KTLO & BROWNFIELD: OVERCOMING CHALLENGES WHEN MODERNIZING PROCESS AUTOMATION AND BUSINESS INTELLIGENCE

Alois Paulin[1]

DOI: 10.24989/ocg.v341.17

Abstract
An estimated 50-80% of an organisation's IT budget is said to be spent on "keeping the lights on" (KTLO), with the remaining resources being often constrained to brownfield development in which legacy software systems define the boundaries to modernisation and innovation. Prominent business systems such as e.g., SAP ERP or SAP HANA fail to address all business needs, hence in-house software development remains unavoidable. This paper gives an overview of the development and successful implementation of a scalable in-house data aggregation and data provisioning system built to enable rapid responses to emerging process digitalisation and business intelligence needs. The architectural implications and considerations are discussed.

1. Introduction

With sophisticated information systems developed and deployed over the course of several decades, enterprises find themselves depending on outdated IT infrastructure that still fulfils key business needs. A prominent such example are the U.S. Internal Revenue Service's and U.S. Social Security Administration's COBOL-based stack of software developed in the early decades of e-government (i.e. the 1960s – 1980s) when the COBOL programming language was state-of-the-art choice for business applications (cf. [9], pp. 1-2).

Apart from such large-scale systems, myriads of smaller legacy systems are serving as vital parts of the enterprises' IT landscape that address everyday requirements such as information retrieval, data translation between domains, various process automation requirements, or business intelligence needs such as insights into the performance of business units or staff. Often these smaller systems have been developed internally as some kind of "shadow IT" projects [1, 11], i.e. as systems that provide needed functionality for business but have been developed and deployed without the knowledge and without having been officially sanctioned by the enterprises' central IT departments. Such "shadow IT" projects can assume any level of complexity and professionality ranging from simple macros built by skilled employees to facilitate reporting, up to professional middleware systems designed to fill the gap between what the enterprises' central IT departments provide and what the individual business units need.

The reliance on system-relevant "shadow IT" solutions bears the risk that the enterprise develops a strong dependency on the personal knowledge of the authors of such systems, who are the ones holding advanced know-how on how to modify their systems, how to find and remove bugs and glitches, add features, adapt the systems to changed interfaces, etc. The graveness of these risks becomes apparent when authors of such systems retire, aim for career changes, or leave the

[1] University of Public Administration and Finance, Ludwigsburg, Germany, alois@apaulin.com

enterprise. Once the original author of such system leaves, the successor is often better off developing new shadow IT solutions from scratch using modern development approaches, as the predecessor's code will likely be outdated, buggy, undocumented, or simply too complex for the successor to adopt within a reasonable amount of time. The new developer is then set in a *brownfield* context (cf. [3]), i.e., a context in which pre-existing systems determine the boundaries to modernisation and innovation.

2. Research Question and Methodology

This paper aims to contribute to the following research question: *How to design an expandable system capable to address future, at design-time not foreseeable, business needs for digitalisation and data analytics, whereby the implementation of such future functionality would not require a change of the system.*

By addressing this question, the work described in this paper aims to contribute towards a best practice for developers facing the challenge to engineer new systems in a brownfield context. By striving for independence of such systems from their initial developers, the newly developed systems can achieve a lifespan that extends far beyond the affiliation of the developer with the organisation. To this end, following such best practice would contribute to improved business continuance and an improved resilience of the available IT systems to hazards associated with staff fluctuation.

This research challenge is rooted in design science. The design-science research methodology recognizes as a valid contribution to science the design of novel artefacts such as prototype instantiations, the design of novel models or methods, improved instantiations or methodologies, etc. ([2], p.87). As Hevner *et al.* (ibid. pp.79-81) argue, the contribution to science must either be capable to create significant added value in form of a relevant instantiation applied into its destined environment or be a relevant addition to the knowledge base from which follow-up research can draw its rigor.

Following these recommendations on how to approach research in the domain of design science, this paper aims to contribute to the knowledge base by describing a case in which the research question as described was faced. More specifically, this paper describes the author's experiences with replacing purposely built legacy middleware systems with a modern system that has been developed in such a way that it could be expanded and reused at other regional departments of the enterprise without direct dependency on its original author.

The work was set in the context of a multinational corporation's building technology unit whose business was the development, provision, and installation of various building technology systems such as systems for air conditioning, fire safety, access control, adaptive lighting and shading technologies, etc. The customers ranged from smaller organisations such as offices or schools, to larger systems such as hospitals, airports, power plants, or research and manufacturing facilities whose complex demands for specific workspace conditions (clean room, pressurized rooms to control virus or gas hazards) could only be met by advanced building automation solutions.

The planning and installation of the solutions and the maintenance of the installed systems were executed by some 450 technicians organised in several regional branch offices. This technical staff regularly reported on their activities in order for their work time to be assigned on projects, and, in turn, charged to the customers. Various IT solutions were deployed to enable work time tracking,

and subsequent analysis of the reports to conclude about efficiency of work time utilisation, enable work optimisation, and facilitate planning for next business periods. To enable analytics of thus gathered data, business intelligence solutions have been built in the past by meanwhile retired staff, leaving behind systems that could not be maintained or adapted to changed needs any longer due to lost knowledge. Consequently, new systems were developed by a new generation of software engineers.

The remainder of this paper is structured as follows: Section 0 describes the initial (brownfield) situation in terms of the architecture of the legacy systems (section 0), discusses the design considerations for the novel architecture (section 0) and the redesigned final system landscape. Section 0 provides a summary of the paper.

3. Replacement of Legacy Systems by new Architecture

3.1. Brownfield Situation and Legacy Systems

The brownfield system landscape consisted of a regional system for reporting work on projects (the MSSC system), and a local SQL database (the Franz database) with Microsoft Access frontend in which affiliation data for the operative staff was maintained. The MSSC system fed data into the regional SAP ERP system. To obtain data from the MSSC system a local SQL database (the MSSC database) was set up and automated data transfers from the regional MSSC system to the local MSSC database were scheduled. Maintenance of the data in the Franz database was done manually – new hires were reported by e-mail to the administrator, as were any changes to contact data, and terminations of employment relationships.

Both the MSSC database as well as the Franz database and frontend can be considered "shadow IT" solutions. They were built in a response to requirements of middle management when reporting to senior management of the regional organisation, as well as to provide a technical basis to build tools to assist line managers in their role to take care of day-to-day business. A number of business intelligence and business analytics systems were built based on the MSSC and Franz databases, using Microsoft Access as the application framework and frontend of choice. A middleware system was developed (also as "shadow IT") that aggregated data from both databases and exposed it as domain objects through a REST API. This API served as a gateway to the data for a variety of process automation tools that improved efficiency of day-to-day business.

The existence of these "shadow IT" solutions is symptomatic for the inability of organisations to fully address all business needs by large systems such as SAP ERP. More specifically, SAP ERP was not able to serve as an analytics tool due to slow loading of data and the inability to model the domain objects required for complex analyses. Furthermore, SAP ERP has not been designed to orchestrate external systems as required by process automation workflows. To the rescue came homebrews of various shapes and sizes that rapidly and efficiently filled the gaps between what the officially sanctioned systems provided and what day-to-day business required. Over the course of years and decades, in-house developers conceived a shadow IT landscape of systems that could not be maintained any further once their authors retired or continued their careers elsewhere.

3.2. Considerations for new Architecture

Beside SAP ERP, large data repositories such as SAP HANA Data Lake (an in-memory database containing mirrored data from regional SAP ERP systems) were made available by the central IT

unit, as well as the data analytics platform Qlik Sense, a platform that allows developers to interactively analyse and visualise data from systems such as SAP HANA. Both the SAP HANA and the Qlik Sense platform offer impressive capabilities to deal with the available data. SAP HANA allows data to be queried by SQL, modelled into domain objects, and exposed via standard interfaces such as OData. Data loaded in SAP HANA can be processed by scripts written in JavaScript ([10], section 1.3.1). The Qlik Sense platform is able to interact with the SAP HANA system, and likewise enables developers to write scripts in JavaScript.

The SAP HANA DataLake and the Qlik Sense platform were made available at a later stage, when the legacy systems described in section 0 were already firmly established as part of the general brownfield situation. Considerations were made to replace legacy systems by the functionality provided by SAP HANA and Qlik Sense. However, restrictions to accessing data available in SAP HANA in order to process it by "shadow IT" systems required for process automation and data synchronisation with entrenched "shadow IT" systems prevented the effective reduction in the "shadow IT" landscape's complexity.

The challenge to be solved could best be described as follows: To design and develop a system that will enable the mash-up of a growing number of data sources into an internal knowledge network able to provide answers to an unknown number of queries pertinent to constantly evolving business needs. This system should seamlessly fit into the existing brownfield situation whereby existing legacy solutions should, as much as possible, be replaced by new solutions to compensate for the knowledge lost by the engineers that have left.

3.3. Development of new Architecture and Solution

The design and development of the new system was driven by an existing demand to establish a KPI[2] dashboard for senior management that would accurately display the past and ongoing performance of business units. To realise this task, data on the affiliation of staff members with business units, data on their work time and absences, and data on their work reports had to be merged, and finally assessed against set targets. This KPI dashboard was expected to give answers to questions such as how much of the worktime has been sold to clients, how much of the planned vacation has been spent already, how much of the expected yearly target has already been reached, or what the proportions of each type of work category are.

Further ad-hoc demands, such as to provide the staff headcount differences between two reporting periods, or to provide a list of external staff, came in during the design process. These spontaneous demands proved valuable in informing the design process about the type of queries the system would need to handle. The requirement to differentiate between data at different points in time called for a solution that would contain a queryable temporal dimension as opposed to simply reflect the latest state of data. The foreseeable heterogeneity of requests called for a solution that would expose a dynamic JSON[3] API[4] in which the client application would define the data model of the domain objects requested, thus adjusting size and complexity of the response to fit its specific needs.

[2] Key Performance Indicator(s)
[3] JavaScript Object Notation, a format for storing objects as text.
[4] Application Programming Interface

The initial data sources made available were, in addition to the already mentioned MSSC database, daily CSV[5] snapshots from a regional system for reporting daily work time and absences such as vacation, sick leave, etc. (the AZM), and weekly CSV snapshots from a global system containing affiliation and contact data of staff (the SCD). The amount of data thus loaded amounted to approx. 4 GB for each year. This data quantity called for an approach in which all data converted to domain objects and interlinked during a nightly load / refresh routine, then kept as interlinked domain objects in memory during run time for a rapid access to the in-memory data network.

A further requirement that emerged during the development process was the ability to process scripts on the server. This would enable server-side processing of sensitive data before sending the results of the processing to a client application. Complex queries that otherwise would require the server sending large quantities of data to the client application could thus be processed on the server in a resource-saving way.
The resulting system was an in-memory multidimensional data cube in terms of a network of interconnected domain objects with a time dimension for those objects for which temporal data has been provided. This system was named the *DataCentre*. Data in the DataCentre could be accessed through either a dynamic Web API in which the client application defined the model for the data requested (defining the fields of the requested data objects) or through Python scripts that were executed by an Iron Python interpreter on the server.

3.4. Designing for Expandability

The design and development process revealed a strong dynamic in terms of requirements that were continuously added, as well as a strong dynamic in terms of data sources that needed to be integrated to address constantly evolving business needs. The risks of this dynamic have been studied previously in the context of e-Government [7, 8] where inflexible systems were identified as a significant factor in why e-Government fails to deliver on its promise to optimise public governance and to reduce public spending (cf. [5], section 2.3).

To avoid the common pitfall of developing a system that would offer only limited functionality as known at design-time, the system was designed to allow for rapid expandability and adaptability to the context and needs of other regions, organisational units, or even other organisations, to which the system might be deployed in the future. This was achieved by three tiers of artefacts that would be designed for each context individually: (1) Data Loaders, (2) Domain Objects, and (3) Serialisation Definitions.

Figure 1 shows the architecture of the DataCentre system. It shows the modular data loaders, domain objects / regional concepts (types), and the regional type conversion & serialisation layer that defines how the API will serialise domain object type members to the data fields requested via the API. Regional Python scripts and apps process the data stored in the DataCentre to provide meaningful business applications for purposes such as process automation, digitalisation, data analytics, reporting, etc.

[5] Comma Separated Values, a format used to structure data as rows and columns.

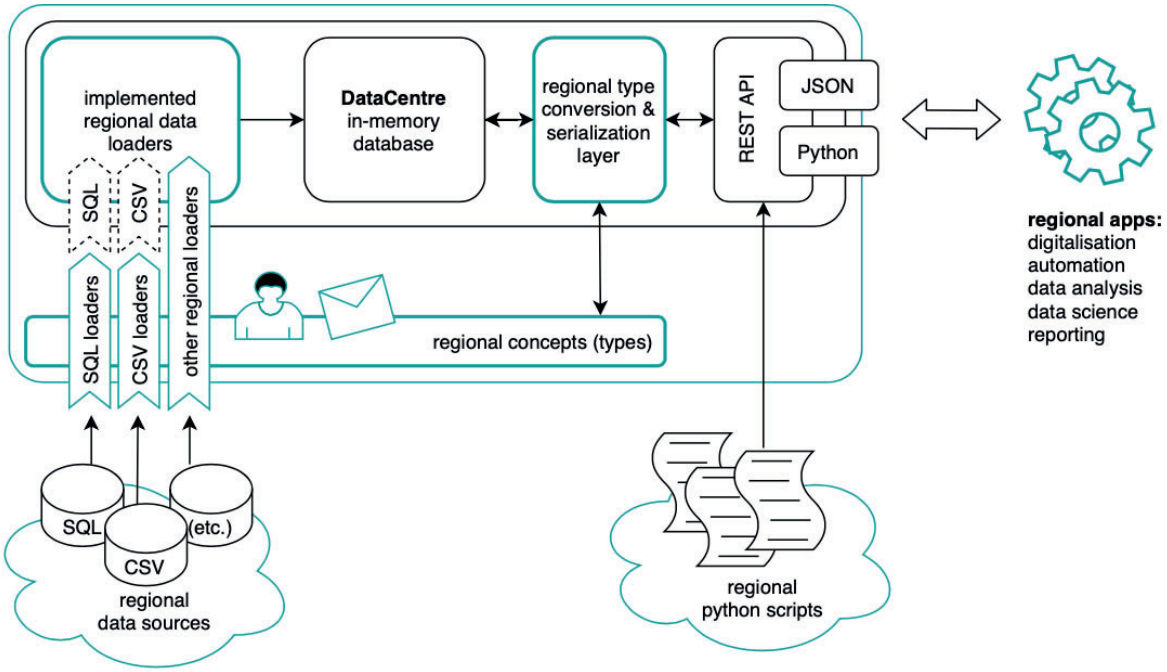

Figure 3: Data Centre system with modular concepts to create regional Data Centre instances

3.4.1. Replaceable Data Loaders

For each data source that is made available to the system, a designated Data Loader class has to be composed that extends from a common base class. The Data Loader class defines the specifics of the data source, such as location (e.g. database server, network file share, web service, etc.), data type / format (relational data obtained via SQL, CSV files, XML files, LDAP, e-Mails), as well as how the read data is converted into domain objects and how thus instantiated domain objects are interlinked with other existing domain objects in the system. The algorithms for refreshing the loaded data source, as well as everything else required for handling the data is to be defined by the data loader class.

The system can have as many data loaders as required to handle all needed and available data sources. New types of data loaders can be developed and plugged into the existing system to address an ever-evolving business need.

3.4.2. Replaceable Domain Objects

Each data loader (see above) loads the data into one or many domain object instances and interlinks the domain objects to create the knowledge network that is hosted by the system. As each environment that the system would be set in might have differing definitions of domain objects, a modular approach to add domain object types to the system has been chosen. Such domain object can be anything required by the particular domain: a person, supplier, employee, a site at which work is done, a project, work report, access credentials, and so on.
The modularity of domain objects guarantees that business need can be flexibly addressed by adding new domain objects or modifying existing ones.

3.4.3. Replaceable Serialisation Definitions

When querying data, the client application provides a set of keywords that define which data or information it is requesting. This set of keywords must be defined for each domain object separately

in form of a serialisation definition that translates between the type members of the domain object and the keywords available to the Web API and Python scripts. Typically, keywords will be directly mapped to type members of the domain object. However, also virtual keywords can be defined, which translate to functions that provide processed data when such virtual keyword is requested.
The modularity of serialisation definitions guarantees that available keywords can be tailored to fit the requirements of the context. This allows that comparable, but not fully equal concepts at different organisational units can be accessed by a shared set of keywords, while keeping the domain objects logically separate. Also, several keywords can be mapped to the same type member of the internal domain object – e.g.: keywords "mail" and "email" might both refer to the same internal representation of an e-mail address.

3.4.4. Applications

The DataCentre enabled the development of an technological ecosystem (cf. [4, 6]) that sourced its data from the DataCentre. The outbound layer of this ecosystem are the applications that access the DataCentre to address the business needs. These applications could be hosted at the server-edge in form of Python scripts, could run as independent applications sourcing data through the JSON API, or utilise a hybrid approach by consuming data pre-processed by Python scripts or tailored-to-fit native extensions to the regional DataCentre instance in form of MVC[6] controllers.

Several applications were built as part of this ecosystem that optimized existing workflows of legacy applications and enabled the development of previously impossible applications. Thus, synchronisation of the local "grey IT" database with personal data, which previously was done manually, was now automated with data from the DataCentre. The KPI dashboard mentioned above, which was the initial driver of demands for this project, was realised as a hybrid application that utilised server-edge processing by natively extending the regional system instance. Numerous other applications connected to the DataSource by means of the JSON API or Python scripts to address business intelligence, data automation, or reporting needs, as well as to use it as a pillar for access control schemes.

4. Summary and Outlook

This paper described some of the challenges that software engineers face when tasked to maintain the functionality of systems built and left behind by their parted predecessors. One of the main such challenges is the lack of documentation available to the new generation, and the incomprehensibility of the source code left behind. A common solution to resolve such legacy is to design and engineer its functionality anew. This way the new generation is able to provide and maintain the functionality for the duration of their affiliation, but once the new generation leaves, the successors face the challenge again. Taking this issue as a point of departure, the research described in this paper aimed to overcome the perpetuation of dependency on individual software engineers by developing a system that would enable succeeding generations to meet future business demands without knowledge of the system’s underlying code.

We described the design and development process of an expandable information system that would act as a knowledge base addressing an ever-evolving business need. The initial need for developing a novel system emerged as existing "shadow IT" solutions in place became unmaintainable due to retirement of the in-house developer who built them over the course of his career.

[6] Model View Controler is a design pattern for composing complex software systems.

The new system was set in a complex "brownfield" landscape, i.e., had to be built so that it remained compatible with existing systems in place. During the design and development process new requirements emerged. These requirements informed the design and development process of an inherent need for utmost flexibility and extensibility of the system. This way, the likelihood that the system would be able to serve beyond just acute requirements of an initial business situation could be increased.

The resulting system was built using a modular architecture which allowed crucial components of the system to be exchanged or expanded over time by developers without a deeper understanding of how the system internally operates. New data sources could be added as plug-ins to the system. More specifically, the algorithms responsible for loading the data, its internal representation as domain objects, and the commands to describe data to be queried via outbound interfaces (a JSON API and a Python interpreter), would be implemented in such a way as to yield a tailored-to-fit solution. Outbound interfaces were provided that allowed consuming systems to themselves describe the composition of data required.

The developed system succeeded in fitting into the existing brownfield landscape. Applications based on the new system were able to replace many of the legacy systems. Legacy systems which earlier had to be maintained manually could now be maintained automatically. Previously unavailable rapid access to the data became possible as the new system successfully joined major data sources into an in-memory knowledge network, allowing for a data collection of several gigabytes in size to be queried within milliseconds.

The approach taken in the case described in this paper has proven as a meaningful solution to overcoming dependency on the knowledge of the original authors of in-house software systems. Accordingly, the described case presents a contribution to the body of knowledge on best practice from which enterprises, public administrations, and any other types of organisations that rely on agile in-house engineering to address their dynamically changing business needs can learn how to engineer systems that are easier to maintain.

5. Bibliography

[1] HANDEL, M. J. and POLTROCK, S., 2011. Working around official applications: experiences from a large engineering project. *Proceedings of the ACM 2011 conference on Computer supported cooperative work - CSCW '11* (Hangzhou, China, 2011), 309.

[2] HEVNER, A. R., MARCH, S. T., PARK, J. and RAM, S., 2004. Design Science in Information Systems Research. *Management Information Systems Quarterly*. 28, 1 (2004), 6.

[3] HOPKINS, R. and JENKINS, K., 2008. *Eating the IT elephant: moving from greenfield development to brownfield*. IBM Press/Pearson plc.

[4] PAULIN, A., 2016. Beyond e-Government: Towards the Ecosystem. *CEEE eGov Days 2016* (2016).

[5] PAULIN, A., 2019. Digitalized Governance—An Embezzled Opportunity? *Smart City Governance*. Elsevier. 39–60.

[6] PAULIN, A., 2019. Economic Value of Technological Ecosystems. *Smart City Governance*. Elsevier. 203–216.

[7] PAULIN, A., 2019. Governing Through Technology and the Failure of Written Law. *Smart City Governance*. Elsevier. 81–108.

[8] PAULIN, A., 2015. Twenty Years After the Hype: Is e-Government doomed? Findings from Slovenia. *International Journal of Public Administration in the Digital Age*. 2, 2 (32 2015), 1–21.

[9] PELED, A., 2014. *Traversing digital Babel: information, e-government, and exchange*. The MIT Press.

[10] SAP SE 2016. SAP HANA Developer Quick Start Guide.

[11] SILIC, M. and BACK, A., 2014. Shadow IT – A view from behind the curtain. *Computers & Security*. 45, (Sep. 2014), 274–283.