Reladomo Philosophy & Vision

Table of Contents

1. Reladomo Philosophy & Vision

1.1. Disclaimer

This document is prepared as a companion document for Reladomo. The opinions expressed in this document pertain expressly to the Reladomo product; as such, they are not meant to be taken as general opinions, evaluations, or considerations regarding other frameworks, libraries, languages, or coding patterns.

1.2. Preface -- or why you should read this

Reladomo is positioned as a framework, not a library. A framework goes beyond what a library provides by being prescriptive and opinionated about what coding patterns fit well, and don't fit well, in conjunction with the framework. Reladomo also generates code (see section Section 1.5, “Development Philosophy”) and the generated API is expected to be used liberally throughout the rest of your code. It is therefore imperative that the framework code and the application code have a unified outlook on what works well and what doesn't.

Reladomo is designed with a set of core values and assumptions about the operating environment. Given those values and assumptions, we consider certain coding patterns to be a good-fit and others a bad-fit. When we mark something as a bad-fit pattern for Reladomo, we are not judging or condemning that practice outright; we simply believe that using a bad-fit pattern with Reladomo will cause long term problems for the code base and should be avoided. Even then, we don't consider it necessary to avoid bad-fit patterns in the short-term, as long as there is a longer term plan to eliminate those. It is best to avoid bad-fit patterns entirely; the latter exception should be used sparingly and mostly for migration from legacy code.

As a user of the framework, this document is meant to allow you to understand the sweet spot for Reladomo. You can evaluate the values, operating environment and coding patterns you like to use against those outlined here and decide whether Reladomo would be a good fit for your use case. Should you decide to use the framework, this document can act as guide to writing code that works harmoniously with Reladomo.

This document also acts as a guide to the Reladomo developers for adding new features and changing the existing code. We will heavily bias our evaluation of new features toward being compatible with the existing values of the framework. We will likely avoid code that would encourage the bad-fit patterns.

1.3. Reladomo Core Values

Reladomo is predicated on five core values. These values are not compatible with every possible use for code; they simply represent the set that underlie the design decisions that embody the code in Reladomo. We will reference these values throughout this document using their short form (in square braces) with either a plus, meaning compatible, or minus, meaning less compatible.

- *Long lived code [LLC]:* We target code that is meant to run in production for a long time, meaning many years or decades. This can often lead to prioritizing practices that may have higher initial cost, but lower runtime cost. We assume that such code will see many collaborators over its lifetime, of varying skill levels and familiarity with the code. Furthermore, we assume code that is long lived will be sizable and possibly grow in complexity over time.

- *Don't repeat yourself [DRY]:* The benefits of DRY are well explained *elsewhere* [https://en.wikipedia.org/wiki/Don't_repeat_yourself]. This value is complimentary to LLC, as it reduces the cost of code ownership in the long run with multiple authors by avoiding transcription errors that occur when information has to be repeated.

- *Agile [AGL]:* in the simple interpretation of being able to respond to change. For a long lived system, the ability to respond to change is important because change is inevitable and systems that can't respond to change won't last long in production.

- *Domain based object oriented paradigm [DOO]:* Application logic is represented by a set of persistent domain objects. We believe this to be consistent with the other values and allow for a harmonious interaction with database systems.

- *Correctness & consistency [CC]:* Data and transactional correctness come first, then consistency, then performance. While we're committed to providing a high performance solution, we will not sacrifice correctness or consistency for that.

1.4. Expected Operating Environment

In developing Reladomo, we've made certain assumptions about what the runtime environment has to be. Some of these assumptions are necessary for ensuring compatibility, others are necessary because of constraints of the JVM, etc.

- We assume the application code is executed in a moderately long running process. Using Reladomo with a process that has the lifetime of "ls" or "cgi-bin" is a bad-fit. The JVM requires time for class loading and JIT optimizations which limit applicability in those realms. Additionally, creating secure connections to databases, running queries and modifying data, etc. are accomplished better when one-time costs are amortized.

- `java.lang` classes work as documented. While this is really all of the Java SE specification, we've found that some environments cannot even provide the functionality in `java.lang`. This includes `java.lang.Thread`, `java.lang.ThreadLocal` and all methods on `java.lang.Class`. Environments that don't support Java SE are considered a bad-fit. There are two reasons behind this choice. First, pragmatically, it would be much harder to write code that works without a specification to rely on. Second, Java's "write once, run anywhere" mantra is highly compatible with our core value of long-lived code [+LLC]. Environments that don't implement Java SE properly reduce the reusability of code, and we prefer not to encourage their use.

- Java's `static` keyword for variables and constants means one per process. While this is not per the Java spec, testing code in environments that violate this assumption is practically impossible. We would have no way of knowing all possible combinations of "static means one or more per process", let alone test

them. Therefore, environments that have complex class loaders and significantly break this assumption are a bad-fit. We recommend Reladomo classes to be loaded from the System classpath loader.

- `private` means private and `final` means final. Environments that randomly or purposefully mutate private or final values are a bad-fit. We can neither reason about, nor test environments that behave as such.

- Thread lifecycle is per the Java specification. Specifically, there is nothing running around in the JVM trying to kill or otherwise disrupt thread execution. Environments that don't support thread execution per the Java specifications are a bad-fit.

- We expect the database to be used as private storage. When the database is not used as a public API or a communications channel, the application is free to make changes [+AGL] without consulting/changing any other code base, or arranging for deployment coordination. When all access is from a singular well tested codebase, changes can be made confidently. With the application logic encapsulated in terms of the objects and domain [+DOO], there should be no need to write repetitious code that delegates computation to the database [+DRY]. Database generated data (identity, triggers, stored procedures, etc.) are considered a bad-fit.

1.5. Development Philosophy

Reladomo's development philosophy is predicated on the core values, operating environment and the needs of an object relational mapper (ORM). As a user of the framework, it's good to be familiar with these, because they can inform your coding decisions beyond the good-fit/bad-fit patterns described later. As a developer or committer for Reladomo, you must ensure that any additions or changes are consistent with this philosophy.

- Object identity must have a sensible contract. In Reladomo, this contract is to represent a persisted object as a singular Java reference. The modified state of that object in a transaction must not be visible outside the transaction. This allows the user to write code properly with ACID (atomic, consistent, isolated, durable) semantics. It also reduces memory footprint and allows for better caching and reasoning about required memory. As part of this contract, we require every row in the database to be uniquely addressable; that is, all tables must have a primary key. Combined, these are well aligned with the long-lived code [+LLC] core value.

- Reladomo must solve the set/object impedance mismatch. Relational databases are constructed with sets as their fundamental unit; operations are implicit over sets. Objects are more graph-like in nature and operations are procedural/imperative. The domain lists in Reladomo are the core feature that bridges this gap. They allow for set-like operations (e.g. `insertAll`, updating all, etc.), as well set-like IO patterns with `deepFetch`.

- Testability is a key requirement for Reladomo itself as well as application code using Reladomo. This is a direct consequence of long-lived code that can remain agile [+LLC][+AGL]. The test resource provided by Reladomo is integral to this capability. It's important that application code using Reladomo be testable using this resource for the majority, if not all of the available features. In particular, user written tests must enable them to perform read-write operations, reason about IO and assert the conditions necessary for a successful execution. Refer to the Reladomo testing philosophy for more details.

- Backward incompatible changes are kept to a minimum [+LLC]. Reasons for incompatible changes include bug fixes, third party library changes, and the occasional design fix. API are first deprecated before being removed, with a removal date set at least one year after the deprecation.

- ACID (atomic, consistent, isolated, durable) is assumed. We will not support storage mechanisms that don't follow ACID. While ACID is the embodiment of our correctness and consistency core value [+CC], we believe ACID is a good-fit paradigm for long lived code of medium or high complexity

[+LLC]. ACID allows local reasoning that is critical to making small code changes over a long time without affecting other parts of the code [+AGL]. We will not allow "read uncommitted" type interactions with the data store. *Brewer* [https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed] points out "the hidden cost of forfeiting consistency, which is the need to know the system's invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are." *Pimentel* [http://www.methodsandtools.com/archive/acidnosqldatabase.php] considers this a "technical debt": "when an application requires concurrent operations, weakened consistency creates a "technical debt" that is pushed onto the developers making use of the data store." Technical debt poses serious challenges for long lived code. Google said in their paper about the *F1 database:* [https://research.google.com/pubs/pub41344.html] without ACID “we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date.”

- In designing our API, we pay attention to IO latency. Our API's should enable proper batch operation and avoid O(n) calls to the storage API. We do not consider bandwidth to be critical on a per-row basis.

- Vendor neutrality is important for long lived code [+LLC] because choice of vendors can change over time. Additionally, codebases that need to transact against multiple vendors should be able to use the same code for all their needs [+DRY]. We aim to support multiple (recent) versions of any particular vendor product, with an option for the application to add minor variations to the supported vendor.

- We construct our API so that the Reladomo objects can be used as the application's main domain representation. No extra code should be necessary to use these objects in such a manner [+DRY]. We believe wrapping these objects to be a bad-fit design, because the API we provide is rich and wide, making any wrapping to be both leaky and repetitive.

- Temporal support must be part of the core API. It's very difficult, if not impossible to build temporal semantics on top of objects that don't support them natively. Many aspects of the API, from the query language, to relationship resolution, to caching, to persistence semantics, etc. are affected by the temporality of the design space. The core data structures for representing temporality are fundamentally different from their non-temporal counter parts.

- A temporal dimension is best represented as two columns in the persistence store, as described by Richard Snodgrass in his *book* [http://www2.cs.arizona.edu/~rts/tdbbook.pdf]. Alternate representations, such a version column, or history tables present too many problems in a consistent view [+CC] of the data, that they are considered bad-fits. A consistent object graph as of a particular point in time is regarded as a critical deliverable of the API.

- Our code generator must work correctly in the context of Reladomo objects used as the applications domain objects. While code generation is a great way to reduce repeated code [+DRY] and apply improvements and fixes to existing code [+LLC], too many code generators don't work well when the application needs to provide behavior for the generated code. We will not generate concrete classes, except as an initial skeleton; only abstract classes will be generated and the concrete classes will be the application's responsibility. We expect to generate code that can be read and debugged by the users, while encapsulating the expert knowledge that makes the code valuable.

- We consider stored procedures to be a bad-fit and will not support them. See Appendix A.

- Our mapping of a class-to-table is predicated on our need to be able to write, cache and provide a reasonable identity contract. Complex transformations between the object schema and the persisted schema prevent that and are therefore considered a bad-fit.

- Our compile-time safe query language provides find-usages, abstraction and refactoring support. We find all those features to be indispensable for long lived code [+LLC]. String based query languages are a bad-fit because they are not compile-time safe, and they offer none of those features.

- Proper caching requires a more nuanced API than a map can provide. Our cache is not a map; it is instead a searchable set, with definable indices and transaction safety. Generic caches, based on map API are therefore a bad-fit.
- Our API is constructed around the persistent domain objects [+DOO]. Reladomo is not a result-set oriented API and we find such API to be a bad-fit.

1.6. Good and bad-fit patterns

Given the core values, operating environment and development philosophy, it should be clear that Reladomo is by no means constructed as a one-size-fits-all solution. The following do's and don'ts will help you use Reladomo without friction and fully leverage its power. You should interpret a sentence that says "Don't do X" as meaning "Doing X is a bad-fit. It might work, but in the context of using Reladomo, there are better ways of accomplishing the same thing."

- Don't assume Reladomo works like any other ORM. Reladomo is constructed differently than other popular ORM's. If you're not certain about the behavior of a piece of code, write a test for it using the `MithraTestResource`, instead of assuming semantics of a different product.
- Study and understand Reladomo's object identity contract. With a persisted object being represented as a singular reference in memory, there is considerable flexibility in what you can do. Understand the transaction isolation of this instance. Leverage non-persisted copies (via `getDetachedCopy` or `getNonPersistentCopy`) in situations where that's desirable.
- Understand the API which allow Reladomo to solve the object-relational impedance mismatch. The `List` objects are key to that, so make sure to use the API they provide. Add methods to the `List` objects that make sense for a collection of your domain objects. Use `deepFetch`, relationship navigation and bulk methods on the List objects.
- Write tests using the `MithraTestResource`. If you're trying to understand IO latency, enable SQL logs in your tests, or write tests that assert on the number of queries being dispatched. Don't write hand-rolled mocks for testing interactions with the persistence API. See the Reladomo Test Resoruce document for more details.
- Update your version of Reladomo regularly. We try to keep the code highly backward compatible, so each upgrade should require little to no changes. In cases of significant changes, we typically deprecate API with a target removal date, rather than outright remove the code, allowing you time to change your code. See the published changelogs for changes to driver details or other significant changes.
- Use the drivers that are tested with Reladomo. Unfortunately, drivers tend to have their share of bugs and issues and using a random version is likely to cause you headache. Additionally, most commercial drivers are not available via open source repositories, so it's not easy to rely on dependency mechanisms. Make sure you don't have multiple drivers on the classpath.
- Write your code using ACID semantics. Be sure to delineate your transaction boundaries correctly. Avoid reading outside the transaction, just to immediately start a transaction and work on the non-transactionally read objects. Reladomo will ensure correctness in such a case, but typically at a cost of re-reading the data for the objects.
- Optimize your code for IO latency. Don't worry about bandwidth; for most (non-columnar) databases, the cost of reading an entire row or a few columns is about the same. Make sure your code is properly batched by using the right API and bundling your unit of work when possible. Use in-clauses with regular or tuple attributes for batch reads. Use Reladomo's ad hoc lists for deep fetching.
- Vendor specific code is considered a bad-fit. Don't write vendor specific code. That includes stored procedures.

- The "surface area" provided by the Reladomo API is large. Do not attempt to wrap this API [+DRY]. Attempting to do so will create a leaky abstraction that's hard to keep abreast of additions and enhancements to the Reladomo API. Reladomo's query API is the right way to find objects. Do not invent your own query language. Most queries naturally belong to a single invocation location in the application code. Additionally, deep fetch needs are highly unlikely to belong to more than one invocation location. Create and use queries at the specific code location they are needed. Use normal OO abstraction if you need to reuse a query. Do not create C-library-like constructs with all queries in one place.
- Use the temporal support provided for Reladomo. Do not invent your own temporal scheme. Temporal support is a core part of Reladomo and is properly represented at all levels of the framework. Rolling your own is a bad-fit.
- Use database views only as temporary, last resort. Views cause problems for identity management, caching and writing. Additionally, many view implementations cause optimization and runtime issues.
- Don't implement the *DAO* [http://www.oracle.com/technetwork/java/dataaccessobject-138824.html] pattern at the application level. This pattern creates a large amount of duplicative code. It's difficult to implement a proper identity contract using this pattern. Navigating relationships also becomes problematic. Use Reladomo objects as your domain objects without any wrappers.
- *Anemic domain model* [http://www.martinfowler.com/bliki/AnemicDomainModel.html] is a bad-fit. Add methods to your Reladomo domain objects and encapsulate the business logic in those methods instead.
- Design your model with both the objects and the persisted schema in mind. Overemphasizing one over the other is not recommended. Be particularly careful with object hierarchies. Prefer composition to inheritance. Avoid complex hierarchies. Follow well established database design principles (normalization, strong typing, etc.).
- Use services as a gateway to your domain logic. Avoid encoding your domain logic inside a service. In other words services should delegate to the domain.
- Complex transformation between the physical schema and application objects is a bad-fit. Such transformations make the object identity contract difficult to create and reason about. Caching and writing also become much harder. Design your schema to avoid transformation.
- Schemaless complex data in your domain is a bad-fit. It violates the first normal form. Not only can this data not be queried properly, the evolution of the format over time can lead to serious complications for long lived code [-LLC].
- Understand the contract of the Reladomo cache. Use "none" cache as a temporary last resort. Don't clear the cache; use Reladomo's notification feature in a multi-process setup. See Reladomo Notification and Reladomo Internal Cache structure documentation for further details.

1.7. Appendix A: Stored Procedures

Stored procedures are in direct conflict with many of Reladomo's core values and development philosophy. A stored procedure is considered a very serious bad-fit. If you want to use Reladomo and you have existing stored procedures, you should convert them to a Java/domain based object oriented paradigm. Here are some of the reasons why:

- When business logic is encapsulated in domain based object oriented code, repetition can be minimized [+DRY]. Stored procedures, on the other hand, tend to create duplicate business logic [-DRY].
- Stored procedures are in direct conflict with Reladomo's vendor-neutrality development philosophy. There is no standard for stored procedures. Porting stored procedures from one vendor to another is usually a complete re-write. Writing a stored procedure is one of the easiest routes to vendor lock-in.

- The parameters passed to a stored procedure pose a significant limitation to proper batching of IO. This can create a bad ping-pong type IO pattern.

- Understanding how an existing piece of code works requires intimate knowledge of both Java and SQL. This problem is exasperated for long-lived code that's editable by many authors of varying expertise [-LLC].

- In a mixed stored procedure/app environment, it's virtually impossible to come up with guidelines regarding where a particular piece of business logic should be implemented. This may lead to multiple implementations of the same logic that diverge over time [-DRY].

- After converting hundreds of stored procedures, we've concluded that the equivalent Java code is faster for 2 reasons: First, in java, it's much easier to use better algorithms. Better collections (maps, trees, sets, bags, etc.) and better language constructs (e.g. polymorphism) help tremendously with this. Second, a stored procedure has a limited number of parameters. This makes it impossible to reduce IO by batching. Batched, multi-threaded Java code can be an order of magnitude faster.

- For long lived code, it's important for the code to communicate its intent [+LLC]. A well written piece of code in Java communicates its intent. A well written piece of SQL typically doesn't.

- There is a wide gap in productivity between the two programming environments.

  - Stored procedures require a live database to develop against. While developing (changing the SP), it's hard to share the database.

  - Automated testing of stored procedures is notoriously difficult. The requirement for a live server per test run is in itself egregious. Overall, this makes the code harder to change [-AGL].

  - Lack of modern debugging facilities (conditional breakpoints, watches, etc.) as well the absence of an integrated debugging facility (break in stored procedures while executing Java code) make development less productive.

  - SQL joins have to be repeated in each query [-DRY].

  - The difference in language semantics is very large. It's hard to do routine tasks (loops, conditionals, etc.).

  - It's hard to modularize stored procedures. Basic units of abstraction are missing (class, method, static method, inheritance, etc.).

  - It's very hard to reuse code. This is easily observable by looking at the number of third party stored procedure libraries available vs. third party Java libraries.

  - Modern OO facilities (refactoring, usage analysis, code coverage, dependency analysis, etc.) are missing.

  - Stored procedures lack basic exception handling (try/catch/finally).

  - How does one make a REST call from a stored procedure or use a data structure like a hash map, or implement a graph algorithm (e.g. Dijkstra's algorithm)?

- There is no notion of compiling/strong typing. Schema modifications often result in failures at runtime. Difficulty of automated testing exasperates this problem.

- Compared to a declared mapping layer (ORM), code that invokes stored procedures is very tightly bound to the database. This makes modern agile approaches much harder to adhere to [-AGL].

- Applications that require highly dynamic, user driven queries are nearly impossible to code with stored procedures