa. *The Clio Project: Managing Heterogeneity.* ACM SIGMOD Record, 30(1):78–83, March 2001. ACM Press.

[MMCG99] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. *Bunch: A clustering tool for the recovery and maintenance of software system structures*. In Proceedings of the 9th International Conference on Software Maintenance, Oxford, UK, pages 50–59. IEEE Computer Society Press, August 1999.

[MMR+98] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. *Using automatic clustering to produce high-level system organizations of source code*. In Proceedings 6th International Workshop on Program Comprehension, Ischia, Italy, pages 45–52. IEEE Computer Society Press, June 1998.

[MOTU93] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. *A Reverse Engineering Approach To Subsystem Structure Identification*. Journal of Software Maintenance, 5(4):181–204, December 1993. John Wiley and Sons, Inc.

[Mul95] D.E. Mularz. *Pattern-Based Integration Architectures*. In J.O. Coplien and D.C. Schmidt, editors, Pattern Languages of Program Design 1, pages 441–452. Addison-Wesley, October 1995.

[Obj99a] The Object People Inc., 885 Meadowlands Dr., Suite 509, Ottawa, Ontario. *TOPLink for Java 2.0 User's Manual*, 1999.

[Obj99b] ObjectMatter Inc., 2450 S.W. 137 Ave. Suite 206 Miami, Fl. 33175, UNITED STATES. *Objectmatter VBSF Object-Relational Framework V2.02 User Manual*, 1999.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. *Object Exchange Across Heterogeneous Information Sources*. In P.S. Yu and A.L.P. Chen, editors, Proceedings of the 11th International Conference on Data Engineering, Taipei, Taiwan, pages 251–260. IEEE Computer Society Press, March 1995.

[RB01] E. Rahm and P.A. Bernstein. *A survey of approaches to automatic schema matching*. The International Journal on Very Large Data Bases, 10(4):334–450, December 2001. Springer Verlag.

[Ris00] L. Rising. *The Pattern Almanac 2000*. Addison-Wesley, 2000.

[RL82] R. Rosenberg and T. Landers. *An Overview of MULTIBASE*. In H. Schneider, editor, Distributed Databases, pages 153–184. North Holland, 1982.

[RR98] S. Ram and V. Ramesh. *Schema integration: Past, Present, and Future*. In A. Elamagrid, M. Rusinkiewicz, and A. Sheth, editors, Management of Heterogeneous and Autonomous Databse Systems, pages 119–155. Morgan-Kaufmann, San Mateo, CA, 1998.

[Sch97] D.E. Schmidt. *Acceptor and Connector.* In R.C. Martin, D. Riehle, and F. Buschmann, editors, Pattern Languages of Program Design 3, pages 191–229. Addison-Wesley, October 1997.

[Sch99] D.C. Schmidt. *Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes*. In C++ Report, volume 11. SIGS Publications, February 1999.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, 1996.

[SH94] R.W. Schwanke and S.J. Hanson. *Using Neural Networks to Modularize Software*. Machine Learning, 15(2):137–168, May 1994. Kluwer Academic Publishers.

[Sie98] Siemens AG - C-LAB, Fürstenallee 11, 33102 Paderborn, Germany. *OpenDM ODMG User's Guide*, 1998.

[SMB+03] M. Saeed, O. Maqbool, H.A. Babri, S.Z. Hassan, and S.M. Sarwar. *Software Clustering Techniques and the Use of Combined Algorithm*. In Proceedings of the 7th European Conference On Software Maintenance And Reengineering, Benevento, Italy, pages 301–306. IEEE Computer Society Press, March 2003.

[Sne02] H.M. Sneed. *Using XML to Integrate Existing Software Systems into the Web*. In In Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, pages 167–172. IEEE Computer Society Press, August 2002.

[SPPB02] P. Sousa, M.L. Pedro-de-Jesus, G. Pereira, and F. Brito e Abreu. *Clustering Relations into abstract ER Schemas for database reverse engineering*. Journal of Science of Computer Programming, 45(2-3):137–153, November 2002. Elsevier Science Publishers B.V (North-Holland), (Special Issue on Software Maintenance and Reengineering).

[Szy99] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[TCHH99] P. Thiran, A. Chougrani, J-M. Hick, and J-L. Hainaut. *Generation of Conceptual Wrappers for Legacy Databases*. In Proceedings of the 10th International Conference on Database and Expert Systems Applications, Florence, Italy, pages 678–687. Springer Verlag, September 1999.

[TH97] V. Tzerpos and R. Holt. *The orphan adoption problem in architecture maintenance*. In Proceedings of the 4th Working Conference on Reverse Engineering, Amsterdam, The Netherlands, pages 76–83. IEEE Computer Society Press, October 1997.

[THB+98] P. Thiran, J-L. Hainaut, S. Bodart, A. Deflorenne, and J-M. Hick. *Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE Support: the InterDB Approach*. In Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York, USA, pages 54–63. IEEE Computer Society Press, August 1998.

[Tho99] Thought Inc., 657 Mission Street, Suite 202, San Francisco, CA 94105, USA. *CocoBase WhitePaper*, 1999.

[TWBK89] T.J. Teorey, G. Wei, D.L. Bolton, and J.A. Koenig. *ER Model Clustering as an Aid for User Communication and Documentation in Database Design*. Communications of the ACM, 32(8):975 – 987, August 1989. ACM Press.

[vC97] J.A. van den Broecke and J.O. Coplien. *Using Design Patterns to Build a Framework for Multimedia Networking*. In Bell Labs Technical Journal, pages 166–187. Lucent Technologies Inc., winter 1997.

[vK99] A. van Deursen and T. Kuipers. *Identifying objects using cluster and concept analysis*. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 246–255. ACM Press, May 1999.

[Wag01] S. Wagner. *Datenbank-Erweiterungen für multimediale Anwendungen*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 2001.

[War99] I. Warren. *The Renaissance of Legacy Systems - Method Support fpr Softwrae-System Evolution*. PRACTIONER SERIES. Springer Verlag, 1999.

[Wie92] G. Wiederhold. *Mediatiors in the architecture of duture information systems*. IEEE Computer, 14(2), March 1992. IEEE Computer Society Press.

[Wie95] G. Wiederhold. *Mediation in information systems*. ACM Computing Surveys, 27(2), m 1995. ACM Press.

[Wig97] T.A. Wiggerts. *Using Clustering Algorithms in Legacy Systems Remodularization*. In Proceedings of the 4th Working Conference on Reverse Engineering, Amsterdam, The Netherlands, pages 33–43. IEEE Computer Society Press, October 1997.

[Wol01] F. Wolf. *Entwicklung eines Generators fr eine objektorientierte Zugriffsschicht auf einer relationalen Datenbank*. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, April 2001.

[XA94] XA. *Distributed Transaction processing: The XA+ Specification, version 2*. X/Open Group, 1994. X/Open Company, Reading, UK.

Chapter 5: Model Consistency Management

[CMR96] A. Corradini, U. Montanari, and F. Rossi. *Graph Processes*. Fundamenta Informaticae, 26(3):241–265, June 1996. IOS Press, Amsterdam.

[CVS] CVS. *Concurrent Versions System - The open standard for version control*. http://www.cvshome.org/.

[HHE+99] J.-M. Hick, J.-L. Hainaut, V. Englebert, D. Roland, and J. Henrard. *Strategies pour l'evolution des applications de bases de donnees relationnelles: l'approche DB-MAIN*. In In XVIIe congress INFORSID, La Garde, France, June 1999.

[HWSS00] R.C. Holt, A. Winter, A. Schürr, and S. Sim. *GXL: Towards a Standard Exchange Format*. In Proceedings of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, pages 162–171, Brisbane, Australia, November 2000. IEEE Computer Society Press.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JSWZ02] J.H. Jahnke, W. Schäfer, J.P. Wadsack, and A. Zündorf. *Supporting Iterations in Exploratory Database Reengineering Processes*. Journal of Science of Computer Programming, 45(2-3):99–136, November 2002. Elsevier Science Publishers B.V (North-Holland), (Special Issue on Software Maintenance and Reengineering).

[JW99] J.H. Jahnke and J.P. Wadsack. *Integration of analysis and redesign activities in information system reengineering*. In P. Nesi and C. Vernoef, editors, Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, Amsterdam, The Nederlands, pages 160–168. IEEE Computer Society Press, March 1999.

[JWZ02] J.H. Jahnke, J.P. Wadsack, and A. Zündorf. *A History Concept for Design Recovery Tools*. In Proceedings of the 4th European Conference on Software Maintenance and Reengineering, Budapest, Hungary, pages 59–69. IEEE Computer Society Press, March 2002.

[Nag96] M. Nagl, editor. *The IPSEN Approach*, volume 1170 of Lecture Notes in Computer Science. Springer Verlag, 1996.

[NR99] A. Nica and E.A. Rundensteiner. *View Maintenance after View Synchronization*. In International Database Engineering and Application Symposium, Montréal, Canada, pages 215–223. IEEE Computer Society Press, June 1999.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[SL96] A. Schürr and M. Lefering. *Specification of Integration Tools*. In M. Nagl, editor, Building Tightly Integrated Software Development Environments: The IPSEN Approach, volume 1170 of Lecture Notes in Computer Science, pages 324–334. Springer Verlag, 1996.

[vv02] C. van der Westhuizen and A. van der Hoek. *Understanding and Propagating Architecutural Changes*. In J. Bosch, W.M. Gentleman, C. Hofmeister, and J. Kuusela, editors, IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, Montréal, Canada, volume 224 of IFIP Conference Proceedings, pages 95–109. Kluwer Academic Publishers, August 2002.

[Wad98] J.P. Wadsack. *Inkrementell Konsistenzerhaltung in der transformationsbasierten Datenbankmigration*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1998.

Chapter 6: Conclusions

[BGN+03] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*. In Proceedings of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), pages 51–56, September 2003. online at http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.p%df.

[BS95] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems - Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann Publishers, San Francisco, 1995.

[BSS+99] J. Bergey, D. Smith, Tilley S, N. Weiderman, and S. Woods. *Why Reengineering Projects Fail*. Technical Report CMU/SEI-99-TR-010, Carnegie Mellon Software Engineering Institute, Pittsburg, USA, April 1999.

[DRT98] S. Demeyer, M. Rieger, and S. Tichelaar. *Three Reverse Engineering Patterns*, 1998. darft available at http://www.iam.unibe.ch/ famoos/ Deme98p/threerevpat.pdf.

[ESS02] M. El-Ramly, E. Stroulia, and P. Sorenson. *Mining System-User Interaction Traces for Use Case Models*. In Proceedings of the 10th International Workshop on Program Comprehension, Paris, France, pages 21–30. IEEE Computer Society Press, June 2002.

[Hal96] C. L. Hall. *Building Client/Server Applications Using TUXEDO*. John Wiley and Sons, Inc., 1996.

[Hou98] P. Houston. *Building Distributed Applications with Message Queuing Middleware*. Microsoft Cooperation, 1998.

[Hud94] E. S. Hudders. *CICS: A Guide to Internal Strucure*. John Wiley and Sons, Inc., 1994.

[JTS99] JTS. *Java Transaction Service (JTS)*. Sun Microsystems Inc., December 1999. Version 1.0.

[KK01] C. Kerer and E. Kirda. *Layout, Content and Logic Separation in Web Engineering*. In S. Murugeasn and Y. Deshpande, editors, Web Engineering - Managing Diversity and Complexity of Web Application Development, volume 2016 of Lecture Notes in Computer Science, pages 135–147. Springer Verlag, 2001.

[Lew99] R. Lewis. *Advanced Messaging Applications with MSMQ and MQSeries*. Que, 1999.

[LT01] C. Liebig and S. Tai. *Middleware Mediated Transactions*. In Proceedings of the 3rd International Symposium on Distributed Objects & Applications. Rome, Italy, pages 340–350. IEEE Computer Society Press, September 2001.

[MBPRR01] R.T. Mittermeir, A. Bollin, H. Pozewaunig, and D. Rauner-Reithmayer. *Goal-Driven Combination of Softwrae Comprehension Approcahes for Component Based Development*. In Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, Toronto, Canada, pages 95–102. ACM Press, May 2001.

[MC01] J. Moe and D.A. Carr. *Understanding Distributed Systems via Execution Trace Data*. In Proceedings of the 9th International Workshop on Program Comprehension, Toronto, Canada, pages 60–69. IEEE Computer Society Press, May 2001.

[MCZE02] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. *XMIDDLE: A Data-Sharing Middleware for Mobile Computing*. Journal on Wireless Personal Communications, 2002. Kluwer Academic Publishers.

[MG00] S.A. Mondal and K.D. Gupta. *Choosing a Middleware for Web-Integration of a Legacy Application*. Software Engineering Notes, 25(3):50–53, May 2000. ACM Press.

[MMK02] M. Morrison, J. Morrison, and A. Keys. *Integrating Web Sites and Databases*. Communications of the ACM, 45(9):81–86, September 2002. ACM Press.

[OTS98] OTS-1.1. *Transaction Service Specification*. Object Management Group, February 1998. The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 1.1 Specification, Revision 1.1: OMG Technical Document formal/97-12-17.

[Rot03] A. Rott. *Werkzeug-untersttzte Optimierung komplexer Datenbank- und Applikationsserverumgebungen auf Basis von Zugriffsanalysen*. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, January 2003.

[SL02] T. Schattkowsky and M. Lohmann. *Rapid Development of Modular Dynamic Web Sites using UML*. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, In Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany, volume 2460 of Lecture Notes in Computer Science, pages 336–350. Springer Verlag, October 2002.

[Wad03] J.P. Wadsack. *Enterprise Application Performance Optimization based on Request-centric Monitoring and Diagnosis*. In Proceedings of the Workshop on Remote Analysis and Measurement of Software Systems, Portland, USA, pages 27–30. IEEE Computer Society Press, May 2003.

[WML02] J. Whaley, M.C. Martin, and M.S. Lam. *Automatic Extraction of Object-Oriented Component Interfaces*. In Proceedings of the 2002 International Symposium of Software Testing and Analysis, Roma, Italy, pages 218–228. ACM Press, July 2002.

[XA94] XA. *Distributed Transaction processing: The XA+ Specification, version 2*. X/Open Group, 1994. X/Open Company, Reading, UK.

[YR01] Z. Yu and V. Raijlich. *Hidden Dependencies in Program Comprehension and Change Propagation*. In Proceedings of the 9th International Workshop on Program Comprehension, Toronto, Canada, pages 293–299. ACM Press, May 2001.

INDEX