Exploiting Reuse Techniques in the SERUM Approach[1]

N. Ritter, H.-P. Steiert

Database & Information Systems Group
Department of Computer Science
University of Kaiserslautern
P O Box 3049, 67653 Kaiserslautern, Germany
e-mail: {ritter, steiert}@informatik.uni-kl.de

Abstract: Nowadays the complexity of design processes, no matter which design domain (CAD, software engineering, etc.) they belong to, requires system support by means of so-called repositories. Repositories help managing design artifacts by offering adequate storage and manipulation services. To enable the support of the wide variety of application needs, we propose a generative approach for providing repository manager functionality. For the process of customization of a preliminary specification of repository manager services and generating corresponding implementations, application of reuse techniques is crucial. This paper details our approach of generating repository manager functionality by emphasizing the reuse techniques applied and the objects actually reused.

Keywords: Repositories, Reuse, Generic Methods, Design Patterns, Frameworks, Software Engineering.

1. Introduction

The *Sonderforschungsbereich 501 'Development of Large Systems with Generic Methods' (SFB 501),* founded by the *Deutsche Forschungsgemeinschaft (DFG),* contains (among others) the (sub-)project *Supporting Software Engineering Processes by Object-Relational Database Technology (SENSOR)*. In this project we consider new object-relational database technology [8, 19] for software systems which have to support data management tasks in software engineering (SE) projects. This comprises both, managing product data related to a current system under development as well as managing process data, i. e., information related to the development process itself. Obviously, in large software development projects a shared database should be used in order to support cooperation of developers and reuse of design. Usually, a repository is

1. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501 "Development of Large Systems with Generic Methods".

more appropriate than a pure DBMS, because repositories enhance DBMS functionality by value-added services, as for example management of versioned product data [2].

Although our project is strongly focused to the use and extension of object-relational database management systems (ORDBMS), this paper does not discuss the technical aspects of using an ORDBMS as an enabling technology for building repositories [10]. Instead, we want to argue as follows.

In large software development projects, multifaceted requirements concerning software engineering repositories occur. The variety of tools used in the various phases of the software development process leads to a lot of different, sometimes contradictory, requirements. In our opinion it is hard, if possible at all, to fulfil all these different demands by providing a single 'stand-alone' repository system. Therefore, we have to build a family of systems, each providing functionality, which is customized for the needs of tools used in a particular subdomain.

Building a family of similar systems is a typical area for the application of reuse techniques. For this reason, reuse is the crucial issue in our approach, called SERUM[2]. Our main idea is to generate a repository system, which we call a repository manager, for a particular subdomain from a high-level specification. Starting with this (incomplete) specification, a sequence of transformation steps is performed leading to a specification, which is complete in the sense that it can serve as input for a generator. The SERUM approach defines this (transformation) process and supports the corresponding steps with suitable tools. The idea of reuse is put into practice in form of process reuse and reuse of building blocks. Process reuse is embodied by the SERUM process itself and SERUM tools. Building blocks are reused throughout the process for completing the specification and for composing/generating the repository manager.

This paper emphasizes reuse in SERUM. It is structured as follows. First, we introduce our domain, i. e. the construction of repositories for computer aided engineering, or, more precisely, computer aided software engineering applications (Sect. 2). Afterwards, Sect. 3 outlines the SERUM process for the customization of repository managers and introduces the methods, tools and techniques of our approach. In Sect. 4, the reuse techniques applied in the SERUM approach are compared w. r. t. classifications of reuse techniques, which can be found in the lit-

2. SERUM stands for: Software Engineering Repositories using UML.

erature, especially those presented in [12] and [9]. Furthermore (Sect. 5), we examine the reuse techniques in more detail and discuss the “objects of reuse”, especially the SERUM frameworks (Sect. 5.2), which provide the building blocks. Additionally, the SERUM repository generator (SRG) is explained. Sect. 6 concludes the paper.

2. The Domain

For years the term *repository* has been used in a restricted manner, since it only addressed meta-data management (descriptions of descriptions of things) in the context of database management systems. Nowadays it is used in a much broader sense and covers multiple services supporting design applications. In [2] the term ***repository*** is defined as *a shared database of information about engineering artifacts*. Thus, *a common repository allows* (design) *tools to share information so they can work together.* A corresponding ***repository manager*** *provides services for modelling, retrieving, and managing objects in a repository*. For that purpose, a repository manager has to provide the *standard amenities of a DBMS* (*data model, queries, views, integrity control, access control, and transactions*) as well as some value-added services [2]: *checkout/ checkin, version control, configuration control, notification, context management and workflow control*. Since these terms (as well as their respective meaning) are widely known, we do not detail them.

The requirements ***repository services*** have to fulfil vary w. r. t. different ***domains***. Our domain is *computer aided software engineering*, or, more exactly, *development of large (software) systems with generic methods*. This domain spans different ***domain sections***, as for example *management of versioned product data* or *designflow management*. While the former requires facilities for versioning complex structured objects, the latter may utilize “active” repository services. Hence, each domain section also has its own requirements pertaining dedicated repository services.

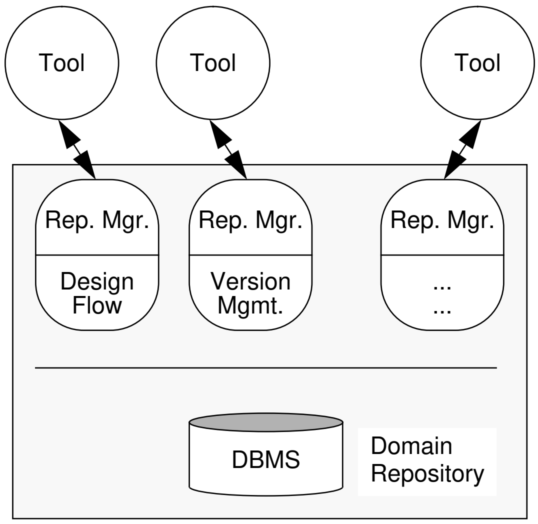

Fig 1: Domain Repository

Usually, repository manager services are as generic as database system services are. Let us consider versioning facilities as an example for generic services. Usually a repository manager implements a given versioning model offering a predefined set of data structures and generic operations. As already discussed in [6] there are very many facets, versioning models may differ in. As we think, many of the different concepts which lead to (lots of) different version models are very application-specific. In our domain the demands regarding a versioning model may differ between managing versions of product data and managing versions of design flow descriptions. Consequently, a generic version model cannot support *all* applications properly, but serves some more, some less appropriately. Our goal is to be able to support all applications by providing basic versioning facilities which may be refined and which are the foundation for generating application-specific functionality. We think that this approach is not only suited w. r. t. versioning, but also w. r. t. other fields of design domains, e. g. designflow management.

In order to cope with the needs of the different domain sections, we want to provide a *framework for building customized repository managers* instead of a single, universally applicable repository manager. We call this framework SERUM and use this term also as a name for the whole project and the approach behind the framework. Since SERUM is a framework and not a "stand-alone" system, it offers a set of methods, tools, techniques and "half-fabricated" components helping to create customized repository managers. The ***domain repository*** is established by a set of repository managers, one for each domain section. All repository managers are based upon the same (logically) centralized DBMS allowing for data sharing among domain sections (Fig. 1).

Actually, all repository managers share a core set of common properties, as for example the need for version management capabilities. Nevertheless, each repository manager has to be enhanced by some special properties which depend on the domain section. The SERUM approach aims at identifying the common features and providing reusable design solutions as well as corresponding source code units. Furthermore, we aim at providing a tool infrastructure, which helps to build up repository managers by exploiting reuse techniques. In the next section, we examine the steps necessary to build up a SERUM repository manager. Afterwards we identify the reuse techniques applied during this process.

3. The SERUM Approach

As part of our work in the SFB501 project we have to build repositories in order to support all steps of the software and hardware development process. We have already mentioned above that this means to build a family of repository managers, each providing customized access to the central database for a particular set of design tools. Because these repository managers have a lot of similarities, we do not want to build each of them from the scratch. Hence, we have to apply reuse techniques. In order to take advantage of them at different levels of abstraction, we have decomposed the process of building a repository manager. The process starts at a high-level of abstraction and for each step towards an executable system SERUM supports the application of reuse techniques. In this section, we give a brief overview of the SERUM process, before we examine the reuse techniques applied in more detail in Sect. 4 and Sect. 5.

Fig. 2 gives a graphical illustration of the SERUM process for customizing a repository manager. If a new repository manager needs to be build or an existing one must be modified, the ***repository designer***[3] first has to give (1) (or modify (2), respectively) a UML[4] specification of the (application-specific aspects of the) services to be provided by this repository manager[5].

Obviously, a transformation process cannot start from an arbitrary UML specification. Thus, some guidelines, which restrict the design space, have to be checked first (3). This check is especially necessary, if the repository designer started with a completely new specification (1) and did not select an existing one (2) in order to adapt it to his special needs. Guidelines are expressed in OCL[6]. If the specification does not fulfil the guidelines, then the repository designer has to improve it. In the case of a versioning service, the repository designer specifies the product data objects and their relationships. The resulting UML model is called a product data model. Since the versioning framework is not able to deal with multiple inheritance, one rule of the guidelines checks, whether or not the product data model only contains single inheritance.

3. The repository designer is the person specifying application-specific semantics needed by SERUM to generate repository manager services.
4. Unified Modelling Language [16, 17]. See Sect. 5.1 for more details about the usage of UML in SERUM.
5. Note that, as described in the following paragraphs, the application-independent aspects of the specification will be automatically incorporated by the subsequent steps of the SERUM process.
6. Object Constraint Language [14].

Guidelines depend on the (kind of) repository services the repository designer wants to specify, e. g., a versioning service, and are part of a corresponding ***SERUM framework***. Such frameworks consists, besides the guidelines, of two main parts, the ***SERUM design patterns*** and the S***ERUM templates***. There is on framework each service. The notion of frameworks and the corresponding components are discussed in more detail later on (see Sect. 5.2).

The next step is to enhance the UML model specified by the repository designer and to complete the specification automatically by exploiting domain knowledge (4). A SERUM tool, the SERUM model enhancer (SME), supports this step. It first checks the guidelines and, if all constraints are fulfilled, the model is enhanced by a set of default base classes needed by the framework. Afterwards the model is altered, following modification rules. These rules are described as scripts, which are part of the SERUM design pattern definition (see example in Fig. 6). For example, all versions managed by our versioning framework have to be instances of a particular class ProductDataObject. This rule is specified by a design pattern which inserts an inheritance relationship between this root class and each (product data) class specified. The (product data) classes themselves are incorporated as base classes. Everything, the repository designer has to do, is to enumerate all classes he wants to be versionable. He can perform this either interactively or use a scripting language. The process of applying design patterns to a UML model can be seen as a customization process in which the framework is adjusted to the needs the repository designer wants to be fulfilled. It results in a UML specification of a repository service, which is stored in the SERUM UML repository.

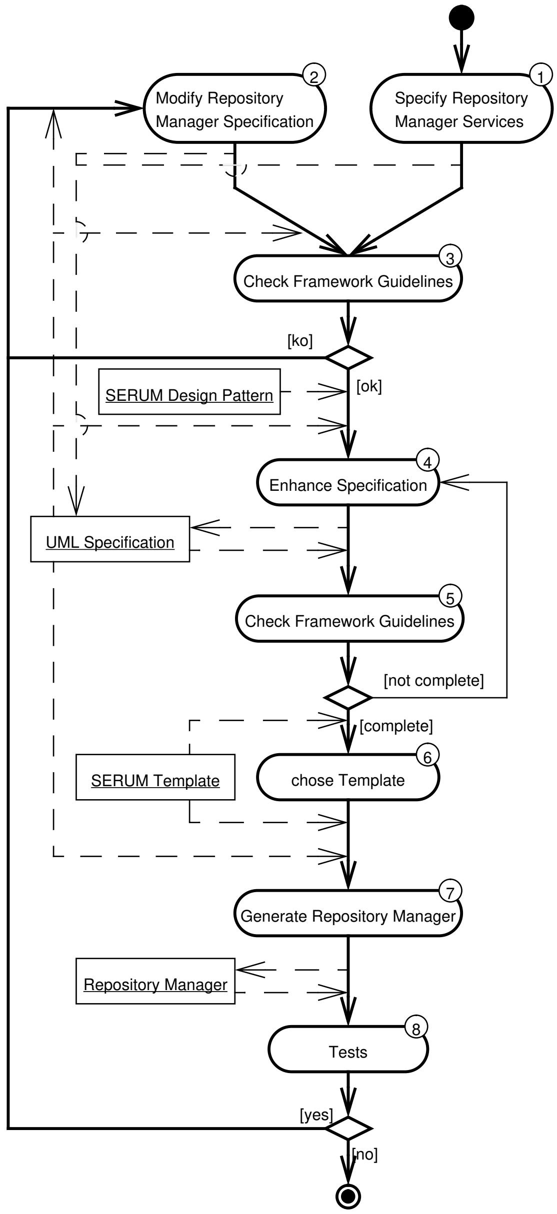

Fig 2: SERUM Process

Now, the UML specification has to be checked again. This time it is proven, whether or not the specification is complete in the sense that the repository generator can generate a repository manager from it. If not, the repository designer has to adapt the initial specification or to continue with the process of enhancing the specification.

While the design patterns provide the implementation-independent parts of a SERUM framework, SERUM templates represent the implementation-specific parts. Each framework can contain multiple templates, where each template includes the components for generating an executable repository manager. Such components are source code templates, 'ready-to-use'-components and source code generation rules. Hence, a template provides the building blocks of a repository manager. It represents the components of a system which follows a particular set of design and implementation decisions regarding programming languages, database systems and software architectures. Additionally, templates determine non-functional properties. Therefore, a framework can contain miscellaneous templates in order to comply with various requirements. The repository designer has to chose a template which fulfils the needs (6) he has in mind. In our versioning example, he may want to access the repository by a Java API. Hence, he would chose a template supporting Java APIs. Consequently, the template includes an implementation of the abstract class ProductDataObject in Java. SERUM templates are discussed in more detail in Sect. 5.2.

The UML specification serves as input for the SERUM ***repository generator*** (SRG). Based on the 'half-fabricated' components provided by the SERUM templates this tool generates the new repository manager (7). It produces a ***repository database schema***, a ***customized tool API*** and (several) ***repository servers*** which together establish the new repository manager (Fig. 3). In this step the application logic needs to be integrated into the repository manager. The repository designer has to provide implementations for those methods, which neither exist as source code templates nor can be generated from input specification. Additionally, the repository generator stores information about the generation process in the SE-

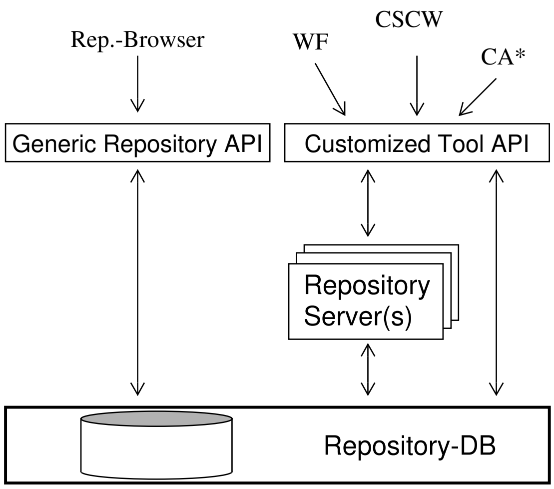

Fig 3: Repository Manager

RUM meta-database. This information enables generic tools, as for example browsers, to access repository data via the ***generic repository API (GRAPI)***.

Afterwards, the new repository manager is tested. If changes are necessary, the whole process restarts with the modification of the specification (2).

Fig. 4 illustrates the tools involved in the SERUM process. The whole process is based on the SERUM repository, which stores all information needed for generating the repository manager. It consists of three main components, the UML repository, the framework repository and the meta-database. UML specifications are stored in the UML repository which manages versions of UML models. In the framework repository the definitions of the frameworks, i. e. the definitions of the patterns and the components of the templates, are stored. The third component is the meta-database. It is used to store intermediate data processed in the steps of the SERUM process. If, for example, a design pattern script is executed by the SME, then identifiers of the initial version of the UML model, the design pattern and the resulting version are stored here. Additionally, the mapping information needed by the GRAPI is stored in the meta-database. An API, called the ***repository definition API***, provides access to the repository for the SERUM tools, mainly the SME and the SRG.

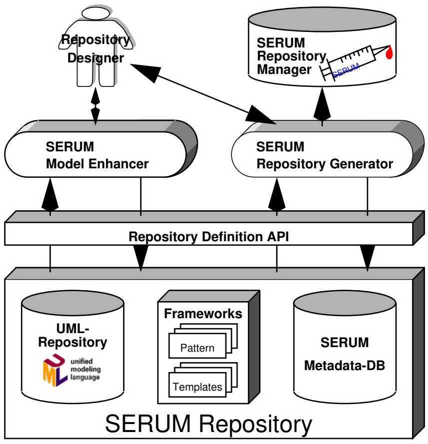

Fig 4: SERUM Tools

4. Exploiting Reuse Techniques

One of the main research goals in the software engineering community is to simultaneously speed up the software development process and increase software quality. Software reuse is widely accepted to be a useful approach in order to cope with these problems. Good surveys of this research field are given in [9], [18] and [12]. As stated in Sect. 2, we want to support the process of building specialized repository managers by starting with a UML specification reflecting the properties all repository managers have in common and, afterwards, supporting a transformation process, eventually delivering an executable system. The steps of this transformation process heavily rely on well-known reuse techniques. Consequently, we want to relate the methods, tools and techniques of the SERUM approach to these techniques. For that purpose, we consider the SERUM process itself as a development process as characterized in [12] and try to identify, which of the reuse techniques discussed in [12] and [9] are exploited in SERUM.

Software reuse enfolds the reuse of the products of earlier software development projects as well as of the processes or methods. In [12] the former is referred to as building-block approach. The idea is to reduce a design problem to sub-problems which are already solved. This is a bottom-up approach, because more complex systems are composed by existing components. Reusing processes and methods of former projects takes place in generative approaches. The process of software development is viewed as a sequence of transformations, whereas each transformation step transforms the description of a software system, eventually resulting in an executable system. Transformation rules describe the transformations to be performed. Following [12], the great advantage of reuse is to reduce the effort for specifying the initial description of the system by exploiting domain knowledge and to reduce the amount of work performed manually in each transformation step by implementing the transformation rules in tools, as for example application generators.

According to [12], we consider the SERUM process itself as a sequence of transformations. It starts with a (usually incomplete) UML specification and finishes by generating an executable repository manager. Main parts of the generation process are parametrizing and composing the components of the templates which serve as building blocks.

Using an object-oriented modelling language for the initial specification allows, at least in data-centric domains, specifying *"in terms of objects that have a direct link to occurrences in the real world"* [4]. Hence, the repository designer is able to cover a broad domain with the same set of modelling elements. Additionally, SERUM reduces the effort of specifying descriptions the processes have to start at by providing the concept of SERUM frameworks. Frameworks reduce the design space to a particular domain section, so that the semantics of a valid UML model can reasonably be captured by using domain knowledge and can be completed by applying design patterns.

Completing the UML specification with design patterns is the first transformation step of the SERUM process, where the design patterns represent the transformation rules. Because the transformation step is encoded within a design pattern script, it is possible to perform the transformation automatically. The repository designer still has to chose the design patterns, whereas the transformation is executed by the SME tool. The next transformation step is to specify the technology-dependent aspects of the target repository manager. Here, the repository designer has to chose a configuration from a list of options, e. g., programming languages, implementations of components with different non-functional properties, database systems and software architectures. This step completes the specification by taking the technology-dependent design decisions. It results in selecting a template providing the information needed by the repository generator. Finally, the repository generator creates the repository manager and transforms the specification into an executable system. Unfortunately, the components of a template cannot provide the application logic. Therefore, the repository designer has to provide it before the generation step can be performed. Basically, the repository generator integrates application logic with 'ready-to-use' components and generated code.

Hence, the SERUM process is a sequence of transformations which are performed by tools with developer assistance. This resembles the point of view presented in [12]. UML and the domain knowledge integrated in a SERUM framework allow for a lean specification of the repository managers. Frameworks reduce the design space to a particular domain section so that the semantics of a valid UML model is definite. Process reuse in SERUM is supported by the components of the frameworks, i. e., design patterns and templates, and the SERUM tools, i. e., SME and SRG. The SERUM design patterns cover the evolution steps and the SME handles the error-prone work of manipulating the model. Most reuse of process knowledge is utilized by the repository generator. Apart from incorporating application logic, the whole implementation process is automated in the repository generator. It takes advantage from reusing the source code templates, components and code generation rules provided by the templates as building blocks for more complex systems. This enables the repository designer to reuse process knowledge w. r. t. technology and implementation skills.

Krueger [9] has classified eight reuse techniques, which are high-level languages, design and code scavenging, source code components, software schemas, applications generators, very-high-level languages (VHLL), transformational systems and software architectures. We will now examine the SERUM approach w. r. t. these reuse techniques.

- *High-level languages*

  High-level languages are a good example of a widely used reuse technique. Although they are often used in today's software projects, nearly nobody recognizes that here reuse is at work. The objects of reuse are machine-code implementation patterns and process reuse occurs in the compilers. Of course, we do not claim that we take great advantage from using high-level languages for building our repository managers. Even if we would build them from scratch, we would use a state-of-the-art high-level language. Nevertheless, it is a really successful reuse technique and we want to mention it for sake of completeness w. r. t. [9].

- *Design and code scavenging*

  In design and code scavenging the objects of reuse are code blocks, which are copied from an existing software product to a new one. If large parts of the code are changed, it is called design scavenging, because only the basic ideas and algorithms are reused. Although design and code scavenging is a wide-spread reuse technique it should not be exploited in a development process, because it is more an ad-hoc than a structured method of reuse. Hence, it is not used in the SERUM process.

- *Software components*

  Following the idea published in [11], mass-produced reusable software components should be used as building blocks for larger systems. In the SERUM approach we take advantage of reusing source code components at various points. First, our components and source code templates are based on standard software libraries like STL [1]. Second, the components and code templates are reused in multiple repository managers.

- *Software schemas*

  Software schemas are more abstract than components. The objects of reuse are abstract algorithms and data structures. A software schema has parameter slots, which need to be filled with abstract entities in order to instantiate the schema. Instantiating a software schema usually means to generate source code. There may exist several implementations of a software schema, which is the main difference to programming language templates. A particular implementation can be chosen in order to fulfil a given set of non-functional requirements. SERUM design patterns are strongly related to software schemas. They provide implementation- independent specifications of design problem solutions and the repository designer can chose a suitable implementation from several SERUM templates. Some more detailed information regarding software schemas can be found in [7].

- *Application generators*

  An application generator is *“a tool or a set of integrated tools that inputs a set of specifications and generates the code of an application within an implementation language”* [12]. In contrast to compilers, the specification may be partial and in a totally non-procedural form. Partial specifications are completed by the application generator using domain knowledge. Hence, application generators are only useful in restricted domains. Application generators are complex systems. Therefore, many similar software systems must be generated in order to reach a pay-off. The SERUM repository generator is used in a restricted domain, which is the family of repository managers it is able to create, and is limited by the available frameworks. Since we think that many of the repository managers of a family are needed by certain applications, we expect to get a pay-off, too. Hence, the SERUM repository generator is a good example of an application generator.

- *Very-high-level languages*

  In contrast to application generators which exploit domain knowledge and are only usable in a particular domain, VHLLs try to support general-purpose software development. Usually, they are based on mathematical abstractions that are more natural and expressive than traditional programming languages. UML is not a VHLL, because it is neither executable nor can a compiler translate a UML model into an executable system. That is why we try to restrict the design space and use the design patterns to complete the specification until the templates can be used in order to build an executable system. However, we do not think that there is any counterpart to VHLLs in our approach.

- *Transformational systems*

  We have already discussed the relation between SERUM and the transformational systems point of view on software development. Although SERUM is not a transformational system w. r. t. the definition given in [9], there are some similarities. As claimed in [9], a transformational system starts with an executable high-level specification and performs a set of transformations, each modifying the system in a semantically correct way in order to reach a higher goal, e. g., more performance. Although in the SERUM approach only the result is an executable system, each step transforms or completes parts of the specification with the intention to generate an executable repository manager in the end.

- *Software architectures*

  Software architectures describe the components of a complex system with their relationships and communication protocols. Reusing software architectures means to reuse design knowledge at a very high level. There is a strong linkage between the reuse of software architectures and application generators. Typically, all systems generated by the application generator have the same (or a similar) architecture. In the SERUM approach, we address the objective of building customized repository managers for applications with different needs. Therefore, only a single architecture would not be sufficient. That is why we are developing a generic architecture which covers the main aspects of our domain. The repository generator composes the repository manager according to the software architecture which serves as a blue print. Some more details of the software architecture are presented in Sect. 5.2.

As shown in this section, the SERUM approach takes advantage of reusing process knowledge as well as software components. It combines the generative approach with the building block approach and exploits domain knowledge. Furthermore, it exploits various reuse techniques identified in [9]. In the following section, we want to complete the discussion by taking a closer look at the objects of reuse.

5. Objects of Reuse

In the previous section we have related the SERUM approach to several reuse techniques. Furthermore, we have shown that the repository building process takes great advantage of process reuse in each transformation step. Additionally, we have identified the reuse techniques applied in our approach, which are SERUM design patterns (software schemas), SERUM templates (source code components, source code templates, software architectures) and the SERUM repository generator (application generator). This section will complete the discussion about reuse in SERUM by identifying the major objects of reuse and the contributions of the repository generator. First of all, however, we want to recall the advantages of UML, since UML specifications are subject of all transformations performed within the SERUM process and serve as input for the repository generator.

5.1 UML - The SERUM Meta-Model

We have chosen the UML meta-model [16, 17] for the following reasons. First, we believe that UML will play an important role as a standard language for specifying and documenting the artifacts of software systems. The Object Management Group (OMG) has integrated the UML into the Object Management Architecture (OMA) and also specified an OA&D CORBAfacility [13] as a standard interface for accessing UML-based tools via CORBA [15]. Furthermore, the current UML standard has become broadly supported by the vendors of graphical modelling tools and is widely used by system architects for analysis and design of software systems. Second, we expect using UML not only during analysis and design but also for specifying value-added data management services will help to reduce the cognitive distance between these two steps and, additionally, may help for bridging the gap between modelling and semi-automatically implementing components of software systems. Third, in comparison with other information models, e. g., the ER-model, UML has lots of advantages:

- UML is object-oriented and object orientation has become the leading technology in software development to cope with the high complexity of modern systems. As we have already mentioned, the advantage of using an object-oriented modelling language is that the specification is done in the vocabulary of the domain.
- UML allows for detailed modelling of class structures and relationships. Especially, semantic relationship support is a key feature of repositories. UML offers a large set of structural modelling elements, including several options to define the semantics for relationships.
- UML allows for specifying constraints in its formal object constraint language (OCL) [14].
- UML provides modelling elements for dynamic (behaviour) and for state-oriented aspects.
- UML is extensible.

5.2 SERUM Frameworks

The generic methods offered by SERUM to customize repository managers are based on a set of pre-defined frameworks. According to [5] "*a framework is a reusable design of all or a part of a system that is represented by a set of abstract classes and the way their instances interact.*"

SERUM provides a framework for each domain section (versioning, activity support, ...); each framework consists of three parts, the framework ***guidelines***, the technology-independent ***design patterns*** and the technology-dependent ***templates***.

SERUM Framework Guidelines

A SERUM framework comes along with guidelines. Guidelines provide the default structures, each specification of a repository manager must include. Additionally, guidelines describe the properties of intermediate results of the SERUM process. They cover the following aspects (Fig. 6, in order of appearance):

- *Domain Guidelines*

  A SERUM framework is linked to a particular domain. All elements of a framework, e. g., design patterns and templates, depend on assumptions made w. r. t. to the framework's domain. Hence, the UML model used for specifying a repository manager have to be in accordance to these assumptions. Therefore, constraints must be specified describing the correctness of UML models. The SME checks these ***domain guidelines***. They are specified in OCL.

- *Default Structures*

  Default structures are, for example, abstract base classes with default attributes, relationships and behaviour. The design pattern scripts depend on the existence of these base structures. Therefore, it is a guideline that these default structures must exist in the model, if a framework is used. They are to be enumerated in the framework definition and the SME subsequently incorporates the corresponding specifications.

- *Model Guidelines*

  The SRG makes assumptions about the UML model used as input. After the repository designer has applied the design patterns, the SMG checks, whether or not the resulting model is a correct input for the SRG. These ***model guidelines*** are also specified in OCL. Additionally, there are guidelines which are related to all frameworks. An example is that for each class added to the specification an implementation in each template is required. These guidelines are not associated to a framework, but embodied in the implementation of the SRG. They are checked as soon as SRG starts to generate the repository manager.

SERUM Design Patterns

In [3] a design pattern is defined as “*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*”. SERUM design patterns are used to automatically enhance UML models initially specified by a repository designer. A SERUM design pattern consists of two parts, an informal one helping the repository designer to select the pattern, and a formal one used to refine user specifications. Due to the second part, SERUM design patterns are much more formal than those presented in [3].

Fig. 5 illustrates the process of using SERUM design patterns. The repository designer specifies an initial UML model of the domain objects (1). Usually, this model is incomplete and mainly contains the application-specific parts. The SME is used to apply the *design pattern script*, as the formal part of a SERUM design pattern is called, to the user-specified UML model (2). This way, the model is enhanced by new classes and relationship types. Furthermore, already existing classes may be modified, if necessary (5).

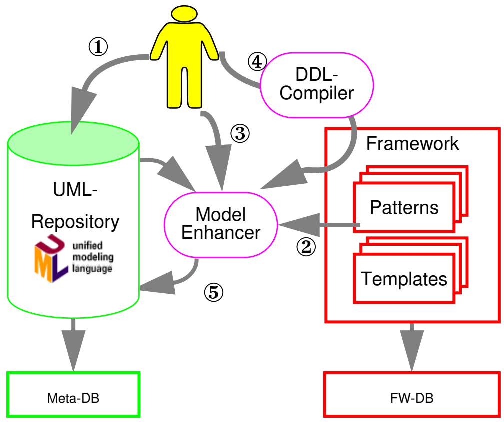

Fig 5: Applying Design Patterns

While the SME automatically applies design pattern scripts to UML models, the repository designer must select the patterns to be applied and determine the generic parameters of the pattern application. This can be done either interactively, where the user chooses the patterns and parameters (3), or by external tools (4). These tools may allow the repository designer to express specifications in a more abstract language which depends on the corresponding SERUM framework. For instance, the versioning framework, we used as an example for the discussion in Sect. 3, comes along with a specific data definition language (DDL) and a corresponding compiler translating DDL statements into sequences of SME calls.

After the mentioned definition steps are finished, a technology-independent specification of the repository model is stored in the SERUM UML repository.

To support the outlined specification process appropriately, a SERUM design pattern must fulfil two requirements. First, searching design patterns must be supported. Several design pattern languages [3] try to cope with this problem. Hence, we want to concentrate on the second requirement which is the provision of an appropriate design pattern scripting language. As we think, a scripting language must at least offer constructs for specifying the following elements (Fig. 6, in order of appearance):

- *Parameters*

  Parameters specify the elements of the input model which are to be manipulated.

- *Preconditions*

  If a SERUM design pattern is applied in order to enhance a UML model, this model has to fulfil preconditions. The SME checks these preconditions, which we specify in our design-pattern definition script using OCL constraints.

- *Constraints*

  Usually, more than one design pattern is applied in order to enhance the UML model. Hence, it must be avoided that a subsequent design pattern script can destroy the changes made by an earlier one. Therefore, a design pattern defines some constraints, which have to be valid after the repository designer has finished his work. These constraints are dynamic extensions to the model guidelines defined by the framework. Constraints are expressed in OCL.

```
begin define framework “VersionManagement”
  begin domain guidelines
    ClassClass.allClasses()
        ->forAll( c
            | c.supertype().size = 1;
    // additional domain guidelines
  end domain guidelines
  begin default structures
    begin mdl
        ( object Class “ProductDataObject”
          // definitions
        )
    end mdl
    // additional default structures
  end default structures
  begin model guidelines

    // additional model guidelines
  end model guidelines
  begin define pattern “ProductDataObject”
    begin parameters
      ClassUList   : aClasses;
    end parameters
    begin preconditions
      aClasses->forAll( c | exists(“Class”,c.name )
    end preconditions
    begin constraints
      aClasses
      ->forAll( c
          | c.allSupertypes
               ->exists( s
                  | s.name()=“ProductDataObject” ));
    end consttraints
    begin alter model
      for ( int i; i < aClasses.length; i++ )
      {
        GeneralizationClass.create_generalization
            (aClasses[i].name(),“ProductDataObject”
      }
    end alter model
  end define pattern
  // additional framework components
end define framework
```

Fig 6: Framework Definition Script

- *Model Evolution Operations*

  Applying a design pattern to a UML model means integrating the user-specified definitions and the (pre-defined) framework infrastructure used by the pattern. This, actually, happens by manipulating the user specifications: Class definitions may be enriched by new attributes, new relationships, new superclasses, new methods, etc. For that purpose, the script may contain calls of model evolution operations; the parameters of these operations are usually elements of the user specification. In the OMG OA&D CORBA facility [13] a set of model management operations is already defined. Hence, we use similar operations.

After the SME has applied all relevant patterns, a technology for implementing the repository manager has to be chosen. Implementation is supported by *SERUM templates*.

SERUM Templates

Partitioning a framework into a technology-dependent (*templates*) and a technology-independent (*patterns*) part enables design solutions without being technology-dependent. Technology refers to programming languages (C++, Java, ...), ORDBMSs, strategies for architectural design (client-centric, server-centric, repository servers, caching and buffering strategies) and communication mechanisms (CORBA, OLE, RPC, ...).

The idea of a SERUM template is to bring together a configuration of components with two main properties. First, these components are able to work together in a repository manager. Hence, SRG does not need to know which configurations are useful. Second, the configuration of components will lead to a repository manager with well-known properties, regarding non-functional requirements. If the repository designer selects a template, then he also determines the properties of the resulting repository manager. Each component is allowed to be included in many templates.

The components of a SERUM template can be divided into three main groups:

- *Code Generation Rules*

  As we have mentioned before, several programming languages need to be supported. We do not want to embody the mapping rules for every target language into the code of the repository generator, because this would restrict the repository generator's usability too much. Therefore, the SRG must be able to generate code following code generation rules provided by the template.

- *Source Code Templates*

  The concept of source code templates is well-known from programming languages, especially C++. Unfortunately, not all programming languages provide this concept. Additionally, the expansion rules are not flexible enough, mostly limited to simple term rewriting mechanisms. Hence, we want to integrate functionality into the repository generator which allows for expanding source code templates.

- *'Ready-To-Use'-Components*

  'Ready-to-use' components can be applied without generating or modifying code. If such components need to be customized, this is done via parameters. The implementations of the abstract base classes (if they result from expanding a source code template) are examples of 'ready-to-use' components. Also, repository servers and client caches are usually used without code modifications. Each component may be part of more than one template, but a template provides the particular set of parameters, as for example the caching strategy, the repository generator needs for customizing the components. Other parameters are set by the repository designer when he chosen the non-functional parameters of the target repository manager.

A SERUM framework must provide building blocks for the customized API, the repository servers and the database schema. Thus, a SERUM template provides rules for creating API libraries from the UML model. These rules map the object model of the framework to the object model of the PL. Furthermore, DDL commands for creating the database schema in the data model of the target DBMS must be derived. Although we cannot detail this aspects due to space limitations, we want to mention that the concepts of object-orientation, i. e. inheritance, overloading, etc., are extremely helpful and, hence, the new object-relational database technology is by far more suitable than the current pure relational DBMSs.

5.3 SERUM Repository Generator

As we have already discussed in Sect. 4, the SERUM repository generator is a typical example of an application generator. The SRG has to perform three jobs: First, it has to expand the source code templates. Second, the code generation rules are executed in order to generate code. Third, the result of the first two tasks has to be combined with the 'ready-to-use' components and the application logic provided by the repository designer. The components are customized with the parameters provided by the SERUM template.

A generic software architecture serves as a blueprint of the repository manager to be build. It determines the components needed to establish a repository manager, the relationships between them, the interfaces and protocols used for cooperation. If, for example, no caching component is needed at the client, then the SERUM template must provide a dummy cache, which is inserted by the SRG. This component has no functionality except routing all requests to the database server.

6. Conclusions

In this paper, we have discussed reuse aspects of the SERUM approach providing an infrastructure for tailoring repository managers to special application needs by exploiting generic methods. The SERUM approach, in general, is beneficial for the following reasons:

- The repository designer only has to model application-specific aspects of the repository manager service, he wants to realize. Corresponding specifications may be expressed in UML.
- Pre-defined, technology-independent design patterns relieve repository designers by automatically enhancing UML models by application-independent aspects.
- Repository manager functionality (code) is automatically generated/composed from the customized UML model by applying technology-dependent design templates.

Obviously, the basic idea of reusing design artifacts has effectively been put into practice by the SERUM approach. We have related the SERUM approach to several reuse techniques. The repository building process takes great advantage of process reuse in each transformation step. Additionally, the reuse techniques in form of software schemas (SERUM design patterns), source code components, source code templates, software architectures (SERUM templates) and application generator (SERUM repository generator), are exploited.

The extensive application of reuse in SERUM considerably contributes to the goals of SERUM, which are providing a basic infrastructure for assisting and relieving repository designers in creating tailored repository manager functionality. Reuse is not only the foundation for the tailoring process, but also simplifies this process for the user by hiding details, reduces error probability, and helps to capture the complexity of the software generation step.

Literature

[1] Austern, M. H.: Generic Programming and the Stl : Using and Extending the C++ Standard Template Library, Addison-Wesley, October 1998

[2] Berstein, P. A., Dayal, U.: An Overview of Repository Technology, Proc. 20th VLDB, Santiago, Chile, September, 1994, pp. 705-713

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995

[4] Jacobson, i., Ericsson, M., Jacobson, A.: The Object Advantage - Business Process Reengineering with Object Technology, ACM Press, 1995

[5] Johnson, R. E.: Frameworks = Components + Patterns, CACM, 40:10, October 1997, pp. 39-42

[6] Katz, R.: Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No. 4, December 1990, pp. 375-408

[7] Katz, S. Richter, C. A., The, K.: PARIS: A System for Reusing Partially Interpreted Schemas, 9th International Conference on Software Engineering, IEEE Computer Society Press, 1987, pp. 377-385

[8] Kim, W.: Object-Relational - The unification of object and relational database technology, UniSQL White Paper, 1996

[9] Krueger, C. W.: Software Reuse, ACM Computing Surveys, ACM Press, Vol. 24, No. 2, June 1992, pp. 131-183

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

[11] McIllroy, M. D.: Mass produced software components, in Naur P., Randel B., eds., Report on a Conference of the NATO Science Comittee, pp. 138-150, October 1968

[12] Mili, H., Mili, F., Mili, A.: Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995, pp. 528-562

[13] OMG, OA&D CORBAfacility Interface Definition, Version 1.1, OMG Document ad/97-08-09, September 1997

[14] OMG, Object Constraint Language Specification, Version 1.1, OMG Document ad/97-08-08, September 1997

[15] OMG, The Common Object Request Broker: Architecture and Specification, Version 2.2, OMG Document ad/98-07-01, August 1998

[16] OMG, UML Notation Guide, Version 1.1, OMG Document ad/97-08-05, September 1997

[17] OMG, UML Semantics, Version 1.1, OMG Document ad/97-08-04, September 1997

[18] Prieto-Diaz, R.: Status Report: Software Reusability, Software, Vol. 10, No. 3, pp. 61-66, May 1993

[19] Stonebraker, M. Brown, P. Moore, D.: Object-Relational DMBSs, Second Edition, Morgan Kaufmann Series in Data Management Systems, September 1998