Industry Case Study: Design Antipatterns in Actual Implementations

Understanding and Correcting Common Integration Design and Database Management Oversights

Mihaela Iridon
Cândea LLC
Dallas, TX, USA
e-mail: iridon.mihaela@gmail.com

Abstract—**Prototyping integration points with external systems and new technologies is an excellent starting point for validating certain design aspects but turning that into a complete enterprise solution goes far beyond implementing a working passthrough prototype. In some instances, the focus on functional features and tight deadlines lead to inadequate attention placed on non-functional system attributes, such as scalability, extensibility, performance, etc. Many design guidelines, best practices, and principles have been established, and antipatterns were identified and explained at length. Yet, it is not uncommon to encounter actual implementations suffering from deficiencies prescribed by these antipatterns. The first part of this paper discusses Leaky Abstractions, Mixing Concerns, and Vendor Lock-in antipatterns, as some of the more frequent offenders in case of system integration design. Ensuing problems such as the lack of proper structural and behavioral abstractions are revealed, along with potential solutions aiming to avoid costly consequences due to integration instability, constrained system evolution, and poor testability. The second half of this industry case study shows how unsuitable technology and tooling choices for database design, source code, and release management can lead to a systemic incoherence of the data layer models and artifacts, and implicitly to painful database management and deployment strategies. Raising awareness about certain design and technological challenges is what this paper aims to achieve.**

Keywords-integration models; design antipatterns; leaky abstractions; database management.

I. Introduction

Translating business needs into technical design artifacts and choosing the right technologies and tools, demands a thorough understanding of the business domain as well as solid technical skills. Proper analysis, design, and modeling of functional and non-functional system requirements is only the first step. A deep understanding of design principles and patterns, experience with a variety of technologies, and excellent skills in quick prototyping are vital. Although conceptual or high-level design is in principle technology-agnostic, ultimately specific frameworks, tools, Application Programming Interfaces (APIs), and platforms must be chosen [1]. Together they enable the translation of the design artifacts into a well-functioning, efficient, extensible, and maintainable software system [2].

Designing a solution that targets multi-system integration increases the difficulty and complexity of the design and prototyping tasks considerably, bringing additional concerns into focus. Identifying integration boundaries and how data and behavior should flow between different components and sub-systems, maintaining stable yet extensible integration boundaries, and ensuring system testability, are just a few of such concerns. This paper intends to outline a few design challenges that are not always properly addressed during the early stages of a project and which can quickly lead to brittle integration implementations and substantial technical debt.

A few recognized design antipatterns and variations thereof are explained here, including concrete examples from actual integration implementations as encountered on various industry projects. Solutions to refactor and resolve these design deficiencies and issues are recommended as well.

Section II presents a simplified perspective of a typical system integration problem. It explains a few general-purpose integration concerns and goals, and how these can help to guide the design of the overall integration solution topology and the underlying componentization boundaries.

Section III will address architectural and integration modeling concerns, focusing on a couple of design antipatterns. The structural aspects discussed in this section range from low granularity models (i.e., data types which support the exchange of data between systems) to large-grained architectural models (i.e., system layers and components). The consequences of designing improper layers and levels of abstractions are outlined, followed by recommendations on how to avoid such pitfalls by refactoring the design accordingly.

In Section IV, antipatterns covered in Section III are extended to the design of the data models and relational databases, also discussing the ability to customize external open-sourced systems that participate in the integration. Additional antipatterns are discussed, the problems behind them, as well as potential solutions that can overcome them.

In Section V, the focus is shifted to the management and delivery of the data layer components and artifacts, as databases are an integration concern that goes beyond the data exchanged between the application tier and the data tier. This section intends to explain how the choice of tools and frameworks can have a significant impact on the overall realization, management, and delivery of a robust and consistent integration solution.

Finally, Section VI summarizes the integration design and database management concerns and issues and the accompanying recommendations presented in this paper.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

II. High-Level Overview of General Integration Objectives

From an architectural perspective, a given system that realizes a variety of features of its own may be designed around one or perhaps a combination of architectural styles, such as N-layered, service orientation, component-based, etc. However, when certain features rely on services or data provided by some external system or systems, employing them properly and efficiently becomes an integration requirement that must be carefully analyzed, designed, and realized.

As a general principle, a software system's quality attributes, such as extensibility, performance, testability, and maintainability, to name a few, should always be targeted by design, achieved, and continuously safeguarded. Casually bringing into the integrating system external concerns, data and behavior along with specialized technologies, libraries, frameworks, and tools, could potentially lead to a variety of problems that are difficult to resolve later.

To better understand the reasons behind this statement, Figure 1 shows an integration approach where the system on the left is integrating with a variety of targets (on the right) that provide some needed functionality. Perhaps the integration targets are added over time, one by one, as new overall features are supported. Quick, ad-hoc integration implementations, facilitated by easy access to service endpoints, APIs, and data, can lead to patchy and brittle solutions, where components from different layers of the current system become riddled with - and directly dependent on - the data, behavior, and technologies of the targeted systems. Furthermore, in some cases, even data and behavior of the integration system may leak into the external systems, if these are accessible for customizations, for example. This bleed of concerns and technology between systems is depicted by the various tiny geometric shapes in Figure 1.

With such an approach, future updates to the integration dependencies (shown as hashed geometric shapes) involve code changes throughout the integrating system, risking the overall system's integration stability, as well as potentially its performance, scalability, testability, and evolution.

Ideally, proper design of the integration points would identify new component(s) where integration concerns would be bounded to – as shown in Figure 2 and discussed in [1]. Features, data, and functionality imported from external systems would be exposed to the integrating system via interfaces/contracts that are vendor- and technology-neutral.

Adding such a layer of abstraction (denoted here as the Integration Layer) around the integration points will not only ensure a robust integration solution, but also the ability to easily swap targeted platforms in case of a product replacement (avoiding vendor lock-in), or for independent component and load testing of the integrating system, where the integration targets' features are simulated or mocked.

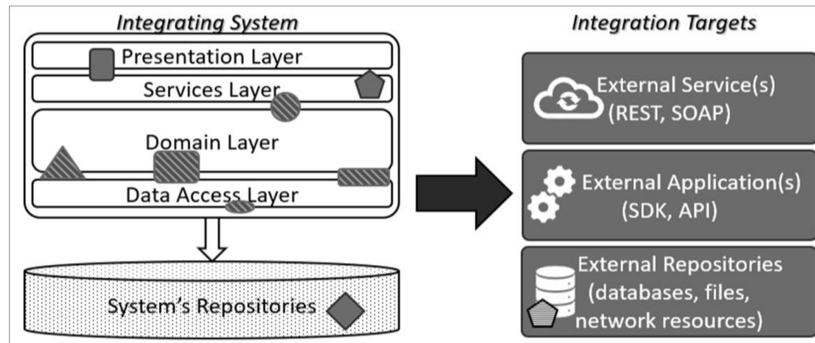

Figure 1. An unsuitable integration solution with external system concerns and technology bleeding into the integrating system (left)

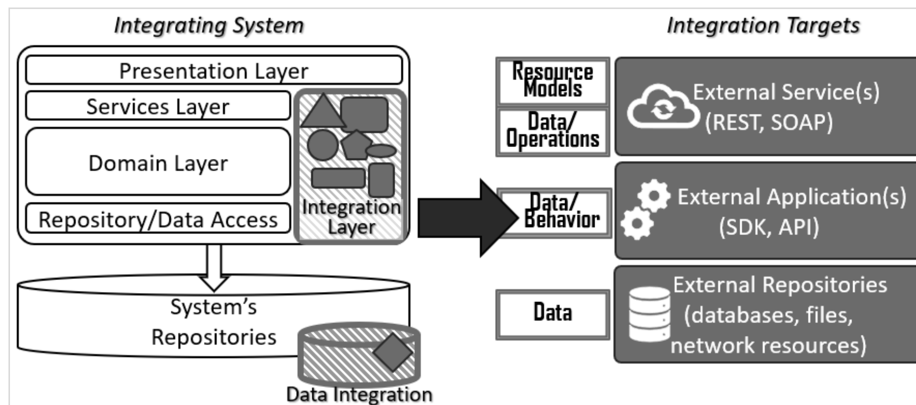

Figure 2. A fitting integration solution with a well-defined integration boundary that isolates external concerns from the rest of the system

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

III. Tight Integration: Leaky Abstractions and Vendor Lock-in

A. The Problem Definition

Let us consider some defined business requirements for building a software system targeting to integrate with - and consume - a third-party service. The exposed data transport models, e.g., REpresentational State Transfer (REST) models or Simple Object Access Protocol (SOAP) data contracts, are already defined, maintained, and versioned by some external vendor or entity (the service provider). Note that this scenario can easily be extended further, to integrations with an arbitrary system by means of some third-party APIs that expose specific behavior and data structures.

Focusing on the data structures rather than behavior, once the service model proxies have been generated via some automation, they tend to become part of the design artifacts for the rest of the system. Their use extends beyond the point where they are needed to exchange data with the external application. These models will percolate throughout the various layers and components of the integrating system. It is not unusual to see development efforts proceed around them, with application and business logic rapidly building on top of these data types. Development costs and tight deadlines, and sometimes the lack of design time and/or technical expertise, are the main reasons leading to this undesirable outcome.

Models exposed by external vendors were not designed with the actual needs of other/integrating systems in mind. External models are characterized by potentially complex shapes (width: number of exposed attributes or properties; depth: composition hierarchy). They cater to most integration needs ("one size fits all"), so they tend to be composed of an exhaustive set of elements to be utilized as needed.

Moreover, allowing these structural characteristics to seep into the application logic layer, beyond the component that constitutes the integration boundary, introduces adverse and unnecessary dependencies to external concerns. Therefore, the system is now exposed to structural instability and will require a constant need to adapt whenever these externally derived models will change. The integration boundary is no longer a crisp and well-defined layer that can isolate and absorb all changes to the external systems – speaking from a data integration perspective.

B. The Antipatterns

The lack of proper structural abstractions and allowing integration concerns to infiltrate into the integrating system is a costly design pitfall and is in fact a variation of the "Leaky Abstractions" problem – as originally defined by Joel Spolsky in 2002 [3]. Such deficient abstractions can be identified not only relative to structural models, but also to behavioral models, which could expose the underlying functional details of the software components to integrate with. This will inevitably lead to increased complexity of the current system, jeopardizing its extensibility and its ability to evolve and to be tested independently. Ultimately this results in a tightly coupled integration between the two systems (with strong dependencies on the target of the integration).

Another perspective or consequence of the problem described is an imposing reliance on vendor-specific technologies, their libraries, and even implementations. This problem is also known as the "Vendor Lock-In" antipattern [4]. External system upgrades will necessitate system-wide changes and constant adjustments on the integration side and will impact the overall stability of the system and the integration solution itself.

Examples range from adopting dedicated libraries for various cross-cutting concerns (logging, caching, etc.) to domain-specific technologies (telephony, finance, insurance, etc.). Vendors will encourage integrators to infuse their

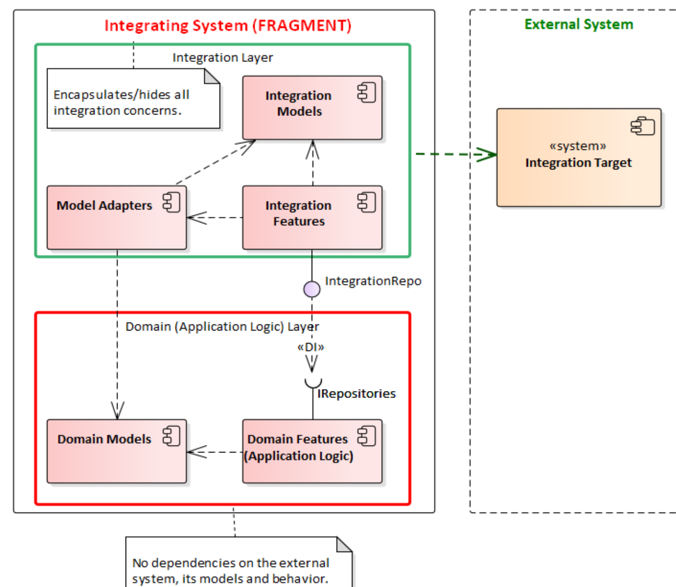

Figure 3. Integration components and the Integration Layer (Adapter) isolating and decoupling the integrating system from the external system

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

specialized technology everywhere, leading to entire (sub)systems taking on pervasive dependencies on their technologies, making it difficult to isolate or replace it. Such third party-entwined architectures must and can be avoided with added effort during the design phase, as described next.

C. The Solution

To avoid such scenarios, the design must unambiguously identify the integration boundary and define custom integration models that abstract away any and all structural and behavioral details related to the system targeted for integration. This architectural approach is exemplified in the component diagram in Figure 3. The integration layer should also hide the underlying technology (REST vs. SOAP, message bus vs sockets, etc.) to avoid tight and unnecessary dependencies. An example of defining canonical models based on the "ubiquitous" integration language in case of multi-system integration is presented in "Enterprise Integration Modeling" [5].

Based on the author's experience, designing proper model abstractions proved extremely useful in the case of building custom integrations with real-time systems. For example, Session Initiation Protocol (SIP) soft switches used in telecommunications networks, such as those from Genesys, the leader in customer experience, pertaining to contact center technology (call routing and handling, predictive dialing, multimedia interactions, etc.). In this case, an extensive array of data types, requests, events, etc., are made available to integrators as part of the Genesys Platform SDKs [6]. These facilitate communication with the Genesys application suite – which in turn enables integration with telephony systems, switches, IVR systems, etc. Most of these data types are very complex and heavy, and introduce acute dependencies on the underlying platform, exposing many implementation details as well. Employing code generation and metadata inspection via reflection, for example, simpler connection-less models were designed to mimic and expose only the needed structural details and are currently used in several production systems. Furthermore, defining and realizing the proper architectural isolation layers will ultimately provide independence from vendor-specific platforms for the rest of the system. For example, considering the integration scenario mentioned above using Genesys' Platform SDK shown in Figure 4, recently the company (Genesys) has been pushing for a new approach to integrate with their systems, specifically using the Genesys Web Services (GWS) [7], a RESTful API. From an integration viewpoint, this substitution is practically equivalent to switching to a different vendor, as the two integration facilities are based on different technologies (web calls versus direct socket connections) and using completely different models, from both a structural model perspective as well as behavioral and consumption views.

Building an explicit and clean integration layer as shown earlier in Figure 3, when dealing with such a significant change (vendor or technology replacement), implementation

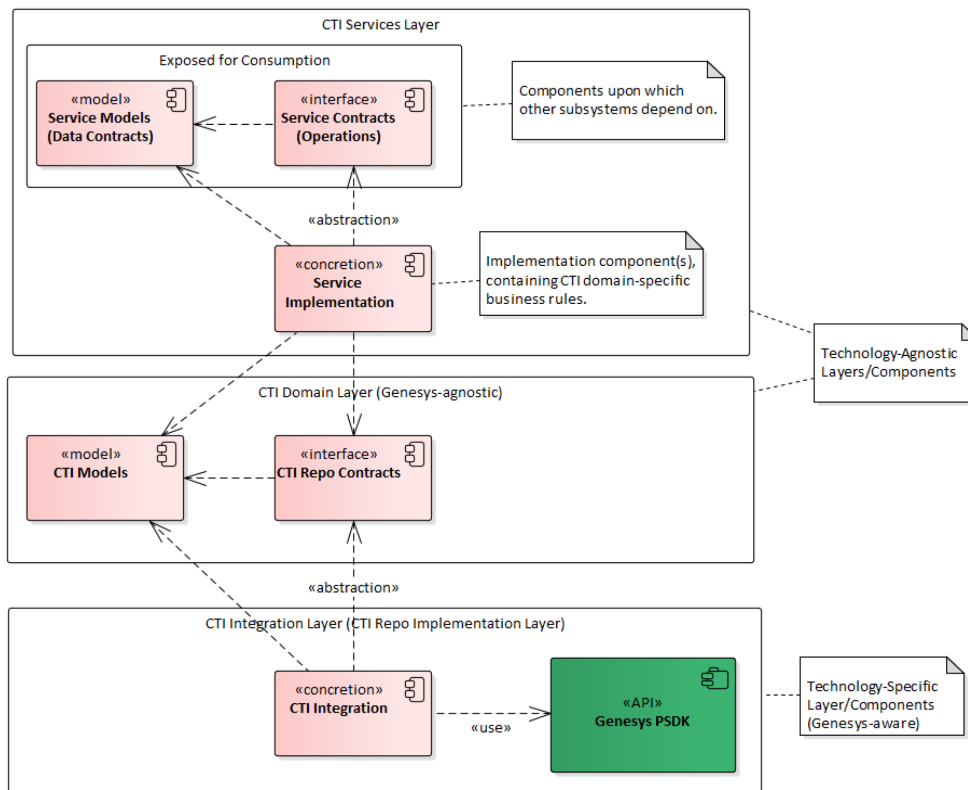

Figure 4. A sample layered architecture for exposing Computer Telephony Integration (CTI) features via integration with Genesys Platform SDK

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

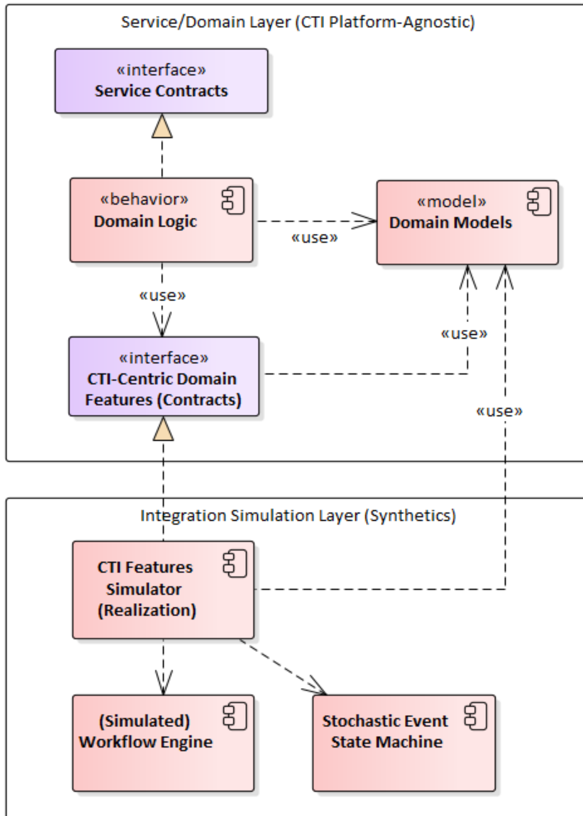

Figure 5. A concrete example of an integration architecture where the integration layer is replaced by components that simulate the integration target's functionality for testing purposes

adjustments will be isolated to this adapter layer without any impact on the business domain layer of the integrating system (assuming similar data and functionality). This includes the specifics of the technology used to communicate between the two systems.

Finally, it is noteworthy that four out of the five SOLID design principles [8] substantiate and drive towards the proposed solution:

- Single Responsibility (SRP), from the component and layering perspective,
- Open-Closed, to avoid changing the underlying implementation every time the integration endpoints change,
- Interface Segregation, exposing only the necessary data types for consumption by the business logic layer,
- Dependency Inversion, where the Domain does not directly depend on the external system, its data and behavior, but rather on abstractions – the repository contracts realized by the integration layer.

D. *Added Architectural Benefit*

Proper design and isolation of the integration components and the use of interfaces and model adapters will enable adequate testing of the custom system without demanding the availability of the external system for integration testing until most defects within the custom system are resolved.

Furthermore, this design approach supports building synthetics that simulate or mock the data and behavior of the external system, providing the means to prototype and test the integration points and functional use cases. This is exemplified in Figure 5, describing at high level a real implementation of a simulation subsystem intended to synthesize the behavior of Genesys' Statistics Server employed in a concrete integration solution.

Even if only a reduced set of features is synthesized, deferring the needs for actual integration testing can be cost-effective, especially in situations where the external system is a shared resource, perhaps expensive to manage and to access in general. Employing Dependency Injection (DI) [9], either the real or the mock implementation of the integration contracts can be injected into the Domain layer, making it easy to swap between the two implementations.

IV. Data Tier Design and Data Access Anitpatterns

One of the most common system integration use cases for many enterprise applications is related to data persistence and access. Integration with (relational) databases that are either part of the custom system or accessible (co-located) components of a third-party system is a pervasive requirement, whether the data tier is needed for storing configuration data, audit/logging, security-related aspects, or to support concrete operational or reporting needs.

This section focuses on several issues related to both database design as well as accessing the data itself.

A. *'Inverted' Leaky Abstractions in Data Integrations*

1) *The Problem*

The previous section discussed Leaky Abstractions that result from allowing third-party concerns to infiltrate custom systems when designing and implementing an integration solution. The directionality of the "leak", as described earlier, is from the external system into the current one. However, it is also possible to encounter the reverse scenario when the integration target is open or transparent to the integrators who then take advantage of this fact to develop and apply their own customizations onto the external system.

Here are two examples:

(a) An Original Equipment Manufacturer (OEM) and/or White Label license of the external system is available to integrators, including access to source code for additional customization and integration options.

(b) The external system contains database(s) accessible to the integrators, either deployed on premise or in a cloud environment, and is open/accessible to change.

In the first example, the same issues and solutions apply, as already discussed in the previous section, only this time from the perspective of the external system. If customization design is not executed properly, software upgrades of the open-sourced third-party system will result in continuous maintenance, or worse, breaking the custom code. Both scenarios will incur high development and system integration testing costs, among other problems.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The rest of this sub-section will focus on the second example, involving third-party databases that are accessible (i.e., open to modification) from an integration and customization perspective.

When expecting and relying on continuous upgrades and patches supplied by the vendor of the external system, it is possible that custom database artifacts (added by the integration provider) will have to be discarded and reapplied, or worse, no longer compatible with the updated system. Moreover, management of database source code targeting the customizations is more difficult if tightly dependent on the elements defined by the external entity/vendor. For example, the custom integration requirements demand two new columns on one of the third-party database tables.

Evidently, with respect to customizations of third-party components (database or otherwise), "Vendor Lock-in" is the status quo as a business-driven need and not a concern here.

2) The Solution

There are several options available and their applicability depends on concrete scenarios and business needs. Ideally, a separate, custom database could be considered, where data collected by the third party system (stored in their databases) would be Extracted, Transformed as needed, and Loaded (ETL) [10]. Detached custom data models are easy to maintain, modify, and version-control by the integration provider. Aligning with the arguments stated in Section III, this approach enforces a well-defined data integration boundary, as shown in Figure 6 below.

Allowing for independent provisioning and evolution of both data models (one provided by the external system and one specifically designed for - and consumed by- the integrating system) will lead to improved extensibility, scalability, performance, testability, and maintainability. With this approach, upgrading the external system will potentially require updating the ETL artifacts and, if needed, some enhancements to the custom database – but both activities can be done in a detached, self-contained fashion.

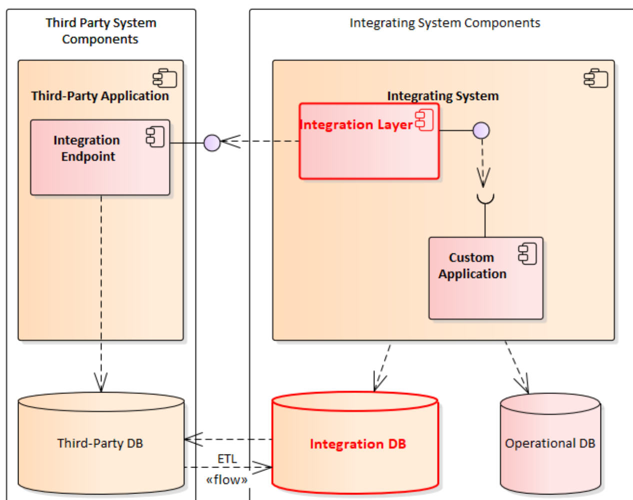

Figure 6. The integration database added to support data integration customizations and to remove direct dependencies on the third-party database

Further details regarding the management of database artifacts will be discussed later, but one noteworthy benefit here is the freedom from having to maintain (a) partial custom database artifacts (divorced from their context) and/or (b) complete external database artifacts (since the database is a self-contained software system, and should not be divided further into sub-components). The reason why maintaining select/partial database artifacts is undesirable is that from a specification perspective, a database (meaning all its defining artifacts) must be valid, consistent, and complete (as it must also be from a deployment perspective).

If database customizations *must* live in the same database as the one that is part of the external system (perhaps for performance considerations), a less optimal solution to the Inverted Leaky Abstractions (i.e., the data model), is to expend proper design effort to minimize tight dependencies and attempt to follow - as best as possible - the Open-Closed design principle at the data tier, in the context of system integration and customization.

For example, if the custom integration components require the persistence of new attributes (fields) in addition to the data captured by the external system, rather than modifying the existing third-party tables by adding new columns, association or edge tables should be considered instead, with custom data residing in new, custom tables. Custom views, parameterized or otherwise, should be designed to transform data into a ready-to-consume format (for operational, reporting, or analytical needs).

In this case, the system quality attributes mentioned earlier must however be carefully monitored, especially query performance and scalability.

On the downside, database code management will become either (a) fragmented/isolated, by extracting the custom database artifacts from the rest of the database into independent scripts, or (b) more complex, by importing the entire third-party database under source control along with the custom artifacts, in order to preserve its integrity.

Section V discusses tools that help validate the full database, warning about invalid or broken object references, binding and syntax errors, thus increasing the probability that database deployments will succeed.

B. Mixing Data Modeling Concerns

1) The Problem

Regardless of the targeted Database Management Systems (DBMS) technology, designing the conceptual and logical data models is a prerequisite to the implementation of the physical data models [11]. Beside ensuring that all data elements outlined by the business requirements are accurately represented, non-functional requirements, such as performance, scalability, multi-tenancy support, security (access to data), etc., will also shape the data architecture.

From an application perspective, the database is used to persist the state of the business processes supported by the application, i.e., operational needs, and to support analysis and reporting needs around the stored business data. The concept of Separation of Concerns (SoC) applies here as well but is often ignored or inadequately addressed. Operational versus reporting concerns are often mixed and data models

designed specifically for operational needs are used as such for reporting or analytics purposes, although these models are usually quite different, in terms of how the data is stored and how it is accessed. Yet, it is not uncommon to find a given database used both as the operational as well as the reporting database. As a direct consequence of violating SoC with respect to data modeling (both logically and physically), stability, scalability, extensibility, and performance are the main quality attributes of the system that will be impacted.

An alternate description of this problem is known as the "One Bunch of Everything" antipattern [12], qualifying it as a performance antipattern in database-driven applications, the author aptly pointing out that "treating different types of data and queries differently can significantly improve application performance and scalability."

2) The Solution

Following general data architecture guidelines, the solution is straightforward. In [13], Martin Fowler suggests the separation of operational and reporting databases and outlines the benefits of having domain logic access the operational database while also massaging (pre-computing) data in preparation for reporting needs. Extract-Transform-Load (ETL) pipelines/workflows can and should be created to move operational data into the reporting database; specifically, into custom-tailored models that cater to requirements around reporting and efficient data reads.

Existing tooling and frameworks can be employed to transform and move data efficiently, on premise or in the cloud (Azure Data Factory, Amazon AWS Glue, Matillion ETL, etc.), for data mining and analytics, for historical as well as real-time reporting needs.

C. Data Access and Leaky Abstractions

1) The Problem

It has been noted [14] that Object Relational Mapping (ORM) technologies, such as Entity Framework (EF) or Hibernate, are in fact a significant cause of data architecture bleed into the application logic, representing yet another example of the Leaky Abstractions antipattern.

Although intended to ease the access to the data tier and the data it hosts, such technologies expose underlying models and behavior to the application tier. In more acute cases – depending on its usage – it also introduces strong dependencies from the domain logic to the data shapes defined in tables, views, and table-valued functions.

Entity Framework, for example, while providing the ability to create custom mappings between these data models and the entity models, as designed, these object models are intended to be used as the main domain entities to build the actual domain logic around them. This forces a strong, intertwined yet inadequate dependency between two very different models, targeting different technologies, employed by very different programming paradigms (OO/functional such as C#.NET versus set-based such as SQL). This not only restricts the shape of the domain models, forcing constrained behavioral models to be implemented around them, but also causes data architecture changes to affect the domain and the application logic itself.

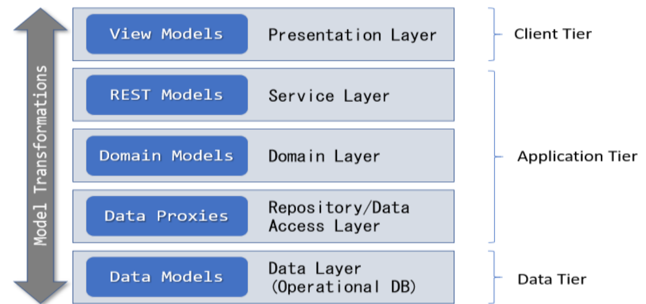

Figure 7. Layered architecture with layer-specific models and model transformations

Not surprising, Microsoft's EF Core framework in fact discourages against using a repository layer [15] (as prescribed by Evans's DDD [16]) on account that EF itself implements the repository pattern/unit of work enterprise pattern [17] – alas, leading towards a rigid and potentially brittle integration. The reason is that ORM technologies push design and development towards data access logic tangled with the domain logic by encouraging multi-purpose models (domain and data access or data proxies).

2) The Solution

Just as with the integration solution presented in Figure 3, the impact of changes to database models should be constrained to one or two components – those that make up the data access layer, and prohibited from affecting the other application layers, specifically the domain and service layers. Sharing a single model across all layers of the application places unnecessary limitations on the overall design and ultimately on the extensibility and stability of the system.

Although it is uncommon to replace the database technology altogether, sometimes it may be required to replace the *data access* technology due to performance and scalability concerns. Without a proper separation of data access from domain logic and models, such design changes targeting the lower layers of the system architecture are impractical without extensive refactoring of the application.

In a layered component-based architecture – as shown in Figure 7 above, it is easy and natural to allow each layer to define its own models (darker boxes) and provide adapters to translate from one model to another as data flows through the layers of the application by means of interfaces. Although this would seem wasteful at first sight, especially if some models hardly vary from one layer to the next, this approach offers two core benefits. It allows for independent evolution of the models, customizing them to serve very specific needs of the layer they belong to, and keeps the propagation of model changes confined to the corresponding adapter (translation) components.

In case of ORM technologies, the data access layer overlaps with the domain layer, while entity models (shown as data proxies in Figure 7) represent the actual domain models. Interestingly enough, even as ORM is recognized as a Leaky Abstraction, its use is nevertheless encouraged [18], most likely because in unsophisticated implementations, it may be able to deliver acceptable results.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

However, as Bilopavlovic points out [14], ORM tools can be successfully used "if there is proper separation of concerns, proper data access layer, and competent developers who know what they are doing and really, really understand how relational databases work." Sooner or later, the inherent deficiencies of such technologies, compounded by inadequate implementations due to the lack of understanding of how the underlying technology works, will surface, in most cases under system load and/or when new features are added.

V. Data Tier Management Concerns

Previous sections discussed antipatterns as relevant to the design of software solutions, specifically to the design of software systems integration. Bad practices and approaches can be encountered in several areas, beside design: in code implementations, in management of the code and software artifacts, in activities pertaining to DevOps, such as deployment and change management, etc.

This section will focus on inadequate practices around management of relational databases, with a detailed focus on Microsoft SQL Server relational databases, tooling and frameworks used for change management and incremental deployments, among other things.

A. *Improper Management of Database Artifacts*

1) *The Problem*

Source code, regardless of the language it is written in, is "a precious asset whose value must be protected", as Atlassian's Bitbucket web site states in their "What is version control" online tutorial [19]. All software-producing companies will employ one tool or another for version control. This allows software developers to collaborate, store (or restore/rollback) versions of the software components they build and perform code reviews, providing a single, stable "source of truth" of the software artifacts they create and release/deploy. As advocated in [20], "source files that make up the software system aren't on shared servers, hidden in folders on a laptop, or embedded in a non-versioned database." Yet, it is rather commonplace to find database implementations that are improperly managed, leading to frustration, bad deployments, making the data tier integration and overall solution delivery unreliable and difficult. There are many online articles and blogs describing such cases.

As encountered by the author, while being engaged as a solution architect and consultant on several projects at various clients, the actual data models and database artifacts were often created and delivered as *ad-hoc implementations* in some arbitrary database, hosted under some *arbitrary* Microsoft SQL Server database instance. Several teams needed these database artifacts: Development for implementation and integration, Quality Assurance for testing, Business team (domain experts and business analysts) for reporting and analytics, and DevOps for deployment. The most common process for deploying this database (fresh install or incremental) to some other environment was to generate and pass around SQL scripts when needed. In somewhat more fortunate situations, these scripts were maintained in some form of source control as SQL/text files but lacking the ability to validate them or trace the source back to the developer responsible for the actual implementation (in the original database).

So then, where does the "source of truth" for the database definition reside? How can multiple developers work on the database code without overwriting each other's changes and without being aware of the latest updates? How does the organization deliver incremental deployments to any number of target environments? When onboarding new team members, what database code should they be pointed to?

The problems derived from not having a stable, accurate, up-to-date, and complete definition of the database source code, one that is under version control and that can be validated before a deployment, are numerous, acute, and rather obvious. Just as one maintains all other application code under source control, entire solutions composed of many components, why should database implementations not follow the same standards and take advantage of the same acclaimed benefits of code well-managed?

Furthermore, when the database (source) code resides in some database, invalid object references (because someone dropped a column on a table or deleted a stored procedure) will surface only at runtime. Often, changes are made to the database post deployment, even in Production environments, changes that could potentially break the code, or which are at best confined to that environment alone, but without being retrofitted/updated back into the "source code database".

A particularly curious approach to database code management and deployment was encountered on a project that used the Fluent Migrations Framework for .NET [21], self-proclaimed as a "structured way to alter your database schema […] and an alternative to creating lots of sql scripts that have to be run manually by every developer involved." In a nutshell, the tool calls for creating a C#.NET class every time the database schema would change (one class per "migration"). These code files (admittedly, version-controlled) attributed with metadata to identify a specific database update, encapsulate two operations that describe the schema changes: one for a forward deployment ("Up") and one for rollback ("Down").

A very simple example, involving the source code of a rather trivial stored procedure, is shown in Figure 8.

With a large database, one that evolved considerably over time, with hundreds of artifacts, the number of C# migration files was astounding (thousands). Database changes were published to the target database as part of the application deployment process. Installing the database from scratch would incrementally apply every single "Up" migration specification found in these files, following the prescribed update. To maintain sanity, these source code files needed to be named such that the chronological order would be preserved when browsing through them in the development environment tool.

However, other more serious problems arise from using this framework, two of them being briefly discussed next.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
using FluentMigrator;

namespace DatabaseMigration.Migrations
{
  [Migration(98)]
  public class M0098_CreateStProcAddNodes : Migration
  {
     public override void Up()
    {
        Execute.Sql(@"CREATE PROCEDURE [cfg].[AddNodes]
          @nodes cfg.NodeType READONLY
          AS
          BEGIN
            SET NOCOUNT ON;
            INSERT INTO cfg.Node
              (Name, Value, ValueType, CreatedBy,
               CreatedDate, UpdatedBy, UpdatedDate)
            SELECT s.Name, s.Value, s.ValueType, s.CreatedBy,
               s.CreatedDate, s.UpdatedBy, s.UpdatedDate
            FROM @nodes as s;
          END");
    }

    public override void Down()
    {
      Execute.Sql("DROP PROCEDURE [cfg].[AddNodes]");
    }
  }
}
```

Figure 8. Sample C#.NET migration code for adding a new stored procedure and rolling back the change

a) SQL code as C#.NET strings??

Say a new stored procedure must be added; the code is developed and tested from SQL Server Management Studio (SSMS) in some local deployment of the database (assuming the objects the stored procedure is referencing do not change in the interim). Next, a migration file is created, with the "Up" method containing the full (CREATE) stored procedure script, *as a C# string* passed as input argument to the "Execute.Sql" method call. A sample migration code snippet describing this scenario is shown in Figure 8.

The major and obvious problem here is the inability to validate SQL syntax and semantics and SQL object references when represented as indiscriminate plain strings, subject to typing errors.

b) No database source code??

Unless deployed on some SQL Server instance, it is impossible to even begin to understand the structure of the database, even the structure of individual objects. The data models and data logic are scattered, fragmented (across many C# files), impossible to validate (syntactically or otherwise) from where the database "source code" is stored.

Moreover, a given database object, say a table for example, can change any number of times, each change being captured in a different source file, with no unified, single view of what that table looks like, what the shape of the data is, with all its columns and corresponding types, with its keys and indexes, constraints and triggers, if any. This problem extends to all database objects, not just tables.

The data models (the source code artifacts) are practically non-existent, disjointed, difficult to comprehend, and cannot be validated until they are deployed. The result is a total and indefensible representational incoherence afflicting the most important component of a data-dependent enterprise system.

2) The Solution

There are various software tools available to address this problem. Both Microsoft and Redgate, for example, provide excellent tooling for developing relational databases, managing database artifacts under source control, facilitating change management and incremental deployment, generating manual update scripts (when automated deployment is constrained), and more.

Microsoft's SQL Server Data Tools (SSDT) [22] is a development tool, available since 2013, using the Data-Tier Application Framework (DacFx). It facilitates the design and implementation of SQL Server and Azure SQL databases, as well as database source control and incremental deployment, all integrated under the Microsoft Visual Studio development environment.

A version-controlled database project contains all distinct database objects as individual files, and it *must* compile – targeting a specific SQL Server (or Azure) database version – before it can be deployed anywhere. Developers can check out individual objects (files) to change as needed or can add new objects using the provided templates. Just as one can see the entire schema of a database in SSMS, similarly they can see and browse these objects in Visual Studio, as shown later in the development environment snapshot in Figure 9. Here, the main database project (`Config.Database`) is – like all projects in the bounding solution – subjected to building or compilation. As a result, two artifacts are being created: a managed assembly file (.dll) and a data tier application package (.dacpac) file. Both are required for actual database deployment, but it is the .dacpac that holds the actual and full database definition. It is used by the Microsoft tooling (SqlPackage.exe) employed for incremental deployments (schema updates) against targeted environments.

It is highly questionable to store Java or C# code in SQL scripts, with artifacts/classes shredded and reduced to SQL NVARCHARs, scattered in an arbitrary number of stored procedures (equal to the number of updates effected upon that class), and passed around to call other stored procedures (via EXEC statements). The reverse scenarios should be equally unacceptable. Treating the database as a proper software implementation artifact is imperative.

B. Database Development and Deployment Concerns

1) The Problem

Tools like SSDT are also capable of identifying the changes (delta) between the source and the destination database in order to create the appropriate deployment scripts, and ultimately allowing rapid and valid delivery of database changes to any environment. Quite frequently, multiple teams are involved in database development: backend developers of applications relying on persisted data as well as data migrations (ETL) and reporting developers. Bringing all teams together to follow unified and consistent database development and deployment processes can be challenging.

Furthermore, how can specialized implementations be properly designed, source-controlled, and deployed seamlessly, while keeping the two implementations (core and custom) separate but dependent solution components?

Considering a database (and hence its associated project) as part of a larger software system, as an essential component of that system, requires indeed some additional effort in designing and managing all system's artifacts under a unified solution framework. If libraries and executables are easy to group around layers and features, whether they cater to domain versus cross-cutting concerns of that system, database componentization strategies may not be straightforward. However, recognizing that even databases and their underlying objects (i.e., code) can be broken down into logical parts, will facilitate management of these artifacts and better extensibility.

To better understand this, consider a database that consists of core objects (tables, procedures, functions, etc.), perhaps part of a product line that evolves over time. Some customers may ask for certain customizations, for example, that require additional database objects to be created, specific to their business rules and models (as would be the case of custom reports that rely on custom views).

One option is to design and implement these new views directly in the targeted environment, without including them into the source-controlled database component. The database/reporting developers would separately maintain these objects, but when later the underlying tables change, the views referencing them may break, and hence the validity of the reporting component is jeopardized.

2) The Solution

Alternatively, extensions of the core database component can be created – as separate database projects, holding only these additional custom objects, with a same-database dependency setting to the main database (project). Teams can independently work on core versus custom components, both being validated (compiled) and source controlled.

Figure 9 shows such a solution, with two database projects (components), one extending the other, with the extension component, `ConfigExt.Database`, having a dependency to the main component via database reference. Then, for actual deployment, the extension database package would be used – as it contains both custom objects as well as the core database objects from the referenced component, resulting in a *full* database installation or update.

The tooling and processes described here, as already mentioned, target the Microsoft technology stack. However, similar options exist for other platforms as well, more or less effective in various areas or others, to assist with development and management of enterprise databases.

Figure 9 shows a snapshot of a solution developed under the Microsoft Visual Studio environment, with two of the 19 projects being a couple of rather trivial database projects, `Config.Database`, and its extension, `ConfigExt.Database`. Either project encapsulates an entire (yet simple) database with all its objects grouped under schemas and object type folders. The top right panel shows the same stored procedure

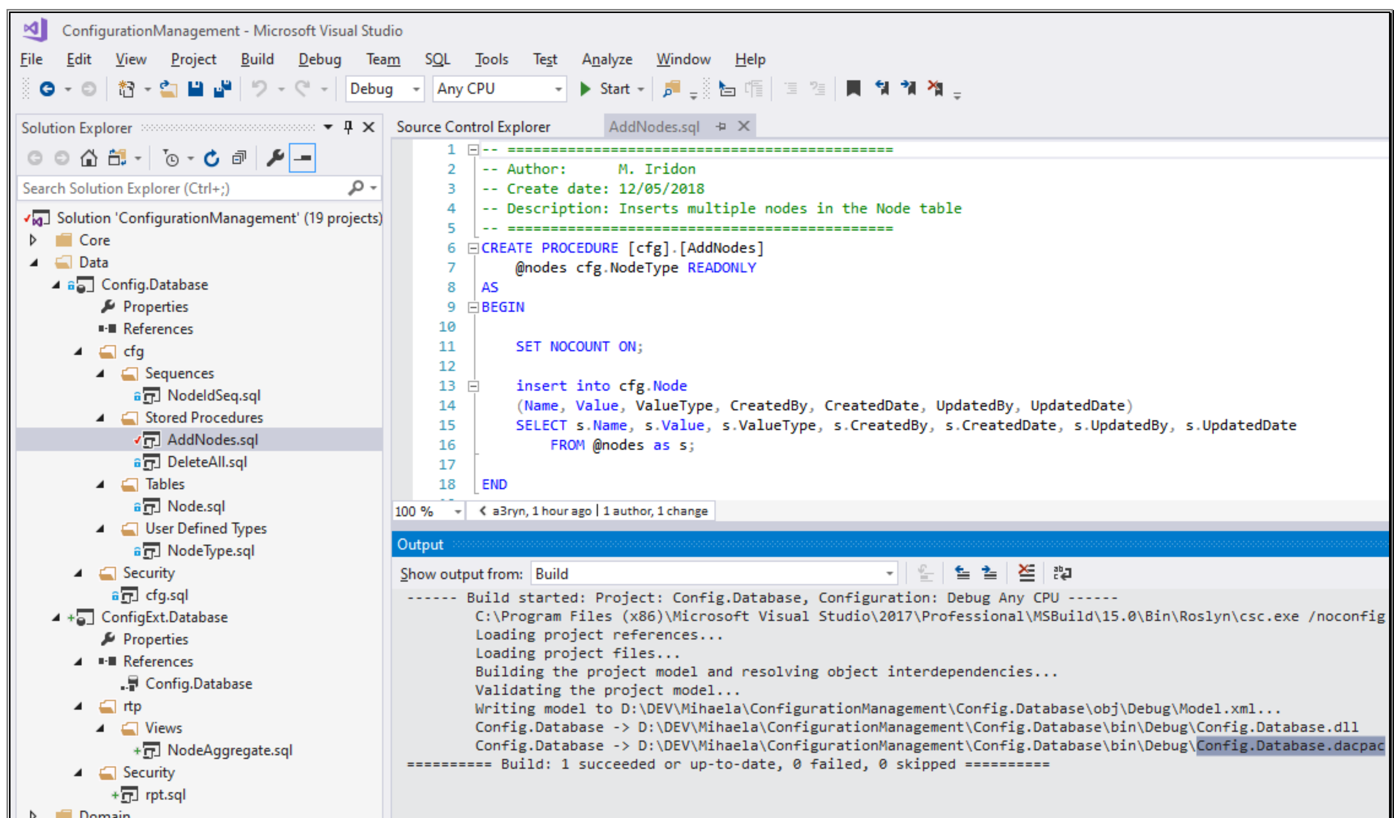

Figure 9. Database source code managed in Microsoft Visual Studio via SQL Server Data Tools.
Code files are checked in or out from a source control repository (shown on the left) as database development in is progress.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

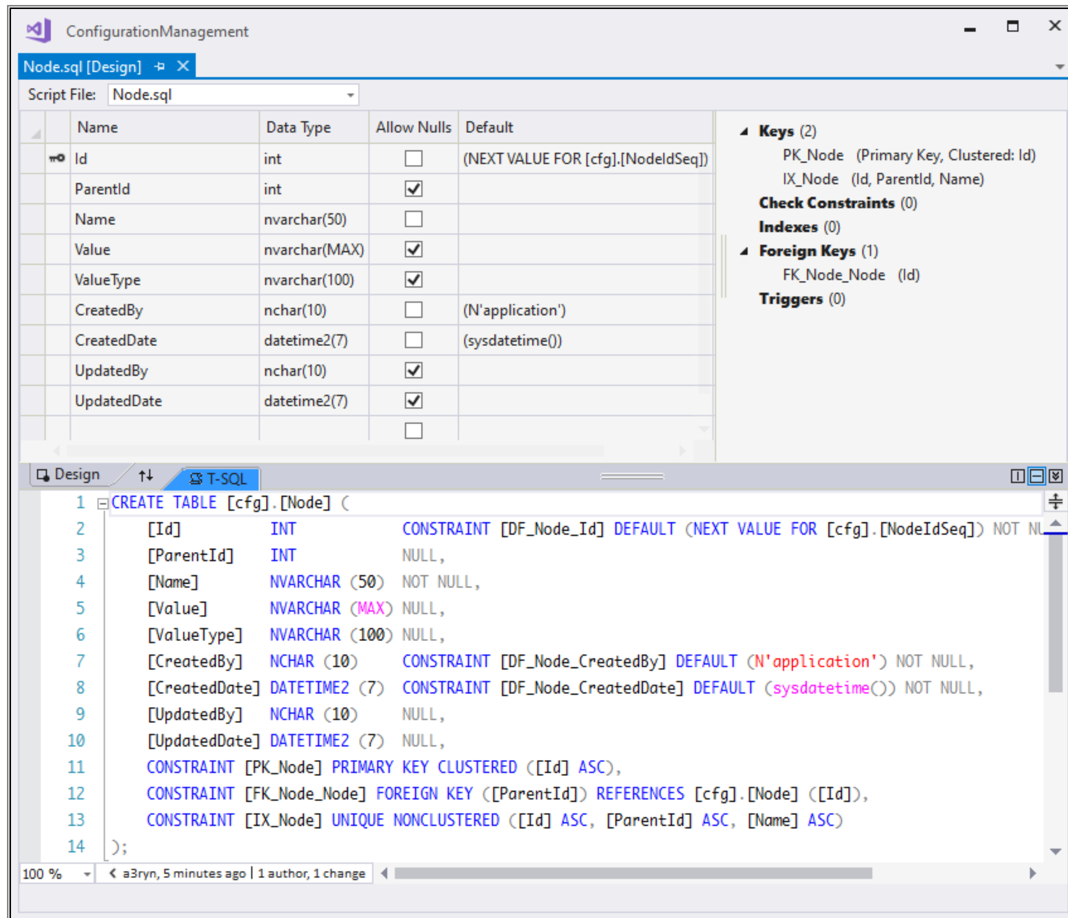

Figure 10. Database table designer (top) and script (bottom) snapshot in Microsoft Visual Studio, using SQL Server Data Tools

from earlier in Figure 8– whose source code was captured there as a C#.NET string. In contrast, here it is managed as a proper element of the database, that can be compiled (validated) and independently tracked for code changes.

The project/database compiles successfully, as shown in the bottom part of the screenshot in Figure 9. The build output artifact, i.e., the data tier application package `Config.Database.dacpac,` is highlighted.

Similarly, table objects (including indexes, constraints, etc.) can be managed in a fashion that resembles the look and feel of the table designer utility in SQL Server Management Studio. This visual design feature is captured by the top section in Figure 10. Otherwise, the scripting option (bottom) is always available, for all object types.

For all database objects, only the CREATE statement is used in all SQL source code. The tooling itself determines, at deployment/publishing time, whether CREATE or ALTER Data Description Language (DDL) statements will be required based on the delta between the concrete target database and the database source code. This greatly simplifies deployment of SQL databases against any environment, including fresh installations as well as incremental updates.

Finally, as far as employing SQL Server Data Tools and treating databases as proper software artifacts, we can enumerate below some of the key benefits that should encourage software companies to adopt SSDT, should they design and develop solutions around Microsoft's SQL Server relational databases.

To briefly summarize, here are these benefits, which should be considered perhaps also as a guiding set of objectives for any database development activity:

- ✓ Providing a unified perspective of a database,
- ✓ Validating correctness of the database definition,
- ✓ Validating completeness of a database definition,
- ✓ Providing support for version-control of the database artifacts (at the database object level),
- ✓ Allowing to perform schema comparison,
- ✓ Facilitating incremental deployments (change management), directly against a target database or via SQL scripts,
- ✓ Enabling the logical and physical componentization of databases, to facilitate the customization, extensibility, and manageability of the underlying artifacts.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

VI. CONCLUSION

This paper aimed to raise awareness about certain design challenges that, when not addressed early and properly, will lead to deficient architectures and rigid solutions concerning various aspects of system integration, as often encountered in practice.

When the design of software systems follows some basic guidelines and principles (SOLID), the resulting architecture will allow the system to be easily built, modified, and extended. In case of system integrations and customizations, violating these principles and particularly the multi-faceted Separation of Concerns design rule, leads to unmanageable and highly complex systems that do not scale well, cannot be extended or modified easily, with tight dependencies on external components and overall brittle integration solutions.

Many design antipatterns have been catalogued and well documented; yet deficient architectures are encountered quite frequently, leading to high technical debt and unhappy stakeholders. This paper discussed "Leaky Abstractions", "Mixing Concerns", and "Vendor Lock-in" antipatterns – from the perspective of concrete industry examples, as encountered and worked on by the author.

Concrete approaches that address these problems to help refactor and realign the design according to best practices and principles were elaborated, explaining how they lead to scalable, extensible, testable, efficient, and robust integration solutions.

Relational database design and management concerns were also presented, with focus on data model design, data access practices, and management of database artifacts. The consequences of improper tooling and frameworks were briefly covered, and a technology-specific solution targeting Microsoft SQL Server databases was discussed.

ACKNOWLEDGEMENT

I would like to thank my husband and long-time mentor, Chris Moore, for his indefatigable guidance and for sharing the extensive technical knowledge and experience he possesses and masters so adeptly.

REFERENCES

[1] M. Iridon, "Industry Case Study: Design Antipatterns in Actual Implementations. Understanding and Correcting Common Integration Design Oversights," FASSI 2019: The Fifth International Conference on Fundamentals and Advances in Software Systems Integration, ISBN: : 978-1-61208-750-4, pp. 36-42, Nice, France, October, 2019.

[2] R. Martin, "Clean Architecture," Prentice Hall, 2018, ISBN-13: 978-0-13-449416-6.

[3] J. Spolsky, "The Law of Leaky Abstractions," [Online] Available from https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/ [retrieved: May, 2020].

[4] SourceMaking, Software Architecture AntiPatterns, [Online]. Available from https://sourcemaking.com/antipatterns [retrieved: May 2020].

[5] M. Iridon, "Enterprise Integration Modeling," International Journal of Advances in Software, vol 9 no 1 & 2, 2016, pp. 116-127.

[6] Genesys, "Platform SDK," [Online]. Available from https://docs.genesys.com/Documentation/PSDK [retrieved: May, 2020].

[7] Genesys, "Web Services and Applications," [Online]. Available from https://docs.genesys.com/Documentation/HTCC [retrieved: May, 2020].

[8] G. M. Hall, "Adaptive Code via C#: Agile coding with design patterns and SOLID principles (Developer Reference)," Microsoft Press, 1st Edition, 2014, ISBN-13: 978- 0735683204.

[9] M. Seemann, "Dependency Injection in .NET," Manning Publications, 1st Edition., 2011, ISBN-13: 978-1935182504.

[10] Microsoft, "Extract, Transform, and Load (ETL), " [Online]. Available from https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/etl [retrieved: May, 2020].

[11] G. Simsion and G. Witt, "Data Modeling Essentials," Morgan Kaufmann; 3rd edition, 2004, ISBN-13: 978-0126445510.

[12] A. Reitbauer, "Performance Anti-Patterns in Database-Driven Applications," [Online] Available from https://www.infoq.com/articles/Anti-Patterns-Alois-Reitbauer/ [retrieved: May, 2020].

[13] M. Fowler, "Reporting Database, " [Online]. Available from https://martinfowler.com/bliki/ReportingDatabase.html [retrieved: May, 2020].

[14] V. Bilopavlovic, "Can we talk about ORM Crisis?". [Online] Available from https://www.linkedin.com/pulse/can-we-talk-orm-crisis-vedran-bilopavlovi%C4%87 [retrieved: May, 2020].

[15] Jon P. Smith, "Entity Framework Core in Action," manning Publications, 2018, ISBN-13: 978-1617294563.

[16] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software," 1st Edition, Prentice Hall, 2003, ISBN-13: 978-0321125217.

[17] M. Fowler, "Patterns of Enterprise Application Architecture," Addison-Wesley Professional, 2002.

[18] M. Fowler, "OrmHate," [Online]. Available from https://martinfowler.com/bliki/OrmHate.html [retrieved: May, 2020].

[19] Atlassian, "What is version control, " [Online]. Available from https://www.atlassian.com/git/tutorials/what-is-version-control [retrieved: May, 2020].

[20] P. Duvall, "Version everything," [Online]. Available from https://www.ibm.com/developerworks/library/a-devops6/ [retrieved: May, 2020].

[21] "Fluent Migrations Framework for .NET," [Online]. Available from https://fluentmigrator.github.io/ [retrieved: May, 2020].

[22] Microsoft, "SQL Server Data Tools," [Online]. Available from https://docs.microsoft.com/en-us/sql/ssdt/sql-server-data-tools [retrieved: May, 2020].

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org