# Understanding Legacy Architecture Patterns

Ricardo Pérez-Castillo[1], Benedikt Mas[2] and Markus Pizka[2]
[1]*Itestra GmbH, Capitán Haya 1, 15th floor, room 16, 28020 Madrid, Spain*
[2]*Itestra GmbH, Destouchesstr. 68, 80796 München, Germany*

Keywords: Legacy Information Systems, Architecture Patterns, Software Maintenance, Modernization.

Abstract: While often being the not very well liked stepchild of IT departments, legacy information systems are valuable assets for companies. After many years of development and maintenance, these systems often contain valuable business logic and implement business processes that are nowadays unknown even to the owner of the system. However, maintenance and further development is often costly and requires an increased effort compared to modern applications. Hence, developing sound strategies for gradually modernizing these applications and lowering the associated costs is of paramount importance. For carrying out such strategies, it is useful to understand why and how certain aspects of these systems are implemented. At itestra, we have collected architectural patterns in legacy information systems and use these to understand legacy information systems better and avoid mistakes in the analysis of behavior of such legacy systems. We present these patterns here in order to facilitate the decision-making process in modernization projects and increase their success probability.

## 1 INTRODUCTION

The questioned challenge lies in a contradiction. Major corporations usually possess a history and large array of so called legacy information systems that consume a great part of their IT budget in operations and maintenance. On the one hand, these systems have been extended over many years to support business processes optimally or business processes have been adapted over time to work efficiently with an existing system and the workforce is used to the established process. A lot of business knowledge is implemented in these mission critical systems, up to the point that certain aspects of processes are not documented elsewhere but in the source code (Paradauskas and Laurikaitis, 2006; Sommerville, 2006).

On the other hand, IT departments and software vendors have cultivated a belief that *"anything new is beautiful and that everything old is ugly"* and we have become *"victims of a volatile IT industry"* (Sneed, 2008). Old legacy systems are frowned upon and it is very hard to find engineers that are interested in working with these applications, creating a maintenance risk. The maintenance cost explodes quickly (Bennett, 1996; Sneed, 2005; The Standish Group, 2010) and software erosion is visible all over the systems (Paradauskas and Laurikaitis, 2006):

- Systems are implemented with obsolete technology which is expensive to maintain.
- The lack of documentation leads to a lack of understanding, making the maintenance of these systems a slow and expensive process.
- A great effort must be made to integrate a legacy system with other systems, since the interfaces and boundaries of the legacy system are not well defined.

Most importantly however, in an increasingly globalized world with 24/7 connectivity, the established business processes are not adequate anymore and need to bring into a world where customers prefer mobile self-service portals over visiting service centers and talking to service agents in person.

Engineers that are given the task to adapt the IT support to 21st century business processes are frequently shocked when they analyze the existing landscape, qualifying the existing applications as "garbage" and stating that a replacement from scratch, even though prohibitively expensive and inherently risky, is inevitable.

We argue that in many cases this attitude is a result of the *not invented here* syndrome or cognitive


Pérez-Castillo R., Mas B. and Pizka M..
Understanding Legacy Architecture Patterns.
DOI: 10.5220/0005467302820288
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 282-288
ISBN: 978-989-758-100-7

dissonance (Harmon-Jones and Harmon-Jones, 2007) and that in many cases an attempt to try and modernize an existing application in small, controllable steps would be both economically more attractive and less risky than a from scratch replacement.

For more than 10 years now, itestra, an IT service company focused on strategy consulting, aims at reengineering, replacing or migrating legacy information systems. Its project portfolio provides itestra with the possibility of analyzing a vast number of legacy information systems, treating a wide variety of legacy technologies and observing different development paradigms and architectures. This hands-on experience has allowed itestra to define a set of architectural patterns frequently visible in legacy information systems.

From our experience, we observe that understanding a set of typical architectural patterns makes the analysis of existing applications easier, avoids the aforementioned cognitive dissonance and ultimately allows the development of an economically efficient and safer modernization strategy for many legacy applications. This paper explains patterns such as the vertical model, one access program per table, one screen per table, step by step, record based, object orientation in COBOL, data collector, data synchronize, and building blocks. We argue that if managers, architects and developers know and understand these architectural patterns (and their causes or origins as well), they are in a better position for understanding legacy information systems and making a sound choice of a modernization strategy.

The paper is organized as follows. Section 2 presents important architectural patterns recognized in legacy information systems. Section 3 explores the root causes and origin of those patterns. Section 4 explains consequences of ignoring those patterns, and finally, Section 5 presents conclusions and main implications.

# 2 ARCHITECTURAL PATTERNS IN LEGACY SYSTEMS

A pre-requisite to optimal and tailored modernization strategies for legacy systems is the understanding of internal details and behaviour of the system under consideration. This understanding is often hampered by a lack of knowledge of architecture, technologies used and a lack of understanding of the environment when the design decisions regarding the system were made.

One of the factors that determine the legacy nature of a system is the paradigms that were *en vogue* at the time these systems were built. Since the 1980ies, the requirements and technologies have changed significantly (see Table 1). Obviously, these differences led to a design vastly different from an architecture that would be chosen in a from-scratch development today.

Table 1: Differences between legacy systems and modern applications.

| | **Legacy** | **Modern** |
|---|---|---|
| **Nature** | Static | Dynamic |
| **Execution of system-based tasks** | Centralized batch processes | Online availability. 24/7 |
| **Tasks organization** | Procedures | Processes |
| **Delivery** | Data | Services |
| **Development and Maintenance focus** | Technical/procedural Steps | Business entities |
| **Developers and maintainers question** | How to automate those procedural steps? | How to model business entities and interactions? |
| **Development challenges** | Limited computing power and centralized IT | Distributed, connected processes and applications |

Nine architectural patterns, explained in next subsections, are frequently encountered in legacy systems: (i) vertical model, (ii) one access program per table, (iii) one screen per table, (iv) step by step, (v) record based, (vi) object orientation in COBOL, (vii) data collector, (viii) data synchronize, and (ix) building blocks. Understanding and recognizing these patterns makes an analysis of these applications easier. Each pattern is described in a few sentences and then its consequences are explained.

## 2.1 Vertical Model

For mainframe applications, transactions offer the possibility to implement online functionality, e.g. user interfaces or interfaces to other applications. In a vertical model, these transactions are implemented

in a set of dedicated programs per transaction, completely avoiding sharing of code between transactions. The only persistent state that is available to these transactions is held in the database. Within the transactions, the programs show a high cohesion and no defined structure, e.g. layers. A legacy information system implementing this pattern can be regarded as a set of silos, with each silo being independent from all other silos. The coupling between the silos is usually highly complex, but very hard to analyze due to the lack of a way to exchange data other than the database.

Internally, these applications repeat functionality many times and show a high rate of code duplication, leading to an enormous code base and ultimately excessive maintenance cost. Vertical model is one of the most recurrent architectural patterns we have found in COBOL-based systems.

## 2.2 One Access Program per Table

When today's legacy systems were built, many engineers had experience in working with files and file based access patterns. Relational databases were new and the nowadays commonplace language *SQL* for reading and writing data was exotic and could only be managed by a few specialists. Additionally, it was still unclear if relational databases would be the accepted solution for many years in the future.

In order to overcome this situation, architects chose to encapsulate SQL and the database in "access programs" per database table, similar to what one would do in a file based system. The access program is used to perform all data access operations concerning a *single* database table. Usually, the program implements only trivial access methods to this table (e.g. READ_ALL or READ_BY_ID), leading to inefficient access patterns in the business logic that has to implement complex selection and filtering routines. SQL operators such as JOIN are hardly used in this pattern, depriving one from powerful instruments offered by modern databases.

The negative consequence of such an architecture are slow applications that are cumbersome to maintain and, due to their high CPU usage, very costly to operate.

## 2.3 One Screen per Table

Similarly to the previous architectural pattern, one screen per table refers to design solutions which provides a user interface (a dialog or form) implementing user interaction so that she or he can execute CRUD (create/read/update/delete) operations in a certain table or data file. In these architectures, the user is presented with a choice of which data to modify on a certain table or file. The execution of a complex use case is then restricted to the execution of individual operations on different database tables by the user.

One of the most extended today patterns to decouple user interfaces and business logic is MVC (Model-View-Controller) (Bodhuin, Guardabascio et al., 2002; Ping, Kontogiannis et al., 2003). Following MVC pattern, each user interface is associated most of times with a use case rather than a database table. The migration of old user interfaces into new ones by following modern patterns like MVC is one important challenge in modernization projects.

Again, the side effect of architectures following this pattern is the necessary extra effort so that to provide new architectures with low-coupling layers for persistency, business logic and user interfaces.

## 2.4 Step by Step

Step by step pattern consists of architectural solutions in which batch procedures are implemented to complete a sequence of tasks in a row. Complex business processes are broken down into many small and individual steps. Each implemented step reads the data to be processed from persistent storage, applies a (often trivial) transformation and saves it back into storage. The ultimate business process that is implemented by a set of steps is unclear in the individual step, making the implementation resilient to change.

The weak of this architecture is that lead to frequent read/write operations of temporary data, which are slow and cumbersome, and increase computing cost as well.

The challenge for this architectural pattern is how to model underlying business processes and entities related to the sequence of steps. Additionally, the implementation of these processes in modern technologies instead of individual batch programs can be tricky.

## 2.5 Record based

Rather than implementing operations that can operate on a set of records in a cross-cutting way, an implementation is chosen where an operation is executed on one isolated record at a time. An example would be to read a record from a database, update a value of that record and write it back rather

than executing an update operation directly in the database.

The main drawback of record-based architecture is that processing in these systems is usually sluggish, and the only way to improve performance is increasing mainframe computing costs dramatically through hardware solutions.

The challenge in modernization projects involving this architecture pattern is how to merge atomic operations focused on isolated records into a more complex, cross-cutting ones.

## 2.6 Object Orientation in COBOL

Another pattern found in legacy information system lies in the fact that architects attempted to model everything by following Object-Oriented (OO) paradigm, even when the technology or programming language does not support it at all. This is the case of various COBOL-based systems in eighties and nineties, in which architects decided to pseudo-implement objects using COBOL.

This design decision very often lead to baroque OO designs, which has at least two major drawbacks. Firstly, objects (represented by single compilation units in most cases) do not have the concept of 'this' to refer to the own class elements. As a result, each time an object operation is called, the respective delegating class has to be linearly searched among all the available objects. The second inconvenience lies in the fact that the OO nature in this way is stateless, i.e., there is no concept of state for objects being used in a running program.

Although the presence of this architecture pattern may facilitate direct translations to OO programming languages, many side defects in such architectures as a result of forcing OO nature have to be considered.

Other threat of this pattern is that the migration of such architectures toward today OO architectures is error-prone due to the pseudo-implementation of OO nature can lead engineers to take erroneous assumptions during migration like, for example, every object in the legacy system should be transformed into an object into the new one. Other open issues are how to model inheritance and how to use polymorphism in the target system by preserving, at the same time, business logic of legacy systems following this pattern.

## 2.7 Data Collector

True to its name, a data collector is usually an isolated batch job in a legacy information system that collects data and stores it in a shadow database. This pattern is frequently found in systems that require monthly batch processes.

Data collector pattern is also challenging in modernization projects since those monthly batch processes may have to be replaced by new collector/backup processes in the new system. The main effort dealing with data collectors focuses on preserving all the embedded business logic of the data collector and integrate it in the target system.

## 2.8 Data Synchronize

In legacy information system we frequently encounter situations where over the long history of the system various data sources have been implemented and hold redundant data. For example, in an attempt to integrate a legacy information system that operates on files into a service-based IT landscape, a set of web services has been implemented. These web services do not operate on the same files, but on a database. Rather than migrating the entire system to the database, data is kept redundantly between the files and the database. To keep the data in sync, a set of batch jobs is implemented and executed recurrently.

The main challenge in modernization projects involving this pattern is how to cope with all the different data sources. Sometimes, some data sources cannot be removed and batch jobs for synchronizing data may have to be migrated to the target system.

In other cases, the main effort could be focus on data migration between old and new data sources. Data migration is an error-prone task since lead in most cases to data quality problems (Batini, Cappiello et al., 2009) in the new system.

## 2.9 Building Blocks

This pattern focuses on the identification and design of building blocks, which are software components that can be independently developed and deployed. Architectural building blocks (e.g., component, connector, or module) is native to a style. Identification of building blocks very often goes along the object dimension. However, there is no restriction and a building block other dimensions as well, e.g., aspect or concurrency dimensions (Müller, 2003).

The main idea of the building blocks pattern is to take descriptions of application domain functionality, commercial product features, system qualities and technology choices as input and

produces a number of architectural models and construction elements.

This pattern is problematic in modernization projects because most time engineers have to transform an architecture based on building blocks into new architectures that lack those blocks. As a consequence, engineers and managers of modernization projects involving this pattern have first to recognize building blocks, and then decide if these blocks are discarded, migrated or modernized.

# 3 ROOT CAUSES

Architectural design rationale describes the decisions made, alternatives considered, and reasons for and against each alternative considered when a software architecture is defined (Wang and Burge, 2010; Zimmermann, 2012). Although architectural design rationale is outside of the scope of this paper, it has to be taken into account to understand the harmful consequences in modernization projects when the mentioned patterns are overlooked.

Many engineers literally laugh when they face some of the mentioned architectural patterns. Today, in a modern context under well-known software engineering principles, the application of most of these architectural patterns does not make sense and their use is considered at least problematic.

However, these patterns were not harmful nor risky when these were applied, or at least, benefits of their application were higher than drawbacks. In many cases the solution that was implemented in a legacy system was the best solution available at the time.

Today's software engineers should take into account that architects had good reasons for applying these patterns. Most times, these reasons are related to the desire of architects for connecting business processes and hardware to fulfill with service provisioning constraints. For example, in the 1980ies, enterprise applications were largely batch-driven and centralized due to the limited amount of hardware available, restricted computing power and lack of network connections between different offices.

As a result, we can conclude that the origin of the application of most of these patterns lied in the necessity of fulfill different architecture constraints as the following ones:

- Software architecture and specifications that include language use, library use, coding standards, memory management, and so forth.
- Hardware architecture that includes client and server configurations.
- Communications architecture that includes networking protocols and devices.
- Persistence architecture that includes databases and file-handling mechanisms.
- Application security architecture that includes thread models and trusted system base.
- Systems management architecture.

# 4 CONSEQUENCES

As a summary of the architectural patterns presented in Section 2, Table 2 provides a matrix with the relationship between the nine architectural patterns and common challenges in modernization projects. Although engineers and managers should pay attention to all the challenges, Table 2 provides the most important concerns for each architectural pattern regarding our hands-on experience.

Table 2: Architectural patterns and common challenges.

| | | Common Challenges | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Business Process and Entity Modelling | Implementation in modern technologies | Architecture migration | Data Migration | Data Wrapping | Replacing/Improving Source Code | Performance |
| Architectural Patterns | Vertical model | | | • | • | | • | |
| | One access program per table | | | | • | • | | • |
| | One screen per table | • | • | | | | | |
| | Step by step | • | • | • | | | • | • |
| | Record based | | | | • | • | • | |
| | Object orientation in COBOL | • | | • | | | • | • |
| | Data collector | • | | | • | • | | • |
| | Data synchronize | | | | • | • | | • |
| | Building blocks | • | • | • | | | • | |

In our opinion, overconfidence of managers, architects, and/or developers who ignore these patterns can be related to one or more of the

following negative concerns in modernization projects:

- Lack of architecture planning and specification; insufficient definition of architecture for software, hardware, communications, persistence, security, and systems management.
- Hidden risks caused by scale, domain knowledge, technology, and complexity, all of which emerge as the project progresses.
- Impending project failure or unsuccessful system due to inadequate performance, excess complexity, misunderstood requirements, usability, and other system characteristics.
- Absence of technical backup and contingency plans.

At the contrary, we state that if managers, architects and developers consider the mentioned architectural patterns when dealing with legacy information systems, they may be in an advantageous position to successfully conduct modernization projects. It becomes possible to see trees in the forest, i.e., engineers who knows these patterns have guidelines to focus on certain concerns and dimensions of legacy information systems. Furthermore, the mentioned architectural patterns engineers are helpful for extracting embedded knowledge from legacy information systems in a more effective and efficient way.

# 5 CONCLUSIONS

Although legacy information systems and approaches for modernizing them have been widely treated in the literature, architectural patterns applied in legacy information systems have been studied superficially.

Itestra has been involved over the last 10 years in modernization projects aimed at reengineering, replacing or migrating legacy information systems. This hands-on experience has allowed itestra to define a set of architectural patterns recurrently detected in legacy information systems, which may be helpful for achieve success in modernization projects.

The main hypothesis treated in this paper is that engineers who know and understand these architectural patterns and their root causes can improve decision-making on modernization projects. In our opinion, the main conclusion of this study is that managers, architects and developers who understood and accept these architectural patterns can be better positioned to conduct effective modernization of legacy systems, and therefore, ensure project success.

As a future work, we have planned to carry out an in-depth study about accurate architectural design rationale of the analyzed patterns. Additionally, an interesting research line to be developed in the future is the study of anti-patterns generated in new information systems after modernizing legacy systems. In combination with this, the relationship between the architectural patterns presented in this paper and possible anti-patterns in target systems is an open issue as well.

# REFERENCES


Batini, C., C. Cappiello, C. Francalanci and A. Maurino (2009). "Methodologies for data quality assessment and improvement." *ACM Comput. Surv. 41(3): 1-52.*

Bennett, K. (1996). "Software evolution: past, present and future." In*formation and software technology 38(11)*: 673-680.

Bodhuin, T., E. Guardabascio and M. Tortorella (2002). Migrating COBOL Systems to the WEB by Using the MVC Design Pattern. Proceedings of the Ninth Working *Conference on Reverse Engineering (WCRE'02), IEEE Computer Society*: 329.

Harmon-Jones, E. and C. Harmon-Jones (2007). "Cognitive dissonance theory after 50 years of development." *Zeitschrift für Sozialpsychologie 38(1):* 7-16.

Müller, J. K. (2003). The Building Block Method. Component-Based Architectural Design for Large Software-Intensive Product Families. *Philips Research Laboratories. Eindhoven.*

Paradauskas, B. and A. Laurikaitis (2006). "Business Knowledge Extraction from Legacy Information Systems." *Journal of Information Technology and Control 35(3):* 214-221.

Ping, Y., K. Kontogiannis and T. C. Lau (2003). Transforming Legacy Web Applications to the MVC Architecture. Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice, *IEEE Computer Society*: 133-142.

Sneed, H. M. (2005). Estimating the Costs of a Reengineering Project, *IEEE Computer Society.*

Sneed, H. M. (2008). Migrating to Web Services. Emerging Methods, Technologies and Process Management in Software Engineering, *Wiley-IEEE Computer Society: 151-176.*

Sommerville, I. (2006). Software Engineering, Addison Wesley.

The Standish Group (2010). Modernization. Clearing a pathway to sucess, *The Standish Group International, Inc.*

Wang, W. and J. E. Burge (2010). Using rationale to support pattern-based architectural design.

Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge. *Cape Town, South Africa, ACM: 1-8.*

Zimmermann, O. (2012). Architectural decision identification in architectural patterns. *Proceedings of the WICSA/ECSA 2012 Companion Volume. Helsinki, Finland, ACM:* 96-103.