Object-relational impedance mismatch

From Wikipedia, the free encyclopedia

The **object-relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style; particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schema.

The term *object-relational impedance mismatch* is derived from the electrical engineering term *impedance matching*.

Contents

Mismatches

Object-oriented concepts

Encapsulation

Object-oriented programs are designed with techniques that result in encapsulated objects whose representation is hidden. In an object-oriented framework, the underlying properties of a given object are expected to be unexposed to any interface outside of the one implemented alongside the object. However, object-relational mapping necessarily exposes the underlying content of an object to interaction with an interface that the object implementation cannot specify. Hence, object-relational mapping violates the encapsulation of the object.

Accessibility

In relational thinking, "private" versus "public" access is relative to need rather than being an absolute characteristic of the data's state, as in the OO model. The relational and OO models often have conflicts over relativity versus absolutism of classifications and characteristics.

Interface, class, inheritance and polymorphism

Under an object-oriented paradigm, objects have interfaces that together provide the only access to the internals of that object. The relational model, on the other hand, utilizes derived relation variables (views) to provide varying perspectives and constraints to ensure integrity. Similarly, essential OOP concepts for classes of objects, inheritance and polymorphism are not supported by relational database systems.

Mapping to relational concepts

A proper mapping between relational concepts and object-oriented concepts can be made if relational database tables are linked to associations found in object-oriented analysis.

Data type differences

A major mismatch between existing relational and OO languages is the type system differences. The relational model strictly prohibits by-reference attributes (or pointers), whereas OO languages embrace and expect by-reference behavior. Scalar types and their operator semantics are also very often subtly to vastly different between the models, causing problems in mapping.

For example, most SQL systems support string types with varying collations and constrained maximum lengths (open-ended text types tend to hinder performance), while most OO languages consider collation only as an argument to sort routines and strings are intrinsically sized to available memory. A more subtle, but related example is that SQL systems often ignore trailing white space in a string for the purposes of comparison, whereas OO string libraries do not. It is typically not possible to construct new data types as a matter of constraining the possible values of other primitive types in an OO language.

Structural and integrity differences

Another mismatch has to do with the differences in the structural and integrity aspects of the

contrasted models. In OO languages, objects can be composed of other objects—often to a high degree—or specialize from a more general definition. This may make the mapping to relational schemas less straightforward. This is because relational data tends to be represented in a named set of global, unnested relation variables. Relations themselves, being sets of tuples all conforming to the same header do not have an ideal counterpart in OO languages. Constraints in OO languages are generally not declared as such, but are manifested as exception raising protection logic surrounding code that operates on encapsulated internal data. The relational model, on the other hand, calls for declarative constraints on scalar types, attributes, relation variables, and the database as a whole.

Manipulative differences

The semantic differences are especially apparent in the manipulative aspects of the contrasted models, however. The relational model has an intrinsic, relatively small and well defined set of primitive operators for usage in the query and manipulation of data, whereas OO languages generally handle query and manipulation through custom-built or lower-level, case and physical access path specific imperative operations. Some OO languages do have support for declarative query sub-languages, but because OO languages typically deal with lists and perhaps hash-tables, the manipulative primitives are necessarily distinct from the set-based operations of the relational model.

Transactional differences

The concurrency and transaction aspects are significantly different also. In particular, relational database transactions, as the smallest unit of work performed by databases, are much larger than any operations performed by classes in OO languages. Transactions in relational databases are dynamically bounded sets of arbitrary data manipulations, whereas the granularity of transactions in OO languages is typically individual assignments of primitive typed fields. OO languages typically have no analogue of isolation or durability as well and atomicity and consistency are only ensured for said writes of primitive typed fields.

Solving impedance mismatch

Solving the impedance mismatch problem for object-oriented programs starts with recognition of the differences in the specific logic systems being employed, then either the minimization or compensation of the mismatch.

Minimization

There have been some attempts at building object-oriented database management systems (OODBMS) that would avoid the impedance mismatch problem. They have been less successful in practice than relational databases however, partly due to the limitations of OO principles as a basis for a data model.[1] There has been research performed in extending the database-like capabilities of OO languages through such notions as transactional memory.

One common solution to the impedance mismatch problem is to layer the domain and framework

logic. In this scheme, the OO language is used to model certain relational aspects at runtime rather than attempt the more static mapping. Frameworks which employ this method will typically have an analogue for a tuple, usually as a "row" in a "dataset" component or as a generic "entity instance" class, as well as an analogue for a relation. Advantages of this approach may include:

- Straightforward paths to build frameworks and automation around transport, presentation, and validation of domain data.
- Smaller code size; faster compile and load times.
- Ability for the schema to change dynamically.
- Avoids the name-space and semantic mismatch issues.
- Expressive constraint checking
- No complex mapping necessary

Disadvantages may include:

- Lack of static type "safety" checks. Typed accessors are sometimes utilized as one way to mitigate this.
- Possible performance cost of runtime construction and access.
- Inability to natively utilize uniquely OO aspects, such as polymorphism.

Alternative architectures

The rise of XML databases and XML client structures has motivated other alternative architectures to get around the impedance mismatch challenges. These architectures use XML technology in the client (such as XForms) and native XML databases on the server that use the XQuery language for data selection. This allows a single data model and a single data selection language (XPath) to be used in the client, in the rules engines and on the persistence server.[2]

Compensation

The mixing of levels of discourse within OO application code presents problems, but there are some common mechanisms used to compensate. The biggest challenge is to provide framework support, automation of data manipulation and presentation patterns, within the level of discourse in which the domain data is being modeled. To address this, reflection and/or code generation are utilized. Reflection allows code (classes) to be addressed as data and thus provide automation of the transport, presentation, integrity, etc. of the data. Generation addresses the problem through addressing the entity structures as data inputs for code generation tools or meta-programming languages, which produce the classes and supporting infrastructure en masse. Both of these schemes may still be subject to certain anomalies where these levels of discourse merge. For instance, generated entity classes will typically have properties which map to the domain (e. g. Name, Address) as well as properties which provide state management and other framework infrastructure (e. g. IsModified).

Contention

It has been argued, by Christopher J. Date and others, that a truly relational DBMS would pose no such problem,[3][4][5] as domains and classes are essentially one and the same thing. A naïve mapping between classes and relational schemata is a fundamental design mistake; and that individual tuples within a database table (relation) ought to be viewed as establishing relationships between entities; not as representations for complex entities themselves. However, this view tends to diminish the influence and role of object oriented programming, using it as little more than a field type management system.

The impedance mismatch is in programming between the domain objects and the user interface. Sophisticated user interfaces, to allow operators, managers, and other non-programmers to access and manipulate the records in the database, often require intimate knowledge about the nature of the various database attributes (beyond name and type). In particular, it's considered a good practice (from an end-user productivity point of view) to design user interfaces such that the UI prevents illegal transactions (those which cause a database constraint to be violated) from being entered; to do so requires much of the logic present in the relational schemata to be duplicated in the code.

Certain code-development frameworks can leverage certain forms of logic that are represented in the database's schema (such as referential integrity constraints), so that such issues are handled in a generic and standard fashion through library routines rather than ad hoc code written on a case-by-case basis.

It has been argued that SQL, due to a very limited set of domain types (and other alleged flaws) makes proper object and domain-modelling difficult; and that SQL constitutes a very lossy and inefficient interface between a DBMS and an application program (whether written in an object-oriented style or not). However, SQL is currently the only widely accepted common database language in the marketplace; use of vendor-specific query languages is seen as a bad practice when avoidable. Other database languages such as Business System 12 and Tutorial D have been proposed; but none of these has been widely adopted by DBMS vendors.

In current versions of mainstream "object-relational" DBMSs like Oracle and Microsoft SQL Server, the above point may be a non-issue. With these engines, the functionality of a given database can be arbitrarily extended through stored code (functions and procedures) written in a modern OO language (Java for Oracle, and a Microsoft.NET language for SQL Server), and these functions can be invoked in-turn in SQL statements in a transparent fashion: that is, the user neither knows nor cares that these functions/procedures were not originally part of the database engine. Modern software-development paradigms are fully supported: thus, one can create a set of library routines that can be re-used across multiple database schemas.

These vendors decided to support OO-language integration at the DBMS back-end because they realized that, despite the attempts of the ISO SQL-99 committee to add procedural constructs to SQL, SQL will never have the rich set of libraries and data structures that today's application programmers take for granted, and it is reasonable to leverage these as directly as possible rather than attempting to extend the core SQL language. Consequently, the difference between "application programming" and "database administration" is now blurred: robust implementation

of features such as constraints and triggers may often require an individual with dual DBA/OO-programming skills, or a partnership between individuals who combine these skills. This fact also bears on the "division of responsibility" issue below.

Some, however, would point out that this contention is moot due to the fact that: (1) RDBMSes were never intended to facilitate object modelling, and (2) SQL generally should only be seen as a "lossy" or "inefficient" interface language when one is trying to achieve a solution for which RDBMSes were not designed. SQL is very efficient at doing what it was designed to do, namely, to query, sort, filter, and store large sets of data. Some would additionally point out that the inclusion of OO language functionality in the back-end simply facilitates bad architectural practice, as it admits high-level application logic into the data tier, antithetical to the RDBMS.

Here the "canonical" copy of state is located. The database model generally assumes that the database management system is the only authoritative repository of state concerning the enterprise; any copies of such state held by an application program are just that — temporary copies (which may be out of date, if the underlying database record was subsequently modified by a transaction). Many object-oriented programmers prefer to view the in-memory representations of objects themselves as the canonical data, and view the database as a backing store and persistence mechanism.

Another point of contention is the proper division of responsibility between application programmers and database administrators (DBA). It is often the case that needed changes to application code (in order to implement a requested new feature or functionality) require corresponding changes in the database definition; in most organizations, the database definition is the responsibility of the DBA. Due to the need to maintain a production database system 24 hours a day; many DBAs are reluctant to make changes to database schemata that they deem gratuitous or superfluous; and in some cases outright refuse to do so. Use of developmental databases (apart from production systems) can help somewhat; but when the newly developed application "goes live"; the DBA will need to approve any changes. Some programmers view this as intransigence; however the DBA is frequently held responsible if any changes to the database definition cause a loss of service in a production system—as a result, many DBAs prefer to contain design changes to application code, where design defects are far less likely to have catastrophic consequences.

In organizations where the relationship between DBAs and application programmers is not dysfunctional, the above point is a non-issue, and the decision as to whether to change a schema or not is driven by business needs. If new functionality is mandated, and it requires the capture of information that must be persisted, schema enhancements to achieve persistence are the logical first step. Certain schema modifications (including addition of indexes) will also be acceptable if they dramatically improve performance of a critical application. But schema modifications that might serve no purpose beyond making a programmer's life modestly easier would be vetoed if they result in unacceptable design decisions such as denormalization of transactional schemas.

Philosophical differences

Key philosophical differences between the OO and relational models can be summarized as

follows:

- **Declarative vs. imperative interfaces** — Relational thinking tends to use data as interfaces, not behavior as interfaces. It thus has a declarative tilt in design philosophy in contrast to OO's behavioral tilt. (Some relational proponents propose using triggers, stored procedures, etc. to provide complex behavior, but this is not a common viewpoint.)
- **Schema bound** — Objects do not have to follow a "parent schema" for which attributes or accessors an object has, while table rows must follow the entity's schema. A given row must belong to one and only one entity. The closest thing in OO is inheritance, but it is generally tree-shaped and optional. A dynamic form of relational tools that allows ad hoc columns may relax schema bound-ness, but such tools are currently rare.
- **Access rules** — In relational databases, attributes are accessed and altered through predefined relational operators, while OO allows each class to create its own state alteration interface and practices. The "self-handling noun" viewpoint of OO gives independence to each object that the relational model does not permit. This is a "standards versus local freedom" debate. OO tends to argue that relational standards limit expressiveness, while relational proponents suggest the rule adherence allows more abstract math-like reasoning, integrity, and design consistency.
- **Relationship between nouns and actions** — OO encourages a tight association between operations (actions) and the nouns (entities) that the operations operate on. The resulting tightly bound entity containing both nouns and the operations is usually called a class, or in OO analysis, a concept. Relational designs generally do not assume there is anything natural or logical about such tight associations (outside of relational operators).
- **Uniqueness observation** — Row identities (keys) generally have a text-representable form, but objects do not require an externally viewable unique identifier.
- **Object identity** — Objects (other than immutable ones) are generally considered to have a unique identity; two objects which happen to have the same state at a given point in time are not considered to be identical. Relations, on the other hand, have no inherent concept of this kind of identity. That said, it is a common practice to fabricate "identity" for records in a database through use of globally unique candidate keys; though many consider this a poor practice for any database record which does not have a one-to-one correspondence with a real world entity. (Relational, like objects, can use domain keys if they exist in the external world for identification purposes). Relational systems in practice strive for and support "permanent" and inspectable identification techniques, whereas object identification techniques tend to be transient or situational.
- **Normalization** — Relational normalization practices are often ignored by OO designs.

However, this may just be a bad habit instead of a native feature of OO. An alternate view is that a collection of objects, interlinked via pointers of some sort, is equivalent to a network database; which in turn can be viewed as an extremely denormalized relational database.

- **Schema inheritance** — Most relational databases do not support schema inheritance. Although such a feature could be added in theory to reduce the conflict with OOP, relational proponents are less likely to believe in the utility of hierarchical taxonomies and sub-typing because they tend to view set-based taxonomies or classification systems as more powerful and flexible than trees. OO advocates point out that inheritance/subtyping models need not be limited to trees (though this is a limitation in many popular OO languages such as Java), but non-tree OO solutions are seen as more difficult to formulate than set-based variation-on-a-theme management techniques preferred by relational. At the least, they differ from techniques commonly used in relational algebra.
- **Structure vs. behaviour** — OO primarily focuses on ensuring that the structure of the program is reasonable (maintainable, understandable, extensible, reusable, safe), whereas relational systems focus on what kind of behaviour the resulting run-time system has (efficiency, adaptability, fault-tolerance, liveness, logical integrity, etc.). Object-oriented methods generally assume that the primary user of the object-oriented code and its interfaces are the application developers. In relational systems, the end-users' view of the behaviour of the system is sometimes considered to be more important. However, relational queries and "views" are common techniques to present information in application- or task-specific configurations. Further, relational does not prohibit local or application-specific structures or tables from being created, although many common development tools do not directly provide such a feature, assuming objects will be used instead. This makes it difficult to know whether the stated non-developer perspective of relational is inherent to relational, or merely a product of current practice and tool implementation assumptions.
- **Set vs. graph relationships** — The relationship between different items (objects or records) tend to be handled differently between the paradigms. Relational relationships are usually based on idioms taken from set theory, while object relationships lean toward idioms adopted from graph theory (including trees). While each can represent the same information as the other, the approaches they provide to access and manage information differ.

As a result of the object-relational impedance mismatch, it is often argued by partisans on both sides of the debate that the other technology ought to be abandoned or reduced in scope.[6] Some database advocates view traditional "procedural" languages as more compatible with an RDBMS

than many OO languages; or suggest that a less OO-style ought to be used. (In particular, it is argued that long-lived domain objects in application code ought not to exist; any such objects that do exist should be created when a query is made and disposed of when a transaction or task is complete). On the other hand, many OO advocates argue that more OO-friendly persistence mechanisms, such as OODBMS, ought to be developed and used, and that relational technology ought to be phased out. Of course, it should be pointed out that many (if not most) programmers and DBAs do not hold either of these viewpoints; and view the object-relational impedance mismatch as a mere fact of life that information technology has to deal with.

It is also argued that the O/R mapping is paying off in some situations, but is probably oversold: it has advantages besides drawbacks. Skeptics point out that it is worth to think carefully before using it, as it will add little value in some cases.[7]

See also

- Object-relational mapping

References

1. ^ C. J. Date, Relational Database Writings
2. ^ Dan McCreary, XRX: Simple, Elegant, Disruptive (http://www.oreillynet.com/xml/blog/2008/05/xrx_a_simple_elegant_disruptiv_1.html) on XML.com
3. ^ Date, Christopher ‘Chris’ J; Pascal, Fabian (2012-08-12) [2005], "Type vs. Domain and Class" (http://dbdebunk.blogspot.com.br/2012/08/type-vs-domain-and-class.html) (World Wide Web log), *Database debunkings*, Google, retrieved 12 September 2012.
4. ^ ——— (2006), "4. On the notion of logical difference", *Date on Database: writings 2000–2006*, The expert’s voice in database; Relational database select writings, USA: Apress, p. 39, ISBN 978-1-59059-746-0, "Class seems to be indistinguishable from type, as that term is classically understood".
5. ^ ——— (2004), "26. Object/Relational databases", *An introduction to database systems* (8th ed.), Pearson Addison Wesley, p. 859, ISBN 978-0-321-19784-9, "...any such *rapprochement* should be firmly based on the relational model".
6. ^ Neward, Ted (2006-06-26). "The Vietnam of Computer Science" (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx). Interoperability Happens. Retrieved 2010-06-02.
7. ^ *J2EE Design and Development* by Rod Johnson, © 2002 Wrox Press, p. 256.

External links

- The Object-Relational Impedance Mismatch (http://www.agiledata.org/essays/impedanceMismatch.html) - Agile Data Essay

- The Vietnam of Computer Science (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx) - Examples of mismatch problems
- O/R mapping Why/When (http://www.rgoarchitects.com/Files/ormappin.pdf) - An article explaining the tradeoffs of Object-relational mapping

Retrieved from "http://en.wikipedia.org/w/index.php?title=Object-relational_impedance_mismatch&oldid=609847966"
Categories: Object-oriented programming | Object-relational mapping | Relational model

---