ABSTRACT
Courses in Relational Databases largely use a domain-specific design approach different from that used in the rest of the curriculum. Use of the Unified Process, UML, and Design Patterns as a pedagogical approach for Databases can leverage previous student experience with design, make knowledge from Database courses more immediately relevant elsewhere, and create greater continuity across the curriculum. This approach allows issues in logical design and in implementation to be more easily connected with similar concerns in other courses (for example, Software Engineering), and supports greater and easier transfer of design between Relational and Object-Oriented Databases, and between databases and embedding applications.

Categories and Subject Descriptors

General Terms. Design.

Keywords

1. INTRODUCTION
The growing importance of design considerations for large applications, driven by both technical and economic considerations, make design a major issue in all practically-oriented courses in computer science and information technology. A common and standardized approach to design will aid students’ acquisition of both knowledge and concepts. The UML specification language [3, 15] is now the standard design syntax, and the Unified Process [7] the standard approach in software engineering. Most students will have seen UML, sometimes even in 1st- or 2nd-year computer science courses. UML is also migrating to other courses; in particular, recent database texts and technical works (for example, [4, 10]) now include UML as an alternative for, or in preference to, ER and EER notation. Design Patterns [5,8], which capture common idioms, patterns, and guidelines in the software design and development process, are a more recent development [at least in their current formulation], and not yet universal in software engineering texts, but are certain to become so in the next few years. Design patterns are useful in software engineering courses not only for structuring large projects, but also to suggest complex concerns, or to make connections between topics in a few strokes. Examples in [5] also show applicability to other areas, including networking, graphics, and systems programming.

Design patterns can also be of great use in database courses, particularly if they have been used throughout the curriculum. The cost of their introduction is recovered when a pattern simplifies or clarifies an explanation, facilitates communication, or applies to a design challenge, and in addition is amortized across the curriculum. This repeated use also assures that most students come out of the computer science major with a good feel for design and for this suite of tools.

There are several direct benefits for database courses. First, their use results in a more structured and standard project implementation; used together with UML and the Unified Process, the result is a greater design emphasis and better group project interaction, without requiring a simpler project. Second, a wide range of database issues and concepts are readily illustrated and understood using patterns and UML, allowing students to make connections with similar concepts in other courses. Third, use of UML and patterns provides a common vocabulary and notation, allowing exams and problem sets to ask more complex, more precisely-defined, and more open-ended questions, while allowing student resources and guidelines in conceptualizing, formulating, and expressing their answers. Finally, recent work [17] suggests that working with design patterns, even for relative novices, results in better and faster design. Since many of the design improvements will translate naturally to the relational database world, this should result in better database design.

Two caveats: First, we are proposing a pedagogical approach; we are not suggesting that industrial database design immediately switch to a UML/Design Pattern approach (although the translation back to more conventional database notation is relatively straightforward), nor that database implementation be changed to follow an object-oriented paradigm. Second, while this paper examines the use of generic design patterns, these should be complemented by use of database-specific and other domain-specific (for example, transaction management) patterns.
The rest of this paper is organized as follows. In Section 2, we briefly overview design patterns, and in Section 3 show how they occur naturally (and sometimes surprisingly) in database design and system implementation, and suggest how patterns might be used in teaching and testing. (A more comprehensive list will be available in an extended version at http://cs.wpunj.edu/~kuc/dp.html.) Section 4 introduces a sample project and discusses three possible subprojects emphasizing design, table and query realization, and system implementation, respectively, indicating the advantages of using design patterns in the course. Finally, Section 5 briefly discusses related work, and Section 6 gives our conclusions.

2. AN OVERVIEW OF DESIGN PATTERNS

Design patterns provide a standardized approach for specifying the toolbox of recurrent guidelines and patterns that are a major part of a software architecture methodology [18] for object-oriented design. Larman’s GRASP patterns [8] provide general guidelines for assigning responsibilities to classes. In contrast, the “Gang-of-Four” Design Patterns in [5] are building-block patterns that provide design and code scaffolding. We are dealing with building-block patterns.

In their seminal book [5], Gamma, et al., describe a design pattern as the general outline of a solution to a commonly occurring problem that can be reused over and over. A design pattern has four essential elements:

1. a pattern name that allows efficient communication about the pattern—in fact, we will be referring to patterns by name in the balance of this paper,
2. a problem statement that describes in general terms when to apply the pattern,
3. a solution that "describes the elements that make up the design, their relationships, responsibilities, and collaborations," and
4. consequences of applying the pattern, including trade-offs.

Using design patterns often incurs costs in terms of space and time, but increases the flexibility, extensibility, and maintainability of the system. There are three varieties of patterns in [5]: Creational, Structural, and Behavioral.

Creational patterns separate the creation of objects or classes from their use in the system. For example, Builder separates the construction of a complex object from the creation of the simpler components that comprise it. In particular, one creates an interface Builder with a method to build each type of component. Concrete subclasses build components for the specific system. A Director calls the Builder to create the components needed for a complex object.

Structural patterns provide ways to combine classes or objects into larger structures. Many of these patterns provide a uniform interface between a complex object or class structure and the rest of the system. For example, the Adapter pattern converts the interface of a server class to the interface that its clients expect. This allows the developer to easily adapt the system to different server classes. Another structural pattern, Proxy, represents a surrogate that controls access to another object. Proxies have multiple uses, including: allowing requests to a real object in a different address space (remote proxies), caching information about the real object to postpone accessing it (virtual proxies), or checking that the caller has permission to execute a request (protection proxies).

Behavioral patterns deal with the collaboration and communication among objects to achieve some desired behavior of the system. For example, the Observer pattern supports dependent objects: when a principal object changes state (and an appropriate trigger condition is met), its dependent objects are notified and updated automatically. Another behavioral pattern, Iterator, decouples access to an aggregate object's contents from its internal structure. This allows one to vary the structure of the aggregate without having to rewrite code for objects that access its contents, improving maintainability and extensibility.

3. DESIGN PATTERNS IN DATABASES

3.1 Database Design

Design patterns show up in database design in both obvious and surprising ways. Among the more evident examples:

- Command, which supports the undo of operations, can be used together with Proxy to enforce domain and key constraints (including referential integrity) by rejecting or modifying illegal updates. Proxy can also be used for access control.
- Memento, which allows access to old state, can be used (possibly together with Proxy) to model transactions.
- Unions and subclasses in relational databases admit of a number of different table and query realizations, with various tradeoffs. Use of State or Strategy allows a uniform approach to be used in high-level design, deferring the decision or allowing off-line or even online changes in structure. Specific realizations of unions may be instances of various patterns, such as Decorator (shared attributes in base relation, differing attributes decorating the various sets) or Observer again (the union stored in a common table, and the subsets isolated from it).

3.2 Database Implementation

Design patterns can also be used to express concerns in database implementation; this is helpful even if (or especially if) course projects do not touch the implementation layer.

- Proxy and Memento can also be used to model distributed queries and the needed locking.
- Adapter is a natural model for translating user interfaces (whether an interface language such as JDBC [13], forms and reports, or a graphical interface) to the underlying query language. Adapter or Bridge can be used to model the interface between the table-oriented DBMS layer and varying file decompositions and structures in the file system layer. State and/or Strategy
3.3 Relational and Object-Oriented Databases
An important side benefit of this approach is increased portability of design between a relational approach and an object-oriented or a hybrid approach. If EER diagrams and notations are used, transition to an OO approach requires far more work than if UML and design patterns have been employed, where the transition is immediate.

4. DATABASE PROJECTS USING DESIGN PATTERNS
A typical class project in a relational database management course usually involves the following four major steps (sub-projects)—together with review and testing at each stage, and limited phase iteration if required.

- Analyze the database problem and capture the requirements specification in a conceptual model such as Entity-Relationship Model;
- Map the conceptual schema to a logical database model (e.g., relational schema) and make use of various database design concepts (e.g., normalization);
- Create and populate relational tables;
- Create views, stored queries, and ad hoc queries to solve various database problems.

In transferring knowledge between databases and software engineering, the first knowledge to transfer is the design and implementation processes. An appropriate subset of UML and the Unified Process [3, 7, 15] is used to express design and implementation concepts in building the database system from requirements, following the software engineering life cycle. As the steps are followed, design pattern concepts and other software engineering concepts are introduced by the instructor and used by student groups.

Consider for example a project to design a database for a regional repertory theater, a small subset of which will be implemented. (A fuller description is available at http://cs.wpunj.edu/~kuc/dp.html.) Entities include repertory personnel (actors, musicians, directors, technical staff, house staff, and others) and guest personnel, plays, technical assets (costumes, sets, props and lighting), current and pending productions (with directors, casts, and technical assets) and timelines (casting, rehearsal, preview, performance), customers of various types, subscription plans and special events, funding sources, and so on. In addition to the standard key and domain constraints, many others, including scheduling, casting (both temporal and personal), and ticket sales, must be enforced. Ticket purchases, cancellations, and exchanges are handled differently depending on the class of customer and other factors. The target system includes a computer on premises, and a small network hosted at a state university, shared with other regional arts organizations.

There are multiple classes of user, with different permissions, query sets, and user interfaces. Access is through forms and reports, although house and technical staff also use programmed parametric queries. Ad hoc queries may occur, for example in searching for technical assets. Seating status must be available per production and performance as graphics derived from maintained views. There is a special need for undo-able “what-if” updates as production, schedule, subscription and casting alternatives are considered.

In summary, we have the following complications: subsets and unions (disjoint, overlapping, and—in Customer—non-categorical (non-class-based)); maintained views; undo-able operations (from both requirements and constraints, as well as transactions); file interfaces and distribution; security, visibility, and access control; and multiple kinds of (simple and complex) queries and user interfaces, including embedded queries.

From experience, students find it quite difficult to capture many of these distinctions and features in the first two stages of the project, even given substantial interaction with the instructor. Identification of database design idioms and their correspondence with design patterns brings these issues to the forefront, facilitates their solution, and in some cases assists in design-level consideration of alternatives. Particular benefits of design patterns in this example, and possible ideas for milestones in the project, include:

- **Design.** After determining the basic entities, identify and classify instances of subsets and unions. Use UML and design patterns to express different realizations, and evaluate tradeoffs.
- **Table and query realization.** Model table access and query planning using Iterator, Visitor, and Composite. Use OCL [19] together with Proxy, Memento, and Command to model constraint enforcement, or to model support for “what-if” updates. Implement “what-if” updates using subqueries with named results, relation assignment, and conditional flow. Use State and/or Strategy to provide both a “committed” and a “what-if” mode for queries.
- **Implementation.** Use a protection Proxy plus Adapter to model security, visibility, and access control. Assume that other software identifies and validates the current user, and that his/her ID is provided to the DBMS, and further that it can be found among the repertory personnel (but is not super-user). Also assume that rules for visibility and access control are available as meta-information. Implement messages that deny access for invalid queries (via Command), and views (via Observer) that provide filtered information if the query is allowed.

We illustrate our approach using a simplified model of personnel. Performers have a name (first, last, MI, title), a salary, and a status (Available, Unavailable, NotWithCompany) where Unavailable is used for situations such as injury or outside commitments. Performers comprise Actors (with age, sex, and
vocalRange) and Musicians (with, for simplicity, a single instrument).

There are three different standard approaches for implementing this disjoint union in database design—common attributes and differences, unified table with discriminator, and table of references to separate tables—with various tradeoffs. Figure 1 shows the first approach implemented with a non-recursive version of Decorator, while Figure 2 shows the second approach implemented with Observer. Method getName() recovers a printable name from the name record. (There is a simplification in Figure 2, since in reality we need both Actor to be an Observer of Performer, and ActorTable [not shown] to be an Observer of PerformerTable, and likewise for Musician.)

![Figure 1. Performer Specified Using the Decorator Pattern](image1)

An access permissions example, omitted for lack of space, considers the CastableActor view of Performer and Actor provided to the casting director—only Available actors are visible (so status need not be shown), and salary cannot be seen. This view is implemented via Proxy (to redirect the query from Performer and Actor to CastableActor) and Observer (to filter appropriate, up-to-date entries and fields from the base tables).

5. RELATED WORK

Related work falls into two categories: discussion of design patterns in course pedagogy, and use of design patterns in database design. There are large numbers of articles in the first group, many in SIGCSE conferences, or the newsletter. These apply design patterns of course to Software Engineering (as in [8]), but also to the introductory CS sequence (up to proposals to completely restructure the sequence as in [1, 9]) and to courses in data structures and algorithms (see for example [13]), as well as courses devoted to specific applications or devices.

A number of papers suggest use of design patterns for developing or teaching object-oriented databases, but few if any for relational databases. An approach motivated by the Unified Process, software architectures, and design patterns is presented in [2]. Use of a Factory (called a Database Factory) to support differing transaction and query implementations is proposed in [16]. Finally, database-specific and other domain-specific patterns have been developed, following the practice in software architectures. Nock [11] discusses five categories of database-specific patterns (Decoupling, Resource, Input/Output, Cache, and Concurrency), addressing both logical and implementation design; Grand [6] includes patterns relevant to concurrency, transactions, and databases within a broad catalog of (largely pre-existing) patterns.

6. CONCLUSIONS

Design is an intrinsic and critical concern for all application-oriented courses in computing, but most particularly in areas in which application structure is driven by modeling of fuzzy, complex real-world phenomena, such as artificial intelligence, software engineering, and databases. Use of a uniform approach, with a uniform set of tools and idioms, supports better concept acquisition and a more challenging course structure.

It is interesting to view applications courses and design issues in light of software engineering standards. The CMM (Capability Maturity Models) of the SEI [12] classifies software engineering approaches at five levels of maturity/capability: initial, repeatable, defined, managed, and optimized. Shaw and Garlan [18] define the steps in the evolution of an engineering discipline as production and craft, which combine to give commercial, which together with science (theory) gives professional engineering.

With the Unified Process, aided by design patterns and other developments, software engineering, we would claim, supports managed commercial development—the tool base is there, but there is as yet no body of theory. Relational databases have a solid mathematical theory and a good theory of optimization, but, without a well-recognized set of standard tools and idioms, design remains more a craft than a commercial profession [18]. Adding a richer set of design structures, including design patterns, increases replicability, and, in one view, allows a transition through commercial development to engineering. Use of the Design Patterns with UML and the Unified Process throughout the computer science major has equally significant benefits for both the nature and level of (non-theory) courses. More complex problems can be modeled and implemented, and design issues and tradeoffs confronted more directly. Students, working with a common approach that is becoming standard in industry, will develop better abstraction and conceptualization skills, benefiting their future education, professional development, and career opportunities. A database course, with its strong emphasis on modeling of complex problems, offers a natural and effective stage for further presentation and use of these techniques.

7. REFERENCES


