2014 6th IEEE International Workshop on Managing Technical Debt

Managing Technical Debt in Database Schemas of Critical Software

Jens H. Weber
Department of Computer Science
University of Victoria
Victoria, BC, Canada
e-mail: jens@uvic.ca

Anthony Cleve, Loup Meurice
Faculty of Informatics
University of Namur
Namur, Belgium
e-mail: [anthony.cleve|loup.meurice]@unamur.be

Francisco Javier Bermudez Ruiz
Department of Informatics and Systems
University of Murcia
Murcia, Spain
e-mail: fjavier@um.es

***Abstract*—The metaphor of technical debt (TD) has been used to characterize and quantify issues arising from software evolution and maintenance actions taken to modify the functionality or behaviour of a system while compromising on certain "under the hood" quality attributes in order to save cost and effort. The majority of research in this area has concentrated on software program code and architecture. Fewer research considers TD in the context of database applications, particularly TD related to database schemas, which is the focus of this paper. Managing TD in database schemas provides considerable and unique challenges, in particular for applications of safety and security critical nature. We discuss these challenges, point out potential solutions and present an industrial case study in this area.**

I. Introduction

The metaphor of technical debt (TD) has been used to characterize and quantify issues arising from software evolution and maintenance actions taken to modify the functionality or behaviour of a system while compromising on certain "under the hood" quality attributes in order to save cost and effort [1]. The majority of research in this area has concentrated on software program code and architecture. Fewer research considers TD in the context of database applications, particularly TD related to database schemas, which is the focus of this position paper.

We point out that the accumulation, measurement and "down payment" of TD related to database schemas is significantly different from TD in program code or architecture design. A primary reason for that difference lies in the fact that database schemas are intimately coupled with the actual data instances maintained in the application installations and the fact that these data instances are typically not under centralized control and often not even accessible to the software developer.

Analogously to program code and software architecture, there are many different forms of TD to consider in database schemas, including but not limited to data redundancy (similar but not equal to code duplication), weak typing, and missing constraints. In this paper, we focus on referential integrity constraints (RICs) in relational database applications, i.e., constraints that ensure referential integrity between different database tables. RICs can be implemented in relational schemas by declaring so-called *foreign keys* (FK) from one database table to another database table's primary key. Virtually all modern database management systems (DBMS) have the capability of monitoring and enforcing different kinds of integrity constraints, including the validity of FKs. These integrity mechanisms play an important role in preserving and guaranteeing data quality and validity in relational systems.

Notably, the database layer is not the only option for monitoring and enforcing RICs. Other places for ensuring RICs in a typical three-tier architecture include the presentation tier (e.g., implemented in form of JavaScript validation tests run at the user's Web browser) and the business application tier. However, pushing enforcement down to the persistence tier is often seen as the most reliable option (with potentially additional checks at the other tiers), as there can be many concurrent "channels" where data may enter the system and inconsistencies may arise. Moreover, databases provide support for complex *transactions*, which can be used to recover from integrity violations and restore a consistent state.

As a result of the above discussion, we suggest that the absence of FKs that implement "logical" RICs between data items that have been mapped to relational tables should be considered a form of TD. In this paper, we discuss possible reasons for such TD to arise, we suggest ways to measure such TD, and we lay out a process to reduce such TD. We also present large-scale, industrial case study in the context of this research.

The rest of this paper is structured as follows. In the next section, we discuss the question of how TD with respect to missing FKs may arise in a system. This discussion is based on our experiences with empirical studies of existing industrial systems. We will then focus on the question of how to measure FK-related TD in Section III. Section IV presents a process for reducing FK-related TD. We then close the paper with conclusions and a brief discussion of related work.

II. How does FK TD arise?

Before discussing FK-related TD, let us recall the precise definition of a FK in the relational model. Considering a table S with key KS on the one hand, and a set FR of columns of table R on the other hand, FR is a foreign key of R to S if, at any time, for each row r in R, such that $r.FR$ is not null, a row s exists in table S such that $r.FR = s.KS$. In other words, the set of values of FR that appears in table R must be a part of the set of values of KS of table S. The foreign key FR acts as a reference to the rows of S.

Technical debt with respect to undeclared FKs may arise as a result of one of two occurrences in the application

978-1-4799-6791-9/14 $31.00 © 2014 IEEE
DOI 10.1109/MTD.2014.17

development lifecycle: (1) they may arise as a result of incremental modifications of the database application's data model, or (2) they may result from an architectural change, such as a platform migration.

Instances of the first case occur when developers evolve the application's data model without pushing the enforcement of RICs down to the persistence tier. In many cases, developers do not purposefully intend to "cut corners" by omitting to implement the FKs in the database schema. Rather, the omission of declaring FK constraints in the database is often more a reflection of a particular world view, development culture, or philosophy. For example, today's application programmers often see the program code as "the ultimate truth" and the database (schema) as a "necessary evil" that gets generated and used by their code. Such developers may annotate RICs in their application code (e.g., using tags in persistency frameworks such as the Java Persistence Architecture - JPA, Hibernate or other object-relational middleware). However, such developers may not work at the database level directly. In contrast, database engineers have a very different world view; they see the database schema as the "heart" of the system where all data model changes originate from, while programs play the role of peripheral functions.

The second case (architectural change) may occur when an application is migrated from one data persistence model to another, e.g., from file-based or key-value based persistence model to a relational database backend, or when a legacy application is migrated to a new version of a relational DB engine that supports FK monitoring and enforcements. Clearly, depending on the size of the application's data model, such platform changes may result in a large amount of TD. Of course, it can be debated whether the migration to a new platform that supports FK monitoring actually *creates* TD - or whether that TD has existed before. This highlights the question of whether to define TD as *relative* to the choice of technical tools (architecture) employed in a software product (or product line) or whether TD is defined in relation to all possible architectures. We believe the first (relative) definition to be more workable in practice, even if it may allow somewhat non-sensical "solutions" to the problem of reducing TD, e.g., the migration an application back to an architecture that does not support mechanisms for enforcing FKs.

A. Case Study Example

As a real world case study, we consider the OSCAR Electronic Medical Record (EMR) system, a software in production use in hundreds of primary health care clinics in Canada [2]. OSCAR has a relational database backend with a large schema containing over 450 tables, with some tables comprising more than a thousand attributes each. In earlier work we have studied the evolution history of the OSCAR system and noticed the relative absence of FK declarations, even though the underlying DBMS supports FK constraints [3]. In other words, the OSCAR database schema seems to consist (at first sight) of many unrelated tables.

Our investigation showed that one reason for this TD was due to a recent migration from one storage engine to another (MySQL's MyISAM to InnoDB). InnoDB supports FK constraints while MyISAM does not. Moreover, while some of the newer OSCAR developments have resulted in FK declarations in the schema, conversations with the lead developers have indicated a predominantly program code-centric world-view, i.e., developers annotate and monitor RICs and the program code level but do not necessarily see reason for declaring them at the database schema level. Indeed, we were able to detect hundreds of RICs doing program code analysis, RICs that are not reflected in the corresponding database schema [4].

Finally, there is a prevailing culture of caution with respect to "legacy data". Some OSCAR clinics have been collecting data for over a decade. RICs that are enforced at the presentation tier or the application tier will prevent new referential inconsistencies to occur - but they will not cause exceptions with respect to integrity violations in legacy data. This is not the case for RICs that are declared as FK constraints at the database level. The OSCAR community generally tries to avoid updates that may "break" the client's system depending on the quality of the existing legacy data. (For medico-legal reasons such data cannot be simply modified with a "repair" script without manual investigation and approval of a physician.)

III. How to measure FK TD?

Measuring TD with respect to missing FKs is not trivial. A naive approach may be based on FK-detection algorithms developed in the database reverse engineering domain. Indeed many such detection algorithms have been developed, with different types of input artifacts (program code, documentation, schema, data, etc.) and different levels of accuracy. It often requires a combination of several such algorithms to achieve reasonable detection accuracy [4]. However, the result of such a measurement would merely be a crude measure for FK-related TD, as the integrity goals implemented in FKs may have different levels of importance. Complex data models in at-scale applications often have different *regions of criticality*.

For example, the data model of a medical information system like OSCAR has regions that may severely impact patient safety (e.g., diagnosis, medications, allergies, lab results) as well as parts that are less critical (e.g., patient address info, appointment calendar, billing records). Therefore, we argue that a TD measure associated with missing FKs should take into considerations different levels of criticality. Many critical system domains (e.g., medical, avionics, automotive, energy, etc.) have promulgated standards for risk analysis and management that can be used for roughly partitioning the database schema into different criticality regions. For example, we may utilize guidance in ISO 14971 (Medical devices - Application of risk management to medical devices) for partitioning OSCAR's database schema roughly in three regions of criticality with respect to a safety quality goal.

A third factor that should be accounted for when measuring TD with respect to missing FKs is the actual data maintained in a given database. For any given RIC, a system that contains many data instances violating that RIC imposes a higher TD than a system that has no or only very few violations of that RIC. As a consequence of this definition, we observe that the TD for missing FKs tends to increase over time even if the software remains unchanged, because of the possibility of adding data instances that violate the corresponding RIC at the database level.

Equation 1 summarizes the above discussion and defines a measurement for FK-related TD. In this equation, FK is a set of missing FKs, as detected by FK reverse engineering algorithms such as the ones presented in [4]. $cl : FK \rightarrow [0, 1]$ is a function that assigns a *criticality* valuation to each FK. For a given $fk \in FK$ and a given database db, $\#tot_{db}(fk)$ denotes the total number of FK instances of fk in db and $\#viol_{db}(fk)$ denotes to number of instances that violate (the missing) FK fk.

$$?TD_{db} = \sum_{fk \in FK} \left(cl(fk) * \frac{1 + \#vio_{db}(fk)}{\#tot_{db}(fk)} \right) \qquad (1)$$

?

It is worthwhile to note that Equation 1 is a *systems* measure rather than a software measure. In other words, the fact that we consider actual data instances means that the TD measurement may be different from one implementation of the software product to another. Indeed our experience with OSCAR shows that clinics that have been using OSCAR for over a long time have accumulated much higher TD with respect to RICs when compared to new installations of the product. Of course, a software measure can be defined based on the systems measure by averaging over all installations.

However, such an approach is often not practically feasible, as there are often barriers to accessing production data of sensitive informations systems. It is often more practical to provide strictly controlled access to limited, selected installation sites that are used in TD measurement and management. In case of OSCAR, we have access to such a site, which provided us with a hash-encrypted copy of the database instances (approx. 10,000 patients, four physicians). We used this data for FK detection in earlier work [4] and our current work is on computing TD measures for confirmed missing FKs.

IV. How to reduce FK TD?

Reducing TD with respect to missing FKs may require complex changes to not only the database schema, but also the stored data, application programs and even resources that are used for quality assurance (e.g., test code). The process that can be applied for managing these changes during software maintenance releases is simpler for information systems that have only one major deployment (i.e., where there is only a single database to consider). In contrast, if the software is deployed at many customer sites (each having their own database), it is often not realistic to expect that all clients implement FKs "in lock step". Rather, the maintenance process must allow client sites a certain amount of freedom to decide which FKs to implement when. As discussed earlier, the cost of implementing a given FK may be quite different from one customer site to another (e.g., different amounts of TD with respect to "legacy data"). Moreover, since implementing FKs requires cost (effort), different clients may be able to afford this cost at various times.

In the case of our case study, OSCAR is deployed in many hundreds of clinics throughout Canada. Major new product releases are issued only approximately every 2-2.5 years. Maintenance releases are issued much more frequently to fix defects, improve quality and minor system functions. In this paper, we consider a process for reducing FK-related TD within the more frequent maintenance releases. Fig. 1 summarizes this process.

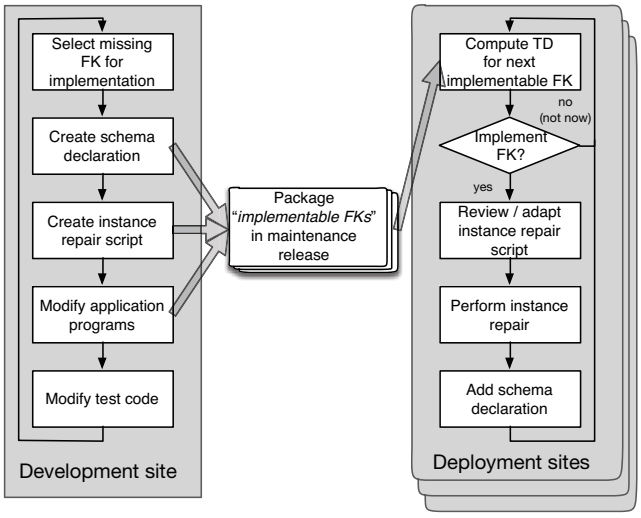

Fig. 1. Process for reducing FK TD

The left-hand side of the figure shows activities carried out at the software development site, which include the following steps:

1) Selection of a missing FK to implement (after prior detection and prioritization). This may involve the computation of associated TD using privileged access to a client deployment site.
2) Creation of a schema modification directive (DDL) for declaring the FK constraint.
3) Creation of a data instance "repair" script for making existing data conform to the new FK constraint. There are multiple principle options of "repairing" data records that violate the missing FK, including the deletion (or archiving) of the offending data record, the creation of a new target record (potentially with default data), the removal of the entries in the FK column of the offending record (if the FK values are optional), etc. The actual choice of these options depends on the nature of the FK and may also depend on the actual data instances to be repaired. The development site prepares a repair script for a proposed FK implementation, which will later on be reviewed, customized and executed at the deployment sites.
4) Application programs may have to be modified as a result of the newly implemented FK. For example, application programs may need to be able to handle new exceptions that may now be raised by the database as a result of integrity violations. An important restriction when adapting the application programs is that the modified programs should still behave properly even if a client site chooses *not* to implement the FK at this time.
5) Test programs may also need modification as a result of the newly implemented FK. For example, existing

tests that generate test data may no longer run because the test data now violates FK constraints. Moreover, there may be performance penalties associated with database constraint monitoring and enforcement, which may require tests of additional quality attributes.

The development site may package a set of FK schema declarations, repair scripts, and associated application code changes into the next maintenance release of the product. We refer to these changes as "implementable FKs" in Fig. 1.

As the right-hand side of that figure shows, client deployment sites can review the "implementable FKs" and should have a means of computing the TD measure with respect to their particular database. They can also review the provided "repair script" and estimate cost involved in implementing the FKs. They can select which FKs they choose to implement and may customize and execute the associated data instance repair scripts. This may require manual intervention or a semi-automatic process. Finally, once the data is conform to the FK, the schema declaration can be added.

V. Conclusion

Referential integrity constraints (RICs) play an important role in assuring data quality in information systems. The most effective way of monitoring and ensuring data integrity is at the database persistence level. In the world of relational DBMS, RICs are implemented by means of foreign key (FK) constraints. Legacy information systems often miss FK constraints, because of various reasons. Mature research exists in the database reengineering community on how to *detect* missing FKs. However, the actual process on how to manage the implementation of detected, missing FKs is less explored - and often causes significant problems in practice. This position paper provides a first attempt of utilizing the technical debt (TD) analogy for developing processes related to missing FK implementation. We discussed the detection of missing FKs, the proposed a measurement for the TD associated with missing FKs, and outlined a process for reducing FK-related TD. We also illustrated our concepts with a real world case study in this area, which we are currently using to gain empirical results confirming the feasibility of our proposed approach.

VI. Related work

The analogy of *technical debt* (TD) has received significant attention and spurred different lines of investigation in the software engineering community [1]. Our focus is on database applications and, specifically, on TD related to referential integrity constraints (RIC) in database schemas. Our work is therefore closely related to works carried out in the database reengineering community [5]. While foreign key (FK) detection and semantic interpretation has been well studied in that community (e.g., [6], [4]), fewer research extends to processes for managing the implementation of missing FKs. However, our experience with complex, real world database applications such as OSCAR indicates that such a process is a critical factor for positive change.

The work in this paper is also related to earlier research of the evolution history of the OSCAR EMR system, that provided us with answers about the provenance of the current schema constructs and the reasons for made observations, such as missing FKs [3].

References

[1] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[2] J. Ruttan, *The Architecture of Open Source Applications, Volume II*. Lulu.com, 2012, ch. OSCAR.

[3] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber, "Understanding database schema evolution: A case study," *Science of Computer Programming*, no. 0, pp. –, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313003092

[4] L. Meurice, F. J. B. Ruiz, J. Weber, and A. Cleve, "Establishing referential integrity in legacy information systems - reality bites!" in *Proc. of ICSME2014-ERA Track (submitted)*, 2014.

[5] J.-L. Hainaut, J. Henrard, V. Englebert, D. Roland, and J.-M. Hick, "Database reverse engineering," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 723–728.

[6] C. Marinescu, "Discovering the objectual meaning of foreign key constraints in enterprise applications," in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*. IEEE Computer Society, 2007, pp. 100–109.