An ORDBMS-based Reuse Repository Supporting the Quality Improvement Paradigm –Exemplified by the SDL-Pattern Approach–

Raimund L. Feldmann, Birgit Geppert*, Wolfgang Mahnke, Norbert Ritter, and Frank Rößler*
University of Kaiserslautern, Computer Science Department
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{feldmann, mahnke, ritter}@informatik.uni-kl.de {b.geppert, f.roessler}@computer.org

Abstract

Comprehensive reuse and systematic evolution of reuse artifacts as proposed by the Quality Improvement Paradigm (QIP) do not only require tool support for mere storage and retrieval. Rather, an integrated management of (potentially reusable) experience data as well as project-related data is needed. This paper presents an approach exploiting object-relational database technology to implement QIP-driven reuse repositories. Requirements, concepts, and implementational aspects are discussed and illustrated through a running example, namely the reuse and continuous improvement of SDL patterns for developing distributed systems. Our system is designed to support all phases of a reuse process and the accompanying improvement cycle by providing adequate functionality. Its implementation is based on object-relational database technology along with an infrastructure well suited for these purposes.

1. Introduction

Learning from experience gained in past projects is seen as a promising way to improve software quality in upcoming projects. As a result, (anti-)patterns, frameworks, and code fragments are being developed to capture the gained experience of software already developed. Recently, SDL patterns [24] have been introduced to increase reusability for developing distributed systems. They can be considered a representative example of such reuse artifacts. But experience is not only represented in the form of (directly) reusable software artifacts. To allow comprehensive reuse, as, for instance, proposed in [4], a large variety of different reusable elements exists. Process descriptions, for example, are used to precisely describe how to develop software in a certain domain, or lessons learned [5] can be used to store qualitative experience. Consequently, every kind of (software engineering) experience, independent of its type of documentation, is regarded as *experience element* in this paper.

However, the benefits that can be achieved by reusing such experience elements strongly depend on their quality. Thus, they always have to represent the latest state of the art. Hence, checking and continuously improving their quality becomes a crucial issue. The Quality Improvement Paradigm (QIP) [2] as suggested by Basili et. al. deals with this problem by integrating systematic evolution and comprehensive reuse of experience elements into an improvement cycle. Within this context, each project can be regarded as an experiment, focusing not only on the developed software products, but also on learning about and improving the applied experience elements. A QIP cycle consists of six steps, with steps 1–4 dealing with the planning and executing of an experiment, step 5 with analyzing its results, and step 6 with packaging the gained experience for later reuse. To store experience elements and offer them, on demand, to the (re-)user, a reuse repository called the Experience Base (EB) [2] is introduced. Within the EB, experience is only stored after it has been carefully analyzed (QIP step 5) and packaged (QIP step 6) for reuse. However, information concerning the experiments in which these experiences were gained is not directly inte-

*Now working at: TrueScope Technologies, Inc., 22 East Chicago Ave. Suite 230, Naperville, IL 60540, USA grated into the EB. Therefore, we developed a conceptual extension of the EB [11] that allows for storing complete experiment documentations, structured in accordance with QIP steps 1–4. Our resulting repository structure consists of two logically disjunct sections called Organization-Wide Section (OWS) and Experiment-Specific Section (ESS), where the OWS is an instantiation of Basili's EB and the ESS holds the experiment documentations.

To evaluate the usefulness of the conceptual extension, we implemented a web-based prototype of the reuse repository and applied it to the SDL-pattern approach [11]. Based on the positive results we now concentrate on the technological advancements for such QIP-driven reuse repositories. Thus, besides discussing the organizational repository structure and the interface functions required to provide comprehensive support, this paper demonstrates that the infrastructure provided by new Object-Relational DataBase Management Systems (ORDBMSs) can effectively be used for realization purposes. It is, for instance, possible to provide data to tools as files stored within an experiment-specific file-system area called Working Area (WA) and link these files to database entries in a way allowing the ORDBMS to control access and maintain consistency. As far as we know, our approach is the first one evaluating *object-relational* database technology in the field of reuse repositories. Fig. 1 illustrates how the ORDBMS-based repository is supposed to be used in the different phases of the QIP-based improvement cycle.

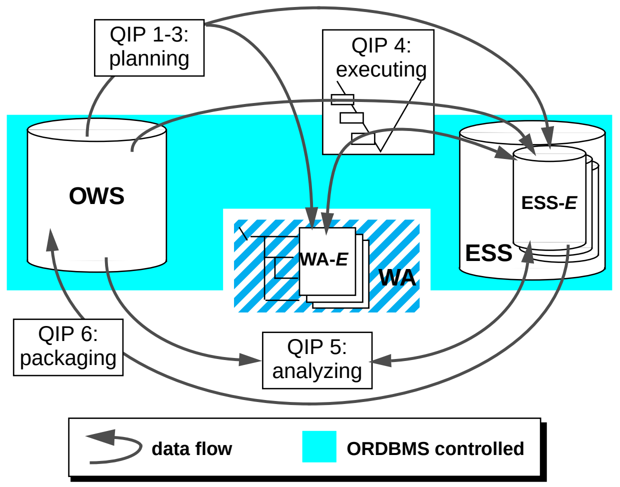

Figure 1. Exploiting the repository in the QIP cycle

This paper is organized as follows: In Section 2, the SDL-pattern approach is outlined. It serves as a representative, running example for discussing and illustrating requirements for an advanced reuse repository supporting a QIP-based improvement cycle. Section 3 outlines our implementation approach by emphasizing how ORDBMS infrastructure can effectively be exploited to realize the required functionality. Related work is discussed in Section 4. Finally, we summarize the results in Section 5.

2. Continuous Improvement of SDL Patterns

The formal description technique SDL [16] is in widespread use in industry as a design language for reactive, distributed systems. Recently, SDL patterns have been introduced as a new concept for increasing reusability in SDL-based system design [10]. In the following, we sketch the basic SDL-pattern approach and discuss its integration with continuous quality improvement. Resulting requirements on data management are discussed subsequently.

2.1. The Basic SDL-Pattern Approach

The basic SDL-pattern approach extends pattern-based reuse for engineering distributed SDL systems. It comprises a product model of reuse artifacts and an incremental, use-case driven design process.

SDL Patterns. Scattered parts of a given SDL design may together provide a certain functionality. By analysis, abstraction, and documentation, such a design solution can be reused whenever the design problem arises again. Roughly speaking, an SDL pattern is defined as a reusable software artifact representing a generic solution for a recurring design problem with SDL as the applied design language.

Design reuse as in the case of patterns is more flexible than code reuse, but the learning curve required before a pattern can be reused could be very high. It is therefore important to keep the specification of SDL patterns precise but intelligible. For this purpose, a standard description template is defined [10]. Physically, SDL patterns can be stored in different formats, e. g., as FrameMaker documents, in HTML, or in Postscript.

In order to support a stepwise and systematic evolution of SDL patterns, we defined different maturity levels [8] that allow developers to participate in their improvement, while reusing premature artifacts. This allows an incremental investment in reuse and contributes to quality, because good artifacts depend on practical and long-term experience. It can generally be stated that the effort in reusing an artifact decreases with higher maturity levels. In other words, the higher the previous investment in reuse, the higher the benefits.

Reuse Process. Generally, an SDL system development process encompasses the following development phases: object-oriented analysis, SDL-based design, formal validation, and, finally, code generation. In [24] we present an incremental, use-case driven design process tailored to SDL patterns (Fig. 2). First, system requirements are decomposed into single functionalities (the use cases), which are further analyzed and then implemented one after the other in separate development steps. Analysis includes developing (and improving) an outline system architecture, finding and exploring collaborations that refine the use cases, and breaking them down into smaller patterns (pattern selection). Detailed SDL design is done incrementally by incorporating the identified collaborations step by step. Each development cycle applies predefined SDL patterns for implementing the current collaboration or develops ad hoc solutions, if necessary. Note that application of an SDL pattern means adaptation and subsequent composition with the embedding context specification according to its application rules. The result of each development step is an executable SDL design specification, which is validated before entering the next step. For implementing the final SDL specification, several SDL development environments provide automatic code generators.

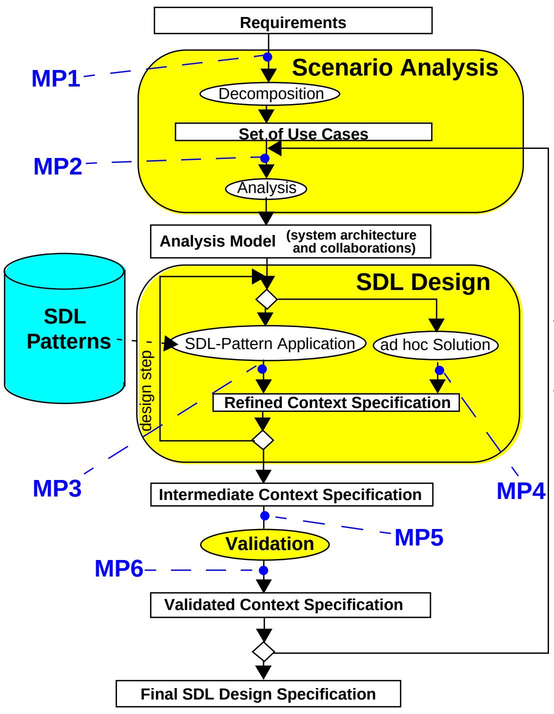

Figure 2. Measuring the reuse process

2.2. SDL-Pattern Based Improvement Cycle

Continuous quality improvement as applied to the SDL-pattern approach supports incremental evolution of reuse artifacts, which is essentially driven by practical experience and triggered upon user demands. Fig. 3 shows a graphical representation of the improvement cycle (bold lines) and its interaction with the reuse process (thin lines). We illustrate this interaction by means of usage scenarios that follow the QIP steps.

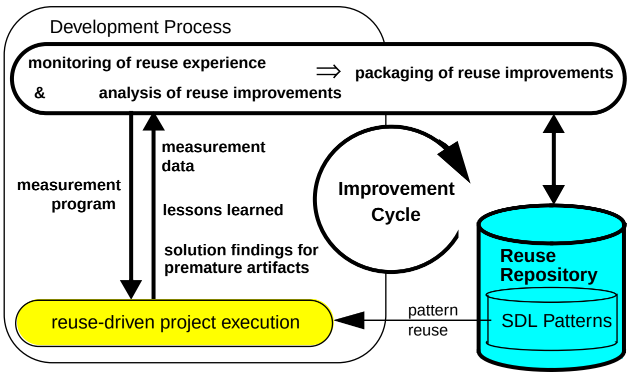
Development Process
monitoring of reuse experience
& analysis of reuse improvements
⇒ packaging of reuse improvements
measurement program
measurement data
lessons learned
solution findings for premature artifacts
Improvement Cycle
Reuse Repository
SDL Patterns
pattern reuse
reuse-driven project execution

Figure 3. Reuse process

Planning a new project (QIP steps 1-3). When setting up a new SDL-pattern based project, a new project database and working area is instantiated, where all kinds of documents coming up during the project can be stored. The first entry is the characterization of the project, including a description of its goals and possible time restrictions. With this information at hand, we can browse other SDL-pattern projects stored in the repository. This guides the project manager in defining a concrete project plan and determining necessary resources. Furthermore, the specific goals of the project (including learning-specific goals) must be fixed quantitatively according to the GQM paradigm [3]. This results in one or more GQM plans, which are integrated by a measurement plan. The measurement plan defines measurement points when different metrics are to be collected. For data collection, questionnaires are developed that must be filled out by the development team during project execution. Fig. 2 shows example measurement points MP1–MP6 as used in one of our case studies. For some standard learning goals, for instance, monitoring the quality of applied patterns, predefined measurement points and questionnaires are supplied in the repository. All documents prepared in this step are deposited in the project database of the repository for later access. This serves as documentation of the project history, which is decisive for analyzing and interpreting the measurement results in QIP step 6. Finally, it must be checked if needed tools (e. g., FrameMaker) and training material (e. g., an SDL tutorial) are ready to use. SDL tools that may be applied are, for instance, SDT (SDL design tool) [29] and the pattern-specific tool components SPEAR (SDL-pattern pool administration tool) and SPEEDI (SDL-pattern editor) [7].

Executing the project (QIP step 4). The project is executed according to the project plan, which is an instantiation of the incremental SDL-pattern based reuse process. Tools that can be applied for scenario analysis are, for instance, the UML [25] and MSC [17] editors of the SDT tool set. During SDL design, the SDL-pattern pool is searched for suitable patterns. This is supported by the SPEAR tool, which offers a search facility and links patterns with other related ones. Patterns are normally represented in HTML. However, SPEAR can also generate the internal pattern representation needed by the SDL-pattern editor SPEEDI. For SDT, pattern-based design is not supported. That means, when using the SDL editor of the SDT tool set –instead of SPEEDI–, the patterns must be inserted manually. However, SDT offers other advantages, such as tool-supported validation of an SDL design. During project execution, the questionnaires are filled out according to the predefined measurement program and, in addition, project-specific lessons learned are determined by the developers. The products produced, such as the SDL design specifications, are added to the project database of the repository.

Analyzing the project data (QIP step 5). The data collected during project execution is now processed and analyzed. This comprises the quantitative measurement data as well as textually fixed lessons learned. In order to get the right interpretation, special events or exceptions that occurred during project execution must be considered. This is made possible by the project trace stored in the project database of the repository.

Due to the detailed process and product models of the SDL-pattern approach, project analysis according to the pattern-specific learning goals proceeds straightforward. As illustrated in [8], the outputs from the measurement program capture many areas of possible improvements, such as increasing the maturity level of an existing pattern, so that “ad hoc” analysis can often be avoided. Another example is the detection of a new pattern event that may result in a proposal for a new SDL pattern.

Before the analysis results can be deposited in the reuse repository, they must be compared and related to previous experience. Only if the suggested improvements turn out to be general enough, can they be stored in the repository as common experience. Otherwise, they keep their preliminary status, but may serve as recommendation for planning similar future projects.

Packaging of analysis results (QIP step 6). To allow effective storage, search, and retrieval of experience data in the repository, a unique representation of similar experience is necessary. Therefore, the analyzed project data is stored according to predefined templates and/or style guides. General lessons learned, for example, are best described according to the template defined in [5]. SDL patterns are represented according to their standard description template. Support for modifying the description of an existing SDL pattern or extending the pattern pool is another functionality of the SPEAR tool. Modifying an existing pattern may result in upgrading its maturity level. When adding a new pattern, it has to be decided which maturity level to start with. Additionally, new patterns must be linked with related patterns of the pool.

2.3. Requirements on Data Management

The discussion in Section 2.2. implicitly addressed various requirements pertaining to data modeling as well as data manipulation. For the sake of clarity, we discuss only the most important requirements.

Data Modeling Needs

R-I: Integrated management of experience data and project-related data. For the purpose of maintaining consistency and avoiding redundancy, it is reasonable to manage experience elements and experiment-specific data (project data) in an integrated manner (within the same database). For example, after having identified an SDL pattern as a reuse candidate, it is generally helpful to have hints to projects in which this particular experience element has been applied before. Furthermore, integrated management is advantageous for the QIP steps analyzing and packaging, because it supports identification of new and unique experience.

R-II: Semantic classification of experience elements. As discussed before, the database area storing experiment elements (in Section 1 identified as the organization-wide section, OWS) must be able to manage semantically different data structures. For example, structures for storing SDL patterns, measurement program elements, lessons learned, and so forth. Similarly, the experiment-specific section (ESS) must also be able to manage such data structures in order to handle project-related data.

R-III: Relationships between experience elements. The experience elements managed in the above-mentioned classification must be connected by various semantic relationships carrying crucial information for the analyzing and packaging steps of the QIP. For raising an SDL pattern's maturity level, for example, the pattern engineer needs to retrieve all information concerning occurrences of this pattern in former projects. Therefore, a `uses/used_in` relationship between the SDL pattern and former projects is needed.

R-IV: Multiple representations of experience elements. We learned that different tools need SDL artifacts in different formats, e. g., HTML, text, proprietary tool formats, etc. This concerns both, OWS and ESS. Since this is a very typical scenario in design applications, there must be data structures allowing to store design artifacts in different/alternative representations.

R-V: Management units for project-related data. The repository has to support several projects simultaneously. Since project work usually results in different release levels of the products created (private data, data released for the project team, data released for public access), private as well as project-related workspaces must be managed. This requirement has already been addressed in Fig. 1 showing the project-specific areas ESS-E and WA-E.

R-VI: Maintaining a project history. Within the context of a single project, different objects have to be associated with a (project) history, which is a crucial prerequisite for the step of analyzing design data. For example, the possibility of tracking the project history often helps to validate and generalize the lessons learned (QIP step 5 in Section 2.2.).

Data Manipulation Needs

R-VII: Role and authorization concept. Prerequisites for data manipulations are an adequate role concept and corresponding authorization mechanisms. The previous discussion already indicated that different roles for managing the OWS (e. g., the SDL pattern engineer), leading a project, working within a project team as a designer or as a quality manager, are needed. The different roles are to be associated with different areas of visibility w. r. t. the data stored. Preliminary data of a project, for example, should not be visible to members of a different project's team.

R-VIII: User interface. The different tasks associated with different roles require corresponding functions to be provided at the repository interface. Generally, functions are needed for browsing repository data, traversing object structures, searching for artifacts, and manipulating the data stored within the repository. For example, the person responsible for managing experience data has to be supported by functions allowing to access the complete OWS. This includes retrieval and manipulation functions. The latter must not be provided to unauthorized personnel.

R-IX: Distributed access. Regarding the frequently occurring situation that teams are geographically dispersed, and the cooperative working style needed in software development projects, it must be possible to contact the repository from different nodes of a distributed environment.

R-X: Tool integration. Besides a (specially designed) interface providing different roles with corresponding access functions, an infrastructure allowing tools to access the repository must be provided. For example, the SDL-pattern based reuse cycle employs applications of the tools SDT, SPEEDI, and SPEAR. Where own implementations like SPEEDI and SPEAR can use the repository's programming interface, commercial tools like SDT require flexible coupling mechanisms for their integration.

3. An ORDBMS-based Approach

Since ORDBMS [26] are not only the most recent but also the most promising trend in database technology, we decided to evaluate their potential w. r. t. implementing our reuse repostory. As this paper will show, we found the extensibility property of ORDBMS to be extremely helpful, and, therefore, ORDBMS to be well suited for our purposes. First, we will describe the data structures resulting from the data modeling needs (see previous section). Afterwards, data manipulation needs are elaborated.

Data Structures

Representations of *experience elements* (EE) are given in many different data formats (**R-IV**). To easily handle the different formats (and to be open for new formats), EE representations should only be saved using a plain data type in the repository without respect to special data formats. ORDBMS offer a special kind of data type for this purpose, called BLOB (binary, large object). A representation can be composed of several parts. For example, an HTML document (e. g., a framed HTML page) can consist of many files. Therefore, each representation is stored in a set of BLOBs. Also, several alternative representations of the same EE can occur. Thus, an EE has an internal structure as illustrated in Fig. 4.

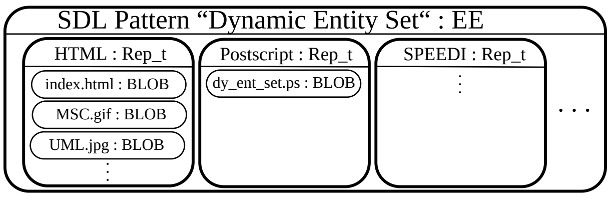

Figure 4. Structure of an experience element

For retrieval purposes, the actual EEs need to be associated with a so-called *characterization vector* (CV) containing describing data, e. g., relevant for (similarity-based) search of potentially reusable design artifacts. For performance reasons, EEs (containing large objects) and corresponding CVs (comparably small amount of data) are stored separately. In Fig. 5 the schema of the reuse repository is shown in (an intentionally incomplete) UML notation. It illustrates the separation of EEs and CVs and shows that each EE is associated with exactly one CV.

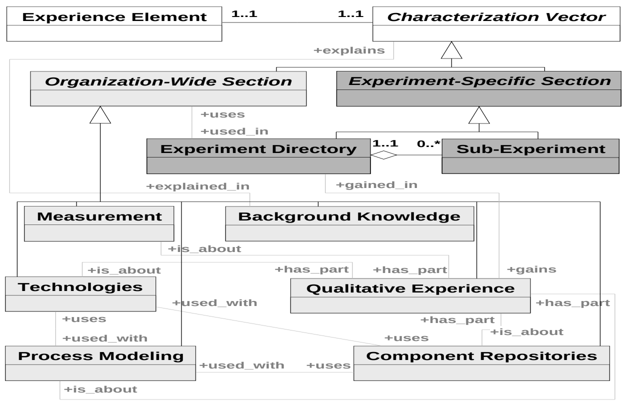

Figure 5. UML schema of the reuse repository (excerpt)

Whereas all EEs are stored using the same data structure, the CVs are classified into semantically different data structures (**R-II).** CV attributes depend on the section (OWS, ESS) that the corresponding EE belongs to. More precisely, the sections are divided into logical areas, and the areas determine the CV attributes. Fig. 5 illustrates the inheritance hierarchy. Sub-classes of the abstract class `Organization-Wide Section` have special attributes of the different kinds of experience elements. For example, `Qualitative Experience` contains the inspection techniques used, whereas `Process Modeling` contains valid process models. Note that some CV attribute values can be determined automatically (e. g., author, creation date, etc.) while others have to be specified manually.

The (abstract) class `Experiment-Specific Section` has the sub-classes `Experiment Directory` and `Sub-Experiment`. Remember that the ESS stores the data of running and completed experiments (**R-I**). Each experiment consists of one `Experiment Directory` object and many `Sub-Experiment` objects (**R-V**). The general data of an experiment like project-manager notes, project type etc. are stored in the `Experiment Directory`, while the data of modules, measurements, reports etc. are stored in the `Sub-Experiment` objects. Fig. 5 shows the corresponding aggregation between these two classes. Besides other attributes, these classes contain attributes allowing to capture the project history, e. g., version identificators and timestamps (**R-VI**). Fig. 5 also shows many associations between the different `CharacterizationVector` subclasses denoting semantic dependencies (**R-III**).

Obviously, the different CV classes are arranged within an inheritance hierarchy. While RDBMS do not support table hierarchies, ORDBMS do. Therefore, the UML classes depicted in Fig. 5 can easily be mapped to a table hierarchy in the ORDBMS schema. Some CV attributes can be mapped to the common data types of an RDBMS (like String or Integer), but we also use complex data types (like multisets) and user-defined types (UDT) as supported by ORDBMS.

Users, Roles, and Authorization

Role Management. Since our approach aims at supporting the overall improvement cycle, all persons involved in this process have to be provided with adequate functionality. Whether or not an interface function (see below) is available to a user, is determined by his current role. To fulfill **R-VII**, the following roles are defined in our repository:*

- *Repository Manager* (managing OWS data);
- *Repository Assistant* (supporting the Repository Manager);
- *Project Manager* (managing experiment-specific parts of the ESS);
- *Quality Manager* (assuring quality of the development process and corresponding products);
- *Developer* (attending an experiment).

Furthermore, the role contributes to determining which data is visible, may be modified, or even deleted within a user session. Regarding running projects, however, the role is not sufficient to determine corresponding rights (and duties). Thus, a Project Manager's possibilities also depend on the project's name. In order to control the data access of a (project) team member, besides the role, the system needs to know the project's and the developer's name.

Authorization. The autorization component contributes to fulfilling **R-VII** by checking security at the beginning of user sessions and by providing an infrastructure for defining and maintaining application-specific access rights.

A User Interface supporting the QIP steps

Planning a new project (QIP steps 1-3). A new project is set up by *creating a project-specific data area* ESS-E, initially taking up project describing data, e. g., description of the project goals. A corresponding function is provided at the repository interface.

Now, facing the project goals, the Project Manager starts investigating the OWS in order to identify reusable EEs. This step is supported by *browsing and navigation* functions, and, additionally, by a specially

* For the sake of clarity, we do not consider more roles than those listed here. Taking additional roles into account would not require additional concepts. In either case, besides the mentioned roles, a *DB Administrator* is needed.

designed *search* function. In the planning step, descriptions of similar, past projects and corresponding training material are usually looked for (Section 2.2.).
When running earlier (not DBMS-based) repository prototypes [11], it already turned out that this retrieval operation should be *similarity-based*. Thus, the system provides datatype-specific measures and similarity functions, which may be adapted to the semantics of attributes (associated with CVs). An example will be given later on in this section.
After having identified a reusable EE, (e. g., the SDL reuse process description), the Project Manager now uses a further interface function for *transferring* (copying) this EE from the OWS to ESS-E. This function is designed to automatically add a `uses/used_in` relation between the original EE (in the OWS) and its copy (in the ESS-E).

Executing the project (QIP step 4). As a default, ESS-E data are not visible to anyone but the project team consisting of the Project Manager, a Quality Manager, and a number of Developers. The data initially transferred from the OWS to the ESS-E (results of the planning step) may be read by all team members. Developers have private working areas (WA-E), in which preliminary results are to be managed. Developers may grant team members or the overall team read and/or write access to private data. Write access does not comprise deletion; thus the Developer who initially created the data remains the owner of these data and, consequently, may withdraw the rights he previously granted.

The ESS-E contains data entries that may be exclusively accessed by the Project Manager or the Quality Manager, respectively. In cooperation with Project Manager and affected Developer, the Quality Manager may release ESS-E data to public. This means, data which has reached a certain stability (w. r. t. product quality) and, therefore, may be beneficial for others, can be made visible. As soon as the Repository Manager acknowledges this step, the corresponding data acquire the (default) status of OWS data (regarding visibility, see below). If necessary, the quality manager can withdraw this release, as long as the corresponding experiment is active.

This short description of the executing step clarifies that functions for *browsing, navigating, searching, manipulating, and releasing* ESS-E data must be provided at the repository. The challenge in developing these functions is to appropriately manage and ensure the respective persons´ rights during function evaluations.

Naturally, during the executing step, OWS data can also be further investigated in order to react to the current project state. Actually, all functions described in the previous subsection can also be applied in the executing step. It might, for example, be reasonable to query the OWS for certain SDL patterns during the executing step, because corresponding needs may evolve during project work.

Analyzing the project data (QIP step 5). After the experiment has been completed, the Quality Manager uses *browsing, navigation, and search* functionality to consider ESS-E data and can perform *manipulations* in order to fix or increase the reuse potential. Additionally, functions for *browsing, navigating, and searching* OWS data are available, helping to identify new, non-redundant experience data (Section 2.2.). An intuitive example for actions to be performed during this step is determining or increasing the maturity level of an SDL pattern. The Quality Manager uses another interface function to *mark* those parts of the experiment-specific data that should be provided to the public as gained experience.

Packaging of analysis results (QIP step 6). In this final QIP step, the Repository Manager can transfer the marked data into appropriate OWS structures in order to commit the final release. Furthermore, he (semantically) integrates these new experience data by establishing relations, e. g., `gains/gained_in` relations, among new entries as well as among new and previously integrated data (Section 2.2.). To do so, he applies a couple of interface functions that are at his disposal not only to specifically support the packaging step, but also to support him in generally managing OWS data.

As a default, all users may read OWS data, but the Repository Manager has the possibility of restricting read access by explicitly detracting certain EEs (to be determined by a predicate on the corresponding CV type) from the visibility of certain users or roles. Furthermore, also as a default, only the Repository Manager is allowed to insert, modify or delete OWS data; but, again, there are possibilities for delegating work, e. g., packaging work. Thus, the Repository Manager may explicitly grant modification (including deletion) rights on certain EEs as well as the right to insert data into the OWS to Repository Assistants (e. g., the SDL pattern engineer). Naturally, he may withdraw these rights as soon as reasonable from his point of view.

Obviously, again, *retrieval and manipulation* functions are needed that observe user rights. The function for *inserting* or *manipulating* OWS data, for example, must not be provided to anyone but the Repository Manager or, if specifically authorized, to Repository Assistants.

Realization aspects and example. In order to further demonstrate the ORDBMS-based approach, we now give an example illustrating several of the aspects mentioned so far and emphasizing the benefits of ORDBMS for that purpose.

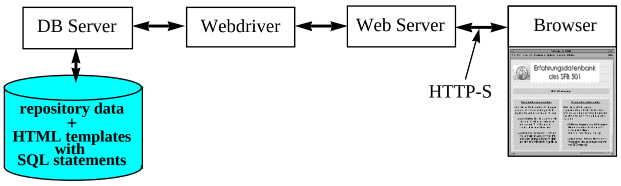

Figure 6. Architectural overview

In order to fulfill **R-IX**, we decided to exploit the web infrastructure provided by ORDBMS to realize the interface functions. As realization platform, we exploited the ORDBMS Informix IDS/UDO [15] including Informix WebBlade as web infrastructure. The architecture of our system is shown in Fig. 6.

The database stores EEs, CVs, pre-defined HTML pages as well as extensions (special tables, user-defined functions (UDFs)) needed to dynamically generate HTML pages and answer user requests. In order to answer special user requests, HTML templates including SQL statements can be stored. A user request that has been specified at the browser and passed to the DB Server may address such an HTML template. The UDF *webexplode* (offered by the WebBlade) evaluates the SQL statements contained in the template, incorporates the results into the template and sends it back to the browser. As we will see in the example below, there is also the possibility of adapting or even generating the SQL statement(s) to be evaluated in order to create the resulting HTML page dynamically.

In order to demonstrate the approach, we have chosen the similarity-based search as an example. After the user has started a user session by authenticating himself, he may start working at the repository interface by invoking the functions outlined in the previous section. For security purposes, we use the net protocol HTTP-S. Since all HTTP protocols are context-free, the current session context, represented by an authentication number, is communicated each time a user request or a resulting HTML document is transferred. An authentication number is generated each time a message is sent for safety reasons.

The authorization component of our repository consists of a couple of database relations and corresponding UDFs checking authentication and generating authentication numbers. Whereas this is sufficient to prohibit unauthorized access to the reuse repository, more fine-granular access control mechanisms are needed to control database queries, like similarity-based search. It is necessary to control access to tuples (e. g., a single experience element) as well as attributes (e. g., notes of the Repository Manager). For these purposes, again, the DBMS has been extended by some management tables and corresponding UDFs.

In order to generate a mask for the similarity-based search, a UDF is used expecting the current role as input parameter, and delivering a list of all accessible CV attributes as output. After the user has specified the comparison instance, the request is sent to the server and there transformed into one or more SQL queries, making heavy use of predefined UDFs basically observing rights and computing similarities. Such a generated SQL statement may look as follows.

```
SELECT  name,(SIM(validity,5,1,1,1)*10+SIM(impl_technique,“SDL/MSC“,...)*3+...)/TW AS Total_Sim
FROM    Component_Repositories  WHERE Total_Sim > 0.5 AND RightsOK(:RID, ID)
```

The UDF `RightsOK()` ensures that only tuples are taken into account that are enabled for the user/role. The similarity of an EE is computed from the similarities of the regarded CV attributes (validity, impl_technique, etc.) and depends on the weight of these attributes. The weight of each attribute can be specified in the mask for the similarity-based search. In order to facilitate the use of the reuse repository, default values are given, but each user may store his/her own default values. Since similarity values are between 0 and 1.0, the sum of attribute similarities has to be divided by the total weight of all attributes (TW). In the query above, only experience elements with a similarity of at least 0.5 are taken into account.

The UDF `SIM()` represents the similarity function associated with a given attribute. For each data type a special similarity function is needed, so the `SIM()` function is overloaded. The function is equipped with parameters to describe their characteristics, in order to allow users to bring in individual notions. If the user did not specify individual default values previously, the system uses default values.

Obviously, the SQL query generated (and the resulting similarity values) to some extent depend on the current user, resp. the user-specific values.

Tool Integration

In order to support the overall improvement cycle, in addition to the functions mentioned above, an infrastructure for coupling tools, e. g., those mentioned in Section 2.2. (SDT, SPEEDI, SPEAR), with the repository is needed. We aim at supporting several tool integration modes:

- *White-box integrated* tools directly use the ORDBMS API for data access.
- Some available tools are designed to be connected to data sources via standardized *communication infrastructures* [27], as ODBC/JDBC [9] or CORBA [22].
- Another possibility is *wrapping* data access operations of a tool. A representative of this integration mode is the DataLinks concept [20, 21]. It allows to redirect data access operations (to the database) in a way, which does not influence to tool implementation at all.
- Last but not least, *manual control* is an integration mode that is easy to realize. Here, the user gets checkout functionality to (lock and) copy database data into the file system. Afterwards the tool can be applied and its results can be (manually) propagated (checkin) by the user. Thus, the functionality needed is to some extent comparable to the unix tool RCS [31].

The different integration modes imply different possibilities of transaction control. For example, the white-box integration suggests exploiting the transaction concept supported by the ORDBMS. This means that a tool application corresponds with a database transaction. Manual control, on the other hand, basically means that only ckeckout/checkin are mapped to short database transactions and that the tool's application is not protected at all, unless the user is made responsible for taking care. Currently, we follow the approach that, with the exception of white-box integration all integration modes are mapped to several short database transactions, respectively. We intend to realize an application server on top of the ORDBMS, allowing to handle versions of EEs appropriately and to control concurrent access to these versions as well.

Supporting the various integration modes discussed in this section leads to a very flexible infrastructure (**R-X**).

4. Related Work

As stated in [19] the subject of storage and retrieval methods of software assets, although under investigation for nearly two decades now, is still an active area of research, because new technologies keep opening new opportunities. Thus, our approach focuses on exploiting new object-relational database technology, which has not been examined for that purpose so far. Consequently, Section 4.2. deals with relating our work to other work from the technological perspective. Regarding the methods used, our approach fits into several categories introduced in [19], descriptive methods (associating CVs with EEs), topological methods (similarity-based search), and structural methods (mapping EE structures to DB schema structures). Structuring EEs, which is another crucial issue in our approach, is considered in the following subsection. Due to space restrictions, a comparison with AI systems (e. g., Case-Based Reasoning (CBR) systems [1]), wich could be used to support the intelligent serach of EEs, is beyond the scope of this paper.

4.1. Repository Structures

In [13,14] Henninger discusses a ***repository for reusable software components***. The paper focuses on the indexing structure of such a repository. A method is proposed supporting index implementation "*with a minimal up-front structuring effort*". Therefore, a rudimental set of reusable components is stored in the repository with a simple, basic index structure to get the reuse activities started. Then, while the components are being reused, the index structure is incrementally improved. The idea of starting the reuse process in a state of incompleteness to gain practical results as the basis for incremental improvement is similar to our approach. The information stored in our repository becomes the more detailed, the more EEs are being reused, caused by the growing number of relations (e. g., the `uses/used_in` relation) between the EEs. However, no predefined relation structure between the components is offered, which is an essential part of our repository structure. On the other hand, our repository is currently missing the ability of direct measures concerning the usage of the stored EEs in the repository that is an integral part of Henninger's repository. These measures would help us, for instance, in defining the maturity levels of the SDL patterns, and therefore, we plan to integrate them in our repository in the near future.

4.2. Technology Exploitation

The ***ASSET Reuse Library WSRD*** [30] is an example of a web-based implementation of an object repository. It is a domain-oriented reuse repository where a certain domain (or collection) can be compared to an experiment documentation of our experiment-specific section. However, complete project documentations are not an integral part of WSRD. Cross-references that interrelate the entries in the collections are offered, but they are much more general and unstructured than the relations defined in our repository.

In the Arcadia project [28] several object management systems [12] have been developed to support their process-centered software engineering environment. Triton [12] is one of those object managers. The whole system is based on the Exodus [6] database system toolkit. Heimbigner describes Triton in [12] as follows: "*It is a serverized repository providing persistent storage for typed objects, plus functions for manipulating those objects*". Whereas we, on demand, link the stored EEs from the ORDBMS to the file system, so that different (commercial) tools can use them, Triton uses Remote Procedure Calls (RPC) for communication between client programs (tools) and the repository. But this is certainly a limitation, since most of the (commercial) tools do not offer the needed functionality to use RPC's and can not be modified because their source code is mostly not available. Mediators as described in [32] can be used to bridge the gap between a repository like Triton and such tools. But this would call for a wide variety of mediators. To avoid this construction effort, we consider EEs as BLOBs to keep the original storage formats of the different tools. The Triton prototype, on the other hand, uses a homogeneous storage schema to provide efficient representation for the wide variety of software artifacts. Consequently, all data that is shared by two or more tools have to be converted to the Triton schema, even if a direct data exchange via a format like RTF is available. Again, this is acceptable in an environment where most of the tools have been developed from scratch, but it is not suitable for an environment where commercial tools are heavily in use.

5. Conclusions

Exploiting ORDBMS features, we succeeded in conceptualizing and realizing a reuse repository that fulfills the major requirements of software development processes following the Quality Improvement Paradigm. We described the database schema, the interface functions, the flexible possibilities of integrating tools, and outlined the software architecture of our approach. To the knowledge of the authors, this is the first approach based on ORDBMS technology for such a kind of repository.

Although we consider ORDBMS to be the best choice w. r. t. our purposes, we have to admit that there are still open questions. One important requirement of software development applications is the support of an adequate versioning model for resulting products and experience elements. Current ORDBMS do not support versions at all. Consequently, we plan to realize an appropriate version model on top of the

ORDBMS by exploiting its extensibility infrastructure. First steps into this direction are described in [18, 23].

References

[1] A. Aamodt, E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. In AICom - Artificial Intelligence Communications, 7(1):39-59, March 1994.

[2] V. R. Basili, G. Caldiera, H. D. Rombach. Experience Factory. In J. J. Marciniak (ed), *Encyclopedia of Software Engineering, Volume 1*, John Wiley & Sons, 1994, 469–476.

[3] V. R. Basili, G. Caldiera, H. D. Rombach. Goal Question Metric Paradigm. In J. J. Marciniak (ed), *Encyclopedia of Software Engineering, Volume 1*, John Wiley & Sons, 1994, 528–532.

[4] V. R. Basili, H. D. Rombach. Support for comprehensive reuse. In *IEE Software Engineering Journal,* 6(5):303–316, September 1991.

[5] A. Birk, C. Tautz. Knowledge Management of Software Engineering Lessons Learned. In *Proc. of the 10th Int. Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco, CA, June 1998.

[6] M. Carey, D. Dewitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, S. Vandenberg. The EXODUS Extensible DBMS Project: an Overview. In S. Zdonik, D. Maier (ed), *Readings in Object-Oriented Databases*, Morgan Kaufman, 1990.

[7] D. Cisowski, B. Geppert, F. Rößler, M. Schwaiger. Tool Support for SDL Patterns. In *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM98)*, Berlin, Gemany, 1998, 107–115. ISSN: 0863-095.

[8] R. L. Feldmann, B. Geppert, F. Rößler. Continuous Improvement of Reuse-Driven SDL System Development. In *Proc. of the 11th Int. Conference on Software Engineering and Knowledge Engineering, (SEKE'99)*, Kaiserslautern, Germany, June 1999.

[9] K. Geiger. Inside ODBC. Microsoft Press, Redmond, Washington, 1995.

[10] B. Geppert, R. Gotzhein, F. Rößler. Configuring Communication Protocols Using SDL Patterns. In A. Cavalli, A. Sarma (ed), *SDL'97 - Time for Testing*, Elsevier Science Publishers, Proc. of the 8th SDL Forum, SDL '97, Paris/Evry, France, September 1997.

[11] B. Geppert, F. Rößler, R. L. Feldmann, S. Vorwieger. Combining SDL Patterns with Continuous Quality Improvement: An Experience Factory Tailored to SDL Patterns. In *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM98)*, Berlin, Gemany, 1998, 97–106. ISSN: 0863-095.

[12] D. Heimbigner. Experiences with an object manager for a process-centered environment. In *Proce. of the 18th VLDB Conference*, Vancouver, British Columbia, Canada, August 1992.

[13] S. Henninger. Supporting the Construction and Evolution of Component Repositories. In *Proc. of the Eighteenth Int. Conference on Software Engineering*, IEEE Computer Society Press, March 1996, 279–288.

[14] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. In *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, April 1997.

[15] Informix Universal Server Documentation, Informix Software Inc., http://www.informix.com/, 1997.

[16] ITU-T Recommendation Z.100 (03/93). CCITT Specification and Description Language (SDL). International Telecommunication Union (ITU), 1994.

[17] ITU-T Recommendation Z.120 (10/96). Message Sequence Chart (MSC). International Telecommunication Union (ITU), 1996.

[18] W. Mahnke, N. Ritter, H.-P. Steiert. Towards Generating Object-Relational Software Engineering Repositories. In *Proc. Datenbanken in Büro, Technik und Wissenschaft (BTW'99)*, Freiburg, Germany, March 1999.

[19] Mili, A., Mili, R., Mittermeir, R.T.: A survey of software reuse libraries. In: *Annals of Software Engineering* 5 (1998), 349-141.

[20] I. Narang, C. Mohan, K. Brannon. Coordinated Backup and Recovery between DBMSs and File Systems. Technical Report, IBM Almaden Research Center, 1996.

[21] I. Narang, R. Rees. DataLinks - Linkage of Database and File Systems. In *Proc. of the 6th Int. Workshop on High Performance Transaction Systems*, Asilomar, September, 1995.

[22] R. Orfali, D. Harkey. Client/Server Programming with Java and CORBA. Wiley Computer Publishing Group, New York, 1997.

[23] N. Ritter, H.-P. Steiert, W. Mahnke, R. L. Feldmann. An Object-Relational SE-Repository with Generated Services. In *Proc. of the 1999 Information Resources Management Association International Conference (IRMA99)*, Hershey, Pennsylvania, USA, May 1999.

[24] F. Rößler, B. Geppert, and P. Schaible. Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns. In *Proc. of the 5th IEEE Int. Conference on Software Reuse, ICSR5*, Victoria, BC, Canada, 1998

[25] J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[26] M. Stonebraker, M. Brown. Object-Relational DBMSs - Tracking the Next Great Wave. Morgan Kaufman, 1999.

[27] C. Szyperski. Component Software, Beyond Object-Oriented Programming. Addison-Wesley, Harlow, 1998.

[28] R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, M. Young. Foundations for the arcadia environment architecture. In P. Henderson (ed), *Proc. of the 3rd. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, November 1988. Appeared as ACM SIGSOFT Software Engineering Notes 13(5), November 1988.

[29] Telelogic. Tau 3.4 SDT User's Manuals. Telelogic, Sweden, 1998.

[30] The ASSET Reuse Library, http://www.asset.com/WSRD/indices/domains/REUSE_LIBRARY.html. Dec. 1998.

[31] W. Tichy. RCS – a system for version control. In Software-Practice and Experience 1985, 15(7):637–654.

[32] G. Wiederhold. Mediators in the architecture of future information systems. In *IEEE Computer*, 25(3):38–49, March 1992.