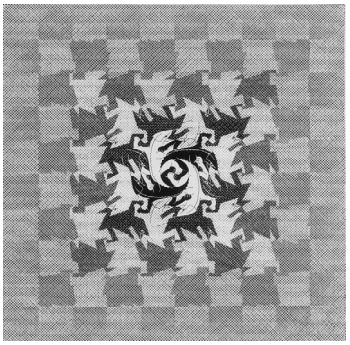

Introduction

Nearly every enterprise application is backed by persistent data. An application's data is often worth considerably more to its users than the application itself. The majority of many systems' resources are dedicated to implementing and managing data access details and logistics. For this reason, it is important to understand both your data's structure and application's interactions with it in detail so you can tune both for efficiency and maintainability.

A thoughtfully designed data model is the best foundation for efficient data access. Normalized databases with properly defined indices result in more efficient queries and updates. More often than not, programmers and designers are not at liberty to alter or redefine existing data models. Changing an aspect of a data model from one software release to the next might require complicated table conversion processes that disrupt an enterprise's computing environment during an upgrade. In addition, your customers are likely to have built their own custom

applications or reports using the data. Changing the data model forces your customers to upgrade these as well. This is likely to deter them from upgrading at all.

Optimizations to data access code are a much more feasible approach, since implementation changes require only the installation of new software. Operations associated with data access can easily be the most expensive in an entire enterprise system. Naive or inefficient data access code makes the difference between streamlined, transparent throughput and slow response times. It follows, then, that data access code is a perfect candidate to target when investigating major optimizations.

Another area to consider where data access strategies pervade is maintenance. Data access code that is spread across many components is difficult to change and optimize, let alone comprehend. Data access code is best left encapsulated within a small number of components so that it is more maintenance-friendly.

The same data access issues tend to arise across a wide variety of application domains. Financial, human resource management, inventory, and customer tracking software can have vastly diverse data models and user interfaces, but still suffer from common data access bottlenecks such as connection overhead, superfluous queries, and concurrency problems.

Just as the same issues appear in many types of applications, developers have solved these problems with common strategies. For example, numerous applications incorporate connection pooling as a mechanism for reducing connection initialization overhead. As this practice became more prevalent and publicized, developers utilized it to improve a growing number of software products. Ultimately, connection pooling has become a common data access optimization strategy supported by many commercial middleware products and defined by standards.

Resource Pool (117) is an example of a data access pattern that has been identified through the abstraction and refactoring of multiple, independent connection pool implementations. Like most of the patterns in this book, it applies to many application domains and platforms. This chapter illustrates how to identify and apply data access patterns and introduces the data access patterns described in the remainder of this book.

APPLICATIONS AND MIDDLEWARE

Creating enterprise software requires two distinct types of expertise. First, developers must be intimately acquainted with the application domain. An *application*

domain includes objects and logic that accurately model business concepts and processes. Creating an effective business application requires you to understand what data is involved, the relationships among the data, and what expert users expect to do with it. It is painfully obvious when an application is not designed by domain experts because it forces users into unnatural paradigms.

The second expertise required to develop enterprise software is the ability to write efficient and transparent middleware. The term "*middleware*" refers to the plumbing between application code and the system. Examples of middleware are system resource libraries, prefabricated graphical user interface (GUI) components, and database drivers. In addition, you can consider any code that you write that acts as a conduit between an application and these libraries to be middleware as well. Writing efficient, maintainable middleware code requires extensive knowledge of database resources and libraries.

These specialties differ in a few regards. Developers with application domain expertise usually attain their skill through industrial experience, while middleware experts learn from professional training and documentation. It is uncommon to find programmers with both skill sets. Most middleware programmers are not able to design a truly marketable business application. Conversely, many business application developers do not write efficient middleware code.

Another difference between applications and middleware is marketability. From the perspective of an application development company, applications generate revenue while middleware does not. Customers purchase applications based on the business processes they implement, so domain features are what makes an application marketable. On the other hand, business customers are less interested in middleware features other than expecting the middleware to be fast and robust. Application vendors invest more in application development than middleware for this reason.

Finally, business functionality is unique among applications. An application's features often focus on processes that are specific to a vertical industry. As a result, application domain code varies significantly. By contrast, middleware code can usually be consistent across all application domains, since all applications require similar middleware functionality.

As middleware developers gain experience working on multiple applications, they understand the common features and add them to their personal bag of tricks. Certain techniques apply to nearly every application domain. Developers refine and reuse these techniques, ultimately abstracting them in the form of reus-

able code or designs. This makes middleware code cheaper and more efficient for each project. As middleware code gets cheaper to develop, software companies can increase their investment in application development, which can in turn lead to higher revenues.

SOFTWARE ABSTRACTIONS

Beginning programmers tend to tackle concrete problems with specific solutions. The resulting code precisely addresses the immediate needs of their customers and requirements. As projects progress and programmers gain experience, they learn how to factor common code and designs into modules.

Modules are a form of *software abstraction*. Software abstraction is the process of making a solution to a specific problem useful in a broader scope. As you abstract a specific solution, you focus on the solution's distinctive characteristics while decoupling it from other components in the system [Booch 1994]. This concentrated focus simplifies complex solutions because it deemphasizes the interactions with other objects. Software abstraction usually leads to more effective and reusable designs and code. Done well, abstractions and reuse significantly reduce overall software development and maintenance costs.

Suppose you have been handed a requirement to cache account information for the purpose of improving a shopping cart application's performance. You start with an empty cache object and add an entry to it every time the application references a new account. Subsequent account references are faster since their information is already available in the cache.

You can approach this problem at any of the following levels:

1. *Hardwiring* is a brute-force approach that solves only the specific problem. You change every aspect of your application code that requires account information to first check the cache and, if necessary, query the database.

2. *Modularizing* involves solving the problem once and utilizing the solution multiple times. You create a common subroutine to check the cache and query the database, if required. Application code calls this subroutine whenever it needs account information.

3. *Generalizing* means designing modules to solve a broader problem set than your specific requirements dictate. In this example, you can define a common cache

lookup subroutine that works with any table and cache, not just those for account data.

4. *Patterns* address design problems across multiple application domains and programming environments. Demand Cache (281) describes a common solution for a class of similar caching problems. The account information cache is a specific instance of Demand Cache.

This list forms a progression of software abstractions. Each abstraction level is potentially more reusable than the previous ones. However, development of abstractions requires a broader understanding of problem domains and predictions of other problems and requirements that are likely to arise in future iterations.

DESIGN PATTERNS

Design patterns provide generic, reusable designs that solve problems at the design level. A design pattern does not define code and is not specific to a given programming domain. Instead, it provides a proven, tested solution for a class of similar design problems. Design patterns also lend common terminology that you can use to make your own designs easier to document and understand [Gamma 1995].

Patterns describe techniques that experts have abstracted from multiple specific solutions. Identifying and understanding patterns provide two essential benefits. First, they introduce effective design strategies to less experienced designers, alleviating them from rediscovering these patterns using trial and error. Second, they attach common names to ideas that you can readily use in conversation, design meetings, and documentation.

Design patterns are not created from theoretical examples. Experienced object-oriented designers recognize that certain object structures and interactions lend themselves more readily to maintenance and reusability than others. Just like architects and civil engineers define broad building concepts that apply at many levels, software designers assemble a set of patterns that they can apply to a variety of domains.

Many seasoned designers have taken the initiative to identify and document patterns. Design pattern catalogs consolidate related patterns in a single reference collection. The most influential design pattern catalog is the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Gamma 1995]. It identifies and describes pat-

terns that solve broad object-oriented design problems. Many other more specialized catalogs exist, targeting domains such as enterprise application architecture [Fowler 2002], J2EE applications [Alur 2001, Marinescu 2002], business software [Carey 2000], embedded systems [Pont 2001], and testing [Binder 1999].

DATA ACCESS PATTERNS

Just as design patterns document solutions to common design problems, data access patterns have a similar role within the field of data access. The data access patterns in this book describe common abstractions for solutions that you can apply directly within your own applications. Some data access patterns are so universally applicable that many commercial products implement them by default. Resource Pool (117) and Object/Relational Map (53) are two such examples. Application servers or plumbing platforms that implement patterns like these save you from building the same infrastructure on your own.

Other data access patterns are less pervasive, but still apply across many application domains. For example, data caching is a common optimization, but you must implement it carefully so that you do not impose more overhead than the physical database operations you are trying to avoid issuing do. In this respect, you must consider specific usage patterns and the nature of the data to be cached. These are characteristics to which a more generic product might have a difficult time adapting.

THE PATTERN CATALOG

This book is a catalog of data access patterns. Each chapter describes a pattern in full and related patterns are grouped into parts. This is only an organizational categorization and has little bearing on the application and interactions of the patterns themselves.

Since it is a catalog, you can read this book in any order. The patterns do relate to each other, and these relations are documented in the "Related Patterns and Technology" section for each pattern. However, these relations do not necessarily act as prerequisites.

Here is a preview of the catalog's organization and its patterns:

Decoupling Patterns

Decoupling patterns describe strategies for decoupling data access components from other parts of an application. This decoupling allows flexibility when choosing an underlying data model, as well as when making changes to the overall data access strategies for a system.

Another essential aspect to decoupling patterns is the data access abstraction they expose to the rest of the system. This abstraction must be sufficiently versatile to expose the appropriate level of data access capabilities. However, it must also be broad enough to make it feasible to plug in alternate data sources and algorithms. All of the patterns in **Part 1, "Decoupling Patterns,"** work toward these goals.

- **Data Accessor** (9)—Encapsulates physical data access details in a single component, exposing only logical operations. Application code maintains knowledge about the underlying data model, but is decoupled from data access responsibilities.
- **Active domain object** (33)—Encapsulates the data model and data access details within relevant domain object implementations. Active domain objects relieve application code of any direct database interaction.
- **Object/relational map** (53)—Encapsulates the mapping between domain objects and relational data in a single component. An object/relational map decouples both application code and domain objects from the underlying data model and data access details.
- **Layers** (75)—Stack orthogonal application features that address data access issues with increasing levels of abstraction.

Resource Patterns

Resource patterns describe strategies for managing the objects involved in relational database access. A substantial amount of relational database access code today employs a standard call-level interface like Open Database Connectivity (ODBC), Object Linking and Embedding Database (OLE DB), and JDBC. Interface standards have matured in recent years, but most retain well-known concepts like database connections, statement handles, and operation processing. The patterns in **Part 2, "Resource Patterns,"** address performance and semantic issues that frequently arise when accessing relational data using a call-level interface.

- **Resource Decorator** (103)—Dynamically attaches additional behavior to an existing resource with minimal disruption to application code. A resource decorator enables the extension of a resource's functionality without subclassing or changing its implementation.

- **Resource Pool** (117)—Recycles resources to minimize resource initialization overhead. A resource pool manages resources efficiently while allowing application code to freely allocate them.

- **Resource Timer** (137)—Automatically releases inactive resources. A resource timer alleviates the effect of applications or users that allocate resources indefinitely.

- **Resource Descriptor** (159)—Isolates platform- and data source-dependent behavior within a single component. A resource descriptor exposes specific platform idiosyncrasies that relate to particular database resources as generic, logical operations, and enables the majority of data access code to remain independent of its physical environment.

- **Retryer** (171)—Automatically retries operations whose failure is expected under certain defined conditions. This pattern enables fault tolerance for data access operations.

Input/Output Patterns

Part 3, "Input/Output Patterns," describes patterns that simplify data input and output operations using consistent translations between relational data in its physical form and domain object representations.

- **Selection Factory** (191)—Generates query selections based on identity object attributes.

- **Domain Object Factory** (203)—Populates domain objects based on query result data.

- **Update Factory** (215)—Generates update operations based on modified domain object attributes.

- **Domain Object Assembler** (227)—Populates, persists, and deletes domain objects using a uniform factory framework.
- **Paging Iterator** (253)—Iterates efficiently through a collection of domain objects that represent query results.

Cache Patterns

Cache patterns address solutions that reduce the frequency of data access operations by storing common data in a cache. These patterns usually cache domain objects rather than physical data. This is an important design point because it optimizes the cached form that the caller requires. The patterns in **Part 4, "Cache Patterns,"** provide structures for generic, efficient, and extensible caching.

- **Cache Accessor** (271)—Decouples caching logic from the data model and data access details.
- **Demand Cache** (281)—Populates a cache lazily as applications request data. A demand cache is useful for data that is read frequently but unpredictably.
- **Primed Cache** (291)—Explicitly primes a cache with a predicted set of data. A primed cache is useful for data that is read frequently and predictably.
- **Cache Search Sequence** (305)—Inserts shortcut entries into a cache to optimize the number of operations that future searches require.
- **Cache Collector** (325)—Purges entries whose presence in the cache no longer provides any performance benefit.
- **Cache Replicator** (345)—Replicates operations across multiple caches.
- **Cache Statistics** (361)—Record and publish cache and pool statistics using a consistent structure for uniform presentation.

Concurrency Patterns

Concurrency patterns address what happens when multiple users issue concurrent database operations that involve common data. Most databases include locking features to help with this class of problem, but customized, application-level solutions can be tuned for specific application semantics and providing better feedback to end-users. The patterns in **Part 5, "Concurrency Patterns,"** describe

strategies for robust, concurrent data access with data integrity preservation being the prime motivating factor.

- **Transaction** (379)—Executes concurrent units of work in an atomic, consistent, isolated, and durable manner. Nearly every database platform supports native transactions.
- **Optimistic Lock** (395)—Maintains version information to prevent missing database updates. Optimistic locking uses application semantics to validate a working copy's version before updating the database.
- **Pessimistic Lock** (405)—Attaches lock information to data to prevent missing database updates. Explicit pessimistic locking often offers better application diagnostics than analogous native transaction support.
- **Compensating Transaction** (417)—Defines explicit compensating operations to roll back units of work that are not part of a native database transaction.

THE PATTERN FORM

Each pattern chapter in this book follows a consistent form. The form serves to formalize various aspects of each pattern's analysis. Every chapter is divided into the following sections, each of which addresses a particular pattern characteristic.

Pattern Name

A pattern's name serves as its most recognizable identifier. You can refer to pattern names in conversation and design documentation to explain which patterns you have employed without requiring a great deal of additional explanation. The convention in this book is to name patterns with short and unambiguous noun phrases. As a quick reference tool, any time a pattern name is mentioned outside its own chapter, it is followed by its page number in parentheses. An example of this is Optimistic Lock (405).

Description

The "Description" section is a sentence or short paragraph that succinctly explains the solution that a data access pattern provides. Use a pattern's description to quickly gauge its relevance to a particular design problem. The inside front cover

of this book lists each data access pattern by name and gives its description as a quick reference.

Context

Engineers do not identify patterns from scratch. A pattern's originators have solved the same problem several times before coming up with a general solution. A pattern's context is a characterization of the problem or class of problems that it solves. This section also includes an illustrative example of the problem.

Applicability

A pattern's applicability is a list of conditions that indicate when its integration into a design is likely to be beneficial. Do not view these items as prerequisites, but rather as general guidelines.

Structure

The static structure of a pattern is illustrated using one or more Unified Modeling Language (UML) class diagrams. The interfaces, classes, and relationships in the diagram represent the pattern's fundamental concepts in broad terms. By no means should you feel obligated to use the precise entity names and relationships that this section defines. They intentionally use generic names to form the basis for subsequent discussion. It is anticipated that you will customize these entities as you apply them to a particular problem.

These terms make up a small naming convention for some of the patterns' structural entities:

- *Client*—The application or caller that consumes the pattern's implementation.
- *Data*—Data in its physical form, as stored in the database.
- *Domain object*—A domain-specific object representation of data.
- *Accessor*—An object whose role within a pattern is to encapsulate data access.
- *Base*—A generic, base implementation of an interface.
- *Concrete*—A specific, concrete implementation of an interface.
- *Factory*—An object whose primary role is to create or resolve implementation instances.

The "Structure" section also describes the role of each participant entity within the context of the pattern.

Interactions

This section describes the interactions among a pattern's participants. In essence, these interactions define how a pattern solves a problem. In some cases, the interactions are illustrated by one or more UML sequence diagrams. Sequence diagrams show the operational flow between participants as they carry out their individual roles. In other cases, this section describes interactions using only text, when that is more succinct than a sequence diagram.

Consequences

The consequences of a pattern describe the positive and negative effects that it can have on the overall system or application. This section contains one or more of each of these consequence types:

- *Benefits*—A pattern's benefits are its positive effects on the overall system. These usually line up with a pattern's description and embody the reason you employ the pattern in the first place.
- *Drawbacks*—A pattern's drawbacks describe its potentially degrading effects on the overall system. Not all drawbacks occur along with every application of a pattern, but they are conditions to keep in mind as you employ it.
- *Tradeoffs*—A pattern's tradeoffs describe considerations involving balancing mutually exclusive benefits and liabilities.

Strategies

This section describes useful techniques for implementing a pattern. It addresses the less obvious aspects of implementation details and offers suggestions for overcoming common problems.

Sample Code

The "Sample Code" section contains examples that usually implement the example scenario presented in the "Context" section. These are not full applications, but rather code blocks that focus directly on illustrating the pattern's key features.

All of the examples in this book use Java, and most of them use JDBC and SQL. JDBC is the standard relational data access interface for Java code, and SQL is the most common language for expressing relational database operations. The patterns in this book are by no means limited to Java, JDBC, and SQL. However, these technologies are widely understood and lend themselves to concise examples. I have made every effort to keep the sample code simple and clearly documented. As a result, you do not need to understand Java syntax details and semantics fluently to read the sample code.

Sample code intentionally omits necessary but uninteresting code details like import statements and connection uniform resource locators (URLs). These omissions are for the sake of brevity and retaining focus on the implementation aspects that relate to the pattern.

Related Patterns and Technology

Every data access pattern relates to other patterns, standards, or products in some way. This section cross-references these relationships. Examples of pattern relationships are:

- *Usage*—One pattern uses another if it builds on its foundation. For example, Domain Object Assembler (227) depends on implementations of Selection Factory (191), Domain Object Factory (203), and Update Factory (215). In a few cases, usage relationships extend across multiple levels. This section only describes direct relationships.

- *Instantiation*—One pattern is an instance of another if it refines its structure. For example, Resource Decorator (103) is an instance of the Decorator pattern described in [Gamma 1995] because it defines an application of it that is geared specifically to database resources.

- *Alternative*—Patterns are alternatives of each other if their solutions can be interchanged with similar results. Demand Cache (281) and Primed Cache (291) are examples of alternatives that can be applied to the same class of caching problems.

- *Cooperation*—Patterns cooperate when they can be applied together to form a comprehensive solution. For example, Cache Collector (325) can cooperate with Cache Accessor (271) to implement a robust caching solution.

APPLYING DATA ACCESS PATTERNS

Reading and understanding patterns exposes you to ideas that may find a place in your own software designs. Applying data access patterns involves several skills. The most important skill is being able to identify an appropriate pattern when an applicable problem arises. In some cases, problems match pattern descriptions exactly and obviously. Others require a certain amount of creativity to notice that a pattern may be specialized in ways other than what the examples in the text describe. Nearly as important is being able to identify when a chosen pattern does not apply to a given design problem. Inappropriate patterns can convolute your design and might force you to build awkward constructs that compromise your design goals.

Practice and a willingness to try and discard solutions are the best ways to attain these skills. Here are some general items to consider as you apply patterns:

- *Be familiar with the patterns*—Read the pattern descriptions enough to understand what each pattern offers. You will not be able to apply a pattern with just this information. However, having a basic familiarity may remind you to investigate certain patterns in more detail when an applicable design or optimization problem presents itself.
- *Refer to patterns in design and code documentation*—A great advantage of documenting patterns is defining reusable nomenclature. Documented patterns give names to general and proven ideas. Refer to any patterns that you apply in your design and code documentation. This gives the reader a place to look to understand your design at a broader level and also saves you from explaining your design in detail.
- *Change entity names to match your domain*—Consider the class, interface, and operation names that patterns describe to be placeholders rather than final names. As you apply a pattern, replace its entity names with those that match your domain. Your code will make more sense to casual readers who are unfamiliar with the pattern.
- *Take liberties with the patterns*—Patterns describe abstract design ideas that have broad application. If a pattern solves a particular problem for you, but does not directly fit into your existing design, then change it. Even if you use parts of a pattern or change its structure or interaction, it still adds value to

your application, and it is still worth documenting. It is always better to keep your design clean than to strictly adhere to a pattern.

- *Do not constrain your design*—Patterns are not a panacea for software design. Pattern catalogs collect and document related patterns, but they are by no means a totality of all solutions. Use patterns when they benefit your design, but do not constrain your design to the exclusive use of patterns.
- *Invent your own patterns*—As you practice applying patterns to design and optimization problems, you may also begin to identify abstract solutions of your own. Take the time to document these as patterns using a form of your choice and refer to them in your design documentation. You can expand on your own patterns in one place and refer to them elsewhere. When you only have one place to document patterns, you are more likely to take the time to explain their concepts in detail.

SUMMARY

Expertise in database technology is diverse because it encompasses data modelling, database administration, interaction with programming interfaces, and understanding many competing technologies. It is rare for a single engineer to specialize in all of these aspects, but it often happens that development organizations grant certain lucky individuals all of these responsibilities.

Database technology changes frequently. Manufacturers of database products face tough competition, and vendors deliver new programming interfaces and features regularly as a means to differentiate their products from others. As a "database expert," you may be expected to keep up on these products and incorporate them into your software when applicable.

One approach you can take to meet this grand expectation is to thoroughly research current technology, choose what works best for your software, and design your software around it. This strategy is effective for your current software release, but it may lead you to make assumptions based on the chosen technology that contradict the incorporation of additional features in subsequent releases of your software. A better strategy is to build your software to be agile. *Agile software* adapts readily and transparently to new requirements and new technology. Even when you cannot predict how rapid technology changes will affect your architecture, you can isolate design decisions based on current technology so that they can

be changed with minimal effect on the rest of the system. This approach is commonly known as future-proofing. The faster you are able to incorporate new, useful database features into your software, the more you can differentiate the software against your competitors.

This book serves as an illustration of this concept. It does not directly depend on the use of any specific, current database technology. Instead, the patterns and code in this book are intentionally generic. The concepts apply across many relational database technologies, because most of these technologies exhibit common characteristics. Consequently, the patterns in this book and other sources enable you to isolate specific design decisions so that you can future-proof the software that you build.