Strategies and Techniques: Reusable Artifacts for the Construction of Database Management Systems[1]

Andreas Geppert, Klaus R. Dittrich

Computer Science Department
University of Zurich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
Email: {geppert|dittrich}@ifi.unizh.ch

Abstract

Abstraction, selection, and integration of reusable artifacts are still open problems in reuse-based software construction. We investigate how these problems can be solved for one sample domain of system software: database management systems. We propose *dimensions* for functional subdomains where different design choices are to be made. *Strategies* are the design choices for dimensions; their application leads to refined architectures and designs. Moreover, they are used to control the integrity of designs. *Techniques* are classes in the sense of object-oriented programming and implement combinations of strategies; they are reused during subsystem implementation. Through the various levels of abstraction, a designer/implementor of a database management system is guided towards reusable artifacts in all phases of a construction process.

Keywords: software reuse, DBMS-construction, system design and specification

1 Introduction and Motivation

Software reuse [11] has been proposed as a means for the systematical construction of software systems in a less costly way than is possible by starting implementations from scratch. Reuse is not restricted to code fragements only, but is also applicable to specifications, designs, etc.. We therefore prefer to talk about (reusable) *software artifacts*. In order to make reuse work, several problems must be solved [11]:

- *abstraction*: software designers need to understand *what* they are going to reuse. In order to be understandable, artifacts must be represented at the appropriate level of abstraction.
- *selection*: especially for large collections of reusable artifacts, it is a problem to find the "right" artifact to reuse for a given problem. Software designers therefore need help for selecting candidate artifacts for reuse.
- *integration*: even if reusable artifacts have been found for each component, those still need to be integrated into a coherent, consistent software system.

Depending on the domain of interest, these problems are aggravated by the following reasons. If for a domain (of software systems to be constructed) standard solutions are not known and many (semantically) different implementation approaches are known, then a reuse-based approach has to support the classification of these variants (the abstraction problem). Consequently, it is then harder to find the appropriate variant in the large collection of reusable artifacts (the selection problem). Furthermore, if standard interfaces do not exist and implementations are not interchangeable, the problem is to

1. To appear in J. Iivari, K. Lyytinen, M. Ross (eds): Proc. 7th Intl. Conf. on Advanced information Systems Engineering (CAiSE), Jyvaskyla, Finland, June 1995.

achieve a consistent design and to make compatible choices of artifacts to be reused (the integration problem).

In this paper, we propose *strategies* and *techniques* as concepts that address the problems of abstraction, selection, and integration in reuse-based software construction. Specifically, we assume that the domain of interest is characterized by large sets of variants for certain functionalities, and that standard solutions do not exist for most functionalities. In our approach,

- *dimensions* represent properties that are used for classification and represent design decisions to be made,
- *strategies* distinguish among variant design decisions on a high level of abstraction,
- *techniques* are reusable components implementing combinations of strategies.

Throughout the paper we assume that object-oriented technology is applied.

We show that dimensions, strategies, and techniques can be used to solve the aforementioned problems in the reuse-based construction of database management systems (DBMSs). A DBMS is software that provides for the persistent management of databases, including modelling and storage of data, declarative retrieval and manipulation, recovery, multi-user access, integrity enforcement, and so forth. A large variety of data models, transaction models, integrity maintenance concepts, authorization models, and so on exists, and many different implementations for each of these models exist. All the problems mentioned above therefore occur in reuse-based DBMS-construction. However, since these problems are present in other domains as well, we feel that our results can be helpful in other fields of software construction, too.

Subsequently, we describe the underlying architecture model of this approach and then introduce the three concepts. We show how they are used during the construction process. Throughout the paper, we present examples from the concurrency control field.

2 The Architecture Model

Reusability of analysis and design information is much more promising than code reuse [3, 13]. Hence, the DBMS-constructor[2] has to be guided through all the construction phases towards reusable artifacts, and —moreover— concepts must be provided that allow the representation of analysis and design information. Second, recall the integration problem: reused artifacts still need to be integrated into a coherent entirety. We thus propose an architecture model for DBMSs that is able to

1. represent reusable architectures and designs,
2. integrate reused components into a coherent DBMS.

Apart from these features, the architecture model is intended to meet a variety of requirements:

- structural decomposition and information hiding,
- modeling of activities and behavioral decomposition,
- support for architectural constraints, and
- transparency and integration.

2. Synonymously, DBMS-Implementor (DBI)

These requirements are all met by the so-called *broker/services architecture model*. The prime concepts of this model are brokers, services, and responsibilities.

A *service* is a specific functionality provided by the DBMS[3] or a part thereof. A service has a *signature*, which in turn consists of a name and a list of typed formal parameters. Services can be requested by DBMS-components through the raising of *events*. In raising such an event, a requestor has to supply actual parameters according to the service definition. Each service defines (a set of) possible *replies*. Whenever a service can be provided successfully, one of the replies is raised. That is, replies are raised like request events, and can have parameters as well.

Brokers are active entities and implement subsystems. They are able to request services and to satisfy requests for services. Brokers have a state, which can be composed out of further (sub)brokers or passive objects.

Responsibilities are relationships between services and brokers, they define which broker provides a specific service. Whenever a request is made, the responsible broker "catches" the event and reacts by executing the service implementation. Services are implemented through sets of production rules. Each production rule defines an event (the triggering event, i.e., the request), a condition, and an action. Actions of production rules in turn are sequences of messages sent to the passive components of the broker, or generic operations such as replies.

In the case that multiple production rules are defined for the same event (i.e., within the same activity), their execution order can be constrained if necessary. The execution order is defined through a partial ordering of production rules.

Based on these concepts, a DBMS-architecture is defined as follows.

Definition 1 (DBMS-Architecture)

A *DBMS-architecture* is a collection of brokers, services, and responsibilities.

A DBMS presents itself to its clients via a special broker called the *globe*[4]. The globe is responsible for those services that the DBMS-clients can request as well as for coordinating the brokers for the various subsystems. In Figure 1 an example of the brokers/ services architecture model is illustrated. Apart from the globe, the architecture contains three brokers on the top level: the object management subsystem (OMS), the transaction management subsystem (TMS), and the integrity management subsystem (IMS). Arrows represent service requests, and the white rectangle represents an "event bus" which is required for exchanging requests and replies.

3 Strategies and Techniques

Feasible reuse techniques must be developed for each aspect of a DBMS. Generation is one possibility (e.g., for query optimizers [8]). For other aspects a large variety of design and implementation alternatives exist that cannot be handled by generators. For these aspects, we propose the use of configuration/composition. Obviously, a reuse-based construction method must support the representation, classification, and selection of reusable artifacts whenever such a variety exists. Second, it is a characteristic

3. Henceforth, we refer to a DBMS under construction simply as "DBMS".

4. **glo**bal **b**rok**e**r, also a sign of power and authority (Webster's dictionary).

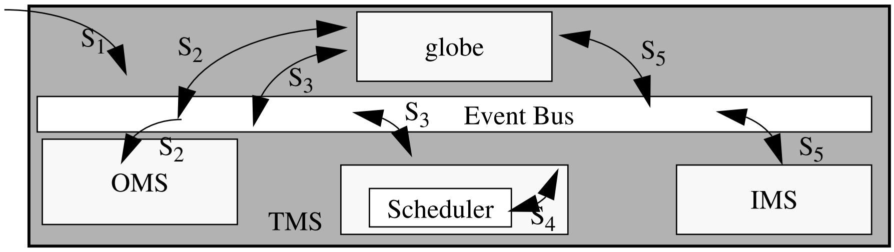

Figure 1. Brokers and Service Requests

of DBMSs (and presumably of other system software, too), that standard interfaces do not exist. Moreover, the correctness of the choice of a specific approach for some DBMS-functionality may depend on the choices already made for other DBMS-components. Hence, the construction approach must enforce consistency of reused artifacts [14]. In this section, we introduce strategies and techniques as the major concepts for consistent reuse of artifacts for large component domains. We first give an overview of the different levels of abstractions used in our approach, and then describe the concepts of strategy and technique in detail.

3.1 Levels of Abstraction of DBMS-Software Artifacts

Selection, abstraction, and integration are the very problems for any reuse-based method. In the DBMS-construction context, these problems are handled by introducing different levels of abstraction. At least for those subsystems that have to be composed, the use of the various abstractions guides the DBI from partial and abstract designs towards reusable classes.

In the sequel, let us consider an arbitrary but fixed (part of an) aspect that has to be realized by means of composition/configuration, such as *database concurrency control*. Concurrency control is the DBMS-task of preventing inconsistencies caused by interleaved, concurrent transaction executions. Strict two-phase locking (s2PL) is the concurrency control mechanism used most often, but many others have been proposed and used as well. Subsequently, we assume that the reader has a basic knowledge on database transaction concepts.

One result of domain analysis is a number of *dimensions* (or categories of strategies) for the aspect in question. For the DBI, a dimension is a set of alternative design decisions, each of which extends the design and/or a broker in a specific way.

As an example, one dimension for concurrency control is *validation time*, and the required services for this dimension are `validate` and `register` (the former checks whether a database access on behalf of a transaction is allowed w.r.t. to the actual schedule, while the latter records this access).

Each possible design decision is termed a *strategy*. Thus, a category of strategies determines a design dimension, and each strategy represents a specific design alternative. Strategies are defined as constraints on the (DBMS- or subsystem) design.

In the example (*validation time*), the following strategies can be identified: *preclaiming* (request `validate` at the begin of the transaction execution), *pessimistic* (perform validation before the actual database access), and *optimistic* (request `validate` after the database access, but before the transaction is allowed to commit).

In this way, a strategy refines (or constrains) the design of brokers and subsystems. Whenever multiple dimensions refer to the same service, it is necessary to consider the selected strategies in their combination in order to determine a realization of the service. *Techniques* specify how combinations of strategies are realized. In special cases, where a strategy is orthogonal to others, a technique specifies the realization of the single strategy. Combinations of strategies can also be realized by various distinct techniques —this is actually the motivation for distinguishing strategies from techniques.

Techniques are components and specify how the combination of strategies is realized in terms of classes. Therefore, each technique specifies a state and a set of method signatures, along with pre-and postconditions for each method as well as invariants to be fulfilled by the state. In addition, a technique determines how it conforms to the strategy combination, e.g., which method realizes which service.

As an example, consider concurrency control again. In this case, let *validation time* and *validation reaction* be two dimensions, where *pessimistic validation* and *blocking* are two strategies, respectively. Then, strict two-phase locking is a technique that realizes the combination of both, `getLock` is a method of the technique, and this method conforms to validation (i.e., has to be called upon the `validate` request within the activity for `validate`).

A technique will be composed out of classes that perform specific tasks of the technique. These classes in turn are called *primitives*, and may be used in an arbitrary number of techniques (e.g., consider a `lockTable` class for the management of locks).

In the sequel, the four levels of abstraction will be defined in more detail.

3.2 Strategies

3.2.1 Formal Specification of Strategies

The definition of strategies specifies their meaning in terms of logic and can be used to reason about strategies and designs.

3.2.1.1 The Domain

The domain models the entities to reason about during broker design and strategy selection. Namely, these entities are brokers, services, service implementations, replies, and strategies as well as techniques themselves.

Let B, D, PC, PN, R, SE, ST, T be finite sets of identifiers, where

- B is the set of broker names,
- D is the set of strategy dimension names,
- PC is a set of parameter class names,
- PN is a set of parameter names,
- R is the set of reply names (events),
- SE is the set of service names,
- ST is the set of strategy names,
- T is the set of technique names.

3.2.1.2 Formulas

Table 1 shows the built-in predicates for the specification of brokers and strategies..

name	domain	meaning
has_component (b_1,b_2)	broker × broker	b1 is a component of b2
supports (b, s)	broker × service	b is responsible for s
applies(b, s)	broker × strategy	b applies s
has_parameter (s, pn, pt)	service × parameter name × parameter class	s has a parameter named pn and pn is of type pt
requests (s_1, s_2)	service × service	s1 requests s2
before (s_1, s_2, s_3)	service × service × service	within s1, s2 is executed before s3
within (s_1, s_2)	service × service	s2 is executed within s1
replies (s, r)	service × reply	s replies with r
raises (s, e)	service × exception	s possibly raises e
incompatible (s_1, s_2)	strategy × strategy	s1 and s2 are incompatible
realizes (t, s)	technique × strategy	t realizes s

Table 1 Predicates and Corresponding Domains

Structural decomposition of brokers is expressed by the predicate `has_component`. This predicate evaluates to `true` whenever the second broker is a direct component of the first one.

The second predicate for brokers expresses responsibilities. Namely, `supports` is true for a broker `b` and a service `s` if `b` is responsible for a service named `s`. The application of strategies by a broker is reflected by the predicate `applies`; this predicate is evaluated to true for a broker `b` and a strategy named `s` whenever `b` is defined to use `s`.

The fact that a service (realization) requires another service is expressed with the predicate `requests`. Note that the second service does not necessarily have to be requested within the realization of the first one. This is specified by the predicate `within`, which evaluates to `true` for services `s1` and `s2` if within the realization of `s1` a request is raised for the service named `s2`. The condition that a specific service has to be requested before another one is modeled by the predicate `before`.

Furthermore the predicate `has_parameter(s,n,p)` requires the signature of a service named `s` to have a parameter named `n`, which in turn has to be of type (or class) `p`. The reply of a service definition is constrained by means of the predicate `replies`, which evaluates to `true` whenever the reply name is a possible reply event for the service in question.

The knowledge on strategies and techniques is modeled by two predicates for incompatibilities and realization of strategies. The predicate `incompatible` is defined for pairs of strategies, it specifies that the two strategies must not be chosen in combination. Second, `realizes(st,t)` evaluates to `true` if the technique named t is a realization of the strategy `st`.

Formulas can be constructed by means of the usual logical connectors and quantifiers. Strategies are then defined as formulas over a set of brokers.

Definition 2 (Strategy)
Let `s` be the name of a strategy, `b` the name of a broker, and `f` a formula with `b` as a free variable. A definition for the strategy named `s` is `strategy(s,b)` :⇔ f.

3.2.1.3 Properties of Brokers and Designs

The notion of strategy as defined above supports capturing the meaning of a specific strategy. Second, it can also be used in order to reason about brokers and designs.

Assume a rulebase of strategy definitions and predicates (facts) reflecting strategy incompatibilities. Furthermore, consider a design as a collection of facts in terms of the predicates introduced above. Then, a broker (or an entire design) is considered to be *strategically correct* if the design does not yield a contradiction to the rulebase.

Definition 3
Let `b1,...,bn` be the set of names of brokers under design, let `F` be a conjunction of formulas over `b1,...,bn`, and let `fs` be the definition for strategy `s`. Then, the design is strategically correct if the following implications are true:

- ∀ i: {1..n} `F` ∧ `applies(bi, s)` ⇒ `fs`
- ∀ i: {1..n} ∀ `s: ST` ∀ `s': ST F` ∧ `applies(bi,s)` ∧ `applies(bi,s')` ⇒ ¬ `incompatible(s,s')`.

These two conditions specify that a design does not apply contradicting decisions w.r.t. the strategy repository. Therefore, we will refer to strategies synonymously as “refinements of brokers and designs” and “constraints on brokers and designs”.

3.2.2 Sample Strategies

The concept of strategies has been used to classify approaches in the field of database concurrency control. Due to space restrictions, we cannot describe the entire performed domain analysis, but only give a summary and an example (for details, see [7]). 25 database concurrency control approaches proposed in the literature have been analyzed. On the top level, we obtained seven dimensions:

- *validation time*: when is a database access validated?
- *validation basis*: is a conflict-based, read/write schema used, or are transaction structure or transaction semantics taken into account?
- *validation reaction*: what happens when access cannot be granted immediately (e.g., transaction abort)?
- *granularity*: what is the granularity of validation (e.g., single objects)?
- *versioning*: are different versions of data items used?
- *data dependency*: does validation take the value of data items into account?
- *type dependency*: are the semantics of data types taken into account?

Most dimensions have two or three different strategies defined. As an example, consider the definition of the strategies for the validation time dimension in Example 1.

```
validationTime := {pre-claiming, pessimistic, optimistic}

strategy(pre-claiming, scheduler) :⇔
        within(begin, validate) ∧ has_parameter(validate, access, list(DBRequest))
strategy(pessimistic, scheduler) :⇔
    ∀ s: SE ∀ db-op: DB-Operation
        ( within (s, db-op) ∧ supports (OMS, s) ) ⇒
            ( within(s, validate) ∧ before (s, validate, db-op) ∧ within (validate, register) )
strategy(optimistic, scheduler) :⇔
    ∀ s: SE ∀ db-op: DB-Operation
        ( within (s, db-op) ∧ supports (OMS, s) )
            ⇒ (within(s, register) ∧ within(commit, validate)

incompatible(optimistic, updateInPlace)
```

Example 1. Validation Time Strategies

The `optimistic` strategy is a typical participant in an incompatibility: this strategy must not be chosen when OMS performs update-in-place for objects.

3.3 Specification of Techniques and Primitives

The first step in designing a broker is the selection of strategies. The ultimate goal, however, is to identify components that can be reused for the implementation of the broker and its subcomponents. How can one bridge the gap between strategies/brokers and components to be reused? Essentially, the following is required:

1. The design and implementation phases (see below) have to support the determination of candidates for reuse. Typically, a class repository will be quite large, such that it is an urgent task to restrict the selection space as far as possible.
2. Candidates can be correctly reused only under specific conditions (i.e., constraints concerning their usage). Hence, supporting *semantic integrity* for combinations of reused components [14] is required. In fact, instead of selecting a number of components and later on checking for consistency, it is preferred to propose only those candidates that can be reused while preserving consistency.
3. An underlying formalism for selection and representation of reusable components is required due to several reasons:
   - to support reasoning about semantic integrity of reused components,
   - to unambiguously describe semantics of components,
   - to support the precise definition of new components and techniques.

Subsequently, we show how techniques and primitives are specified.

3.3.1 Specification of Techniques

The role of a technique is to specialize the description of (a combination of) strategies and to bridge the gap between strategies and primitives (classes). The following properties characterize a technique:

- a unique name,
- the strategies it realizes,
- the primitives or techniques it uses, and

- conditions under which the use of this technique or its components is correct.

Techniques are akin to *contracts* [9], but we conceive techniques as formal specifications of *classes* as well (unlike the contract approach). A technique is specified by means of the following features:

- an enumeration of the participants,
- the signatures of the methods it provides,
- the signatures of the methods it requires from the participants,
- preconditions and postconditions for methods,
- invariants,
- conformances to the definition of strategies it realizes.

Techniques are implemented in terms of their component objects (the participants). The knowledge about participants is reflected in the technique specification.

Most important for ensuring the semantically correct specification of techniques are conditions and invariants. Preconditions and postconditions are always attached to single method signatures. A precondition states constraints on parameters to be obeyed by each client of the method. In other words, a precondition defines under which condition the method can be legally applied. A postcondition defines how any object executing the method is obliged to behave, i.e., it is a constraint on the server side. This principle is termed "programming by contract" [12], since pre- and postconditions are conceived as obligations for clients (preconditions) and servers (postconditions).

Invariants are another kind of constraint and specify conditions to be preserved by the technique implementation and its participants. Especially, invariants can refer to the state of instances and do not have to be attached to single methods.

Conformances specify how the technique implements specific strategies. Conformances are mappings from services defined for strategies to methods of a technique.

Consider the strict lock management technique in Example 2. Note that this specification is incomplete for the sake of brevity. The formal syntax of the technique specification language is not given, but the meaning of the various clauses is described subsequently. The `INCLUDES` clause specifies the participant techniques or primitives. Some participants are still generic and need to be instantiated (or specialized) for a concrete application of this technique (e.g., `Modes` for the concrete set of lock modes). In this case, the "<" sign specifies that the type before the sign has to be a subtype of the type after the "<". In other words, these participants are comparable to framework objects, i.e., abstract classes. The `STATE` clause specifies instance variables specific to the technique. The `METHOD` clauses specify the methods the technique provides to its clients. A method can be constrained by pre- and postconditions. The optional `ACTION` clause specifies messages that a component instance can send to other objects. Finally, the `CONFORMANCES` clause specifies *how* the technique implements specific strategies or services. In the example, `getLock` is an implementation of the `validate` service.

```
COMPONENT                                  // Refer to the text for explanations.
      StrictLockManager
INCLUDES                                   // Participating techniques, primitives
      TAID
      Modes < LockModes       // Modes is a subclass of the generic class LockModes
      Compat< Compat_Table
```

```
        DeadlockMgr
        LockRequest < DBRequest
        LockReqReply < ValidationReply
STATE                                                   // instance variables
        locktable: LockTable
        compat: Compat
        deadlockMan: DeadlockMgr
METHOD                                      // methods provided by the technique
        getLock (req: LockRequest): LockReqReply
        PRECONDITIONS
            req->ta->state = RUNNING
        POSTCONDITION
             LockReqReply = GRANT
                ⇒ locktable' = locktable->insert(req->ta, req->oid, req->mode)
            LockReqReply = BLOCKED
                 ⇒ deadlockMan->add_edge(req->ta, lockTable->lockholder(req->oid))
METHOD
        release_locks (ta: TAID)
        ...
ACTION
        resume (DBRequest)
CONFORMANCES                            // how the technique implements strategies
    getLock CONFORMS TO (validate, register)
    ...
END StrictLockManager
```

Example 2. Technique StrictLockManager

3.3.2 Specification of Primitives

A primitive is a class and is described by the following properties:

- a unique name,
- a formal specification, and
- a documented implementation.

Therefore, for the sake of uniformity, the same formalism as for techniques is used for their specification. See Example 3 for an example.

```
COMPONENT
        LockTable
INCLUDES
        HashTab
        LockEntry < HashEntry
        TAID
        OID
STATE
        locks:HashTab [Key::OID,HashEntry:: LockEntry]
        taLocks: HashTab [Key::TAID, HashEntry::LockEntry]
METHOD
        setLock(ta: TAID, object: OID, entry: LockEntry)
        ...
METHOD
        deleteLocks(ta: TAID)
        ...
METHOD
```

```
getLock(object: OID) : list(LockEntry)
...
```

Example 3. Primitive LockTable

4 The Role of Strategies and Techniques in the Construction Process

In this section, we describe the role of strategies and techniques in the DBMS-construction process and discuss their usefulness with respect to reusability.

4.1 Outline of the Construction Process

In this section we outline the construction process model as far as necessary for understanding the role of strategies and techniques; details can be found in [7].

The first phase covered by a construction process for a DBMS is analysis. This phase determines the functionality required from the DBMS. One major outcome of this phase is the determination of *relevant aspects*. In the second phase, an architecture skeleton is selected for reuse (e.g., the one represented in Figure 1). In a subsequent phase, *subsystem design* is performed. In this phase, strategies are selected for the relevant aspects and dimensions and are applied to brokers. In other words, each strategy selection refines the corresponding broker design. Subsequently, during subsystem implementation techniques are determined that realize selected strategies.

The knowledge on strategies and techniques is stored in repositories. These repositories are used during subsystem design and implementation in a possibly iterative process (Figure 2). Given certain combinations of strategies, the DBI queries the repository for techniques realizing the strategies. If techniques are found (“full match”), he/she can decide simply to reuse the technique. Otherwise, he/she has two possibili-

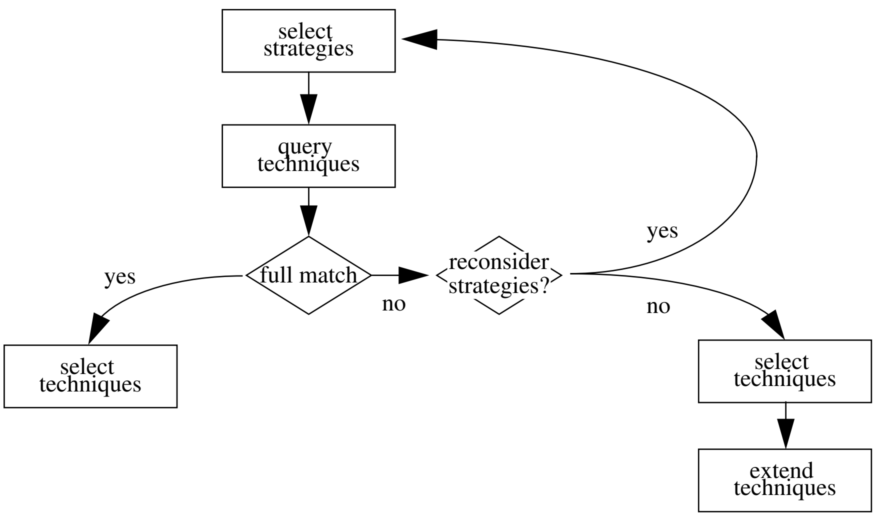

Figure 2. Iterative Selection of Strategies and Techniques

ties: he/she can revoke one or more strategy selections, or develop a new technique. In the second case, she/he can select the "best" technique (the one that realizes most of the strategies) as a starting point, extend this technique, and finally update the repository with the new technique.

4.2 Strategies, Techniques, and Reusability

Concrete dimensions, strategies, and techniques are the results of a domain analysis perfomed before DBMSs are actually constructed. Such a domain analysis can be successively refined and extended as more experience on the domain becomes available. Dimensions, strategies, and techniques are also intended to help the DBI understand the universe of discourse (here, the design and implementation approaches of an domain and their respective relationships with and implications on other aspects).

During the specific phases of the actual construction process, concepts of strategies and techniques are used to assist the DBI in various respects:

- for a given component to be realized, the set of categories represents knowledge on decisions to be made,
- decisions concerning strategies affecting "many" other components and heavily restricting further decisions can be made first, such that backtracking is decreased and the search space is "pruned" early,
- conflicting decisions can be recognized early based on information on incompatible strategies,
- the gap between design and implementation is bridged in that suggestions concerning reusable techniques are made,
- implementation of techniques is supported by reusable primitives.

The proposed approach balances important facets of reuse: *reuse potential* and *reuse payoff*. Reuse payoff denotes the efforts that have been saved through reusing artifacts. It refers to the time it would take to implement the artifact from scratch. Reuse potential denotes the likelihood that a reusable artifact can be found in a specific situation. Both measures are balanced for the following reasons:

- Strategies and techniques reflect information on previous designs (in the case of techniques on a more concrete level). Reuse of design information is considered to be the most promising area of reuse (high reuse potential).
- A refinement approach is applied: starting with partially defined brokers, their behavior is refined in terms of strategies, then further refined in terms of techniques, followed by completing design and implementation by means of primitives. In the case of strategies and techniques the reuse potential is quite high. Reuse payoff is high whenever reusable techniques and primitives can be found.

5 Related Work

Research in DBMS-construction has yielded the concept of "extensible DBMSs" [5]. These systems (e.g., EXODUS [4], GENESIS [1], OpenOODB [15]) all apply reuse techniques, such as generation/transformation or configuration/composition. For certain functionalities, libraries of modules have been proposed in the EXODUS approach [4]. However, this idea has not been followed further, and questions such as the representation and selection of alternatives during a construction have not been answered in EXODUS.

The GENESIS approach has been extended for hierarchical software systems in general [2]. The central notion of this approach is the *component*, where each component is an element of a *realm*. All the elements of a realm have the same interface, but possibly different implementations. Then, a software system is described as a kind of algebraic expression. In this approach, the selection problem is still not solved.

The *Open OODB* approach [15] attempts to support the construction of object-oriented DBMSs. Open OODB distinguishes a meta-architecture and an extensible collection of modules implementing specific functionalities (*policies*). The meta-architecture defines a set of kernel modules, mechanisms to define the system architecture (boundaries between components), and so forth. For some specific functional tasks, various policies can be applied (and can even be exchanged dynamically). Each domain for which multiple policies can be used is controlled by a policy manager, and all the policies of a specific domain are required to guarantee the same invariants (which ensures that they are interchangeable). The notion of policy is similar to our concept of strategy, however in the Open OODB it is still unclear how alternative strategies can be represented and how selection can be supported.

Frameworks [10] and contracts [9] have been proposed as means of design reusability. However, both allow for less abstraction than our concepts of strategies, and the multiple levels of abstraction in our approach are better suited to solve the selection problem.

Ultimately, software reuse has also been a major issue in the ITHACA project [6]. In ITHACA, generic application frames are engineered and reused, resulting in specialized application frames. Specification is supported through the RECAST tool, and composition of components is supported by the VISTA visual scripting tool. Both tools use the software information base (SIB) and its selection tool for retrieving and storing reusable artifacts. The SIB would be a well-suited platform for storing and retrieving information on strategies and techniques. However, when compared to the RECAST/VISTA approaches, dimensions, strategies, and techniques better support the classification, representation, and selection of components whenever a large variety of such alternatives exists for a given domain.

6 Conclusion and Future Work

In this paper, we have introduced the notions of dimension, strategy, technique, and primitive. These concepts are helpful in a construction approach for DBMSs that is based on large-scale reuse. Especially, the concepts foster reuse during the subsystem design and subsystem implementation phases.

Though not stressed in this paper, dimensions, strategies, and techniques are helpful for understanding the universe of discourse (such as concurrency control). As such, a domain analysis results in definitions of dimensions and strategies.

The current state of this work is as follows. An implementation of a strategy information system has been performed on top of an object-oriented database system. A domain analysis has been done for the field of transaction management, and the concepts of strategies and techniques have been used intensively for the concurrency control domain. Authorization, query processing, and access paths are further sample domains where these concepts appear as necessary, since in these domains a large variety of alternative approaches exists, too. Currently, we intend to build a large technique repository, again for the concurrency control domain in the first phase.

We have developed the notion of strategy and technique in the context of DBMS-construction, since this is the area of software engineering we are especially interested in. An interesting question will be whether the concepts can be helpful for the reuse-based construction of other kinds of software systems. Actually, we feel that our proposal is valuable at least for other kinds of *system software*, whenever they are characterized by a large set of alternative implementations for the same functionality. On the other hand, the concepts are possibly less necessary in areas where standard designs and implementations are known (e.g., certain kinds of application systems).

7 References

1. D.S. Batory: *GENESIS: An Extensible Database Management System*. IEEE Trans. on Software Engineering, 1988.
2. D. Batory, S. O'Malley: *The Design and Implementation of Hierarchical Software Systems with Reusable Components*. ACM ToSEM 1:4, 1992.
3. T.J. Biggerstaff, C. Richter: *Reusability Framework, Assessment, and Directions*. IEEE Software, July 1987.
4. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson: *The Architecture of the EXODUS Extensible DBMS*. In K.R. Dittrich, U. Dayal (eds): Proc. Intl. Workshop on Object-Oriented Database Systems, IEEE Computer Science Press, 1986.
5. M. Carey, L. Haas: *Extensible Database Management Systems*. SIGMOD Record 19:4, 1990.
6. M.G. Fugini, O. Nierstrasz, B. Pernici: *Application Development Through Reuse: The ITHACA Tools Environment*. ACM SIGOIS Bulletin 13:2, August 1992.
7. A. Geppert: *Methodical Construction of Database Management Systems*. Doctoral Dissertation, University of Zuerich, Switzerland, 1994.
8. G. Graefe, D.J. DeWitt: *The EXODUS Optimizer Generator*. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, San Francisco, CA, May 1987.
9. R. Helm, I.M. Holland, D. Gangopadhyay: *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*. Proc. ECOOP 90.
10. R.E. Johnson, B. Foote: *Designing Reusable Classes*. Journal of Object-Oriented Programming 1:2, 1988.
11. C.W. Krueger: *Software Reuse*. ACM Computing Surveys 24:2, 1992.
12. B. Meyer: *Object-Oriented Software Construction*. Prentice Hall, 1988.
13. J.M. Neighbors: *Draco: A Method for Engineering Reusable Software Systems*. In T.J. Biggerstaff, A.J. Perlis (eds): *Software Reusability*. Volume I: Concepts and Models. ACM Press, 1989.
14. W. Stacy, R. Helm, G.E. Kaiser, B. Meyer: *Ensuring Semantic Integrity of Reusable Objects* (Panel Discussion). Proc. OOPSLA, Vancouver, Canada, October 1992.
15. D.L. Wells, J.A. Blakeley, C.W. Thompson: *Architecture of an Open Object-Oriented Database Management System*. IEEE Computer 25:10.