The Dublo Architecture Pattern for Smooth Migration of Business Information Systems: An Experience Report

Wilhelm Hasselbring Ralf Reussner
University of Oldenburg
Department of Computing Science
Software Engineering Group
D-26111 Oldenburg, Germany
{hasselbring|reussner}@informatik.uni-oldenburg.de

Holger Jaekel Jürgen Schlegelmilch Thorsten Teschke
OFFIS
Business Information and Knowledge Management
D-26121 Oldenburg, Germany
{jaekel|schlegelmilch|teschke}@offis.de

Stefan Krieghoff
KDO
D-26121 Oldenburg, Germany
krieghoff@kdo.de

Abstract

While the importance of multi-tier architectures for enterprise information systems is widely accepted and their benefits are well published, the systematic migration from monolithic legacy systems toward multi-tier architectures is known to a much lesser extent. In this paper we present a pattern on how to re-use elements of legacy systems within multi-tier architectures, which also allows for a smooth migration path. We report on experience we made with migrating existing municipal information systems towards a multi-tier architecture. The experience is generalized by describing the underlying pattern such that it can be re-used for similar architectural migration tasks. The emerged Dublo *pattern is based on the partial duplication of business logic among legacy system and newly deployed application server. While this somehow contradicts the separation-of-concerns principle, it offers a high degree of flexibility in the migration process and allows for a smooth transition.*

Experience with the combination of outdated database technology with modern server-side component and web services technologies is discussed. In this context, we also report on technology and architecture selection processes.

Keywords: Architecture Pattern, Software Architecture, Legacy Systems, Migration.

1. Introduction

The major benefit of using multi-tier architectures for enterprise information systems is a clear separation of concerns between user interface logic (in the presentation tier), business logic (in middle tiers) and data persistence management (in the data tier). This separation of concerns yields many beneficial properties, such as independent exchangeability and adaptability of components, traceability of business requirements to the implementation in a business logic tier, and transparency of database management issues.

As these properties support information systems to accomplish the business requirements, multi-tier architectures became common practice for modern enterprise information systems. However, as most old systems are not structured in this way, but instead are usually monolithic packages of legacy code, many enterprises are faced with the problem of how to migrate these old systems to modern multi-tier architectures. Needless to say that one cannot simply neglect the existing code and replace it in one step by a new architecture:

- Legacy systems represent substantial investments that cannot simply be disposed.
- Business must go on during the transition. An enterprise cannot stop its business or stop selling its products and services for several months just for the purpose of re-organizing its internal software systems.

- Legacy software often is the unique place where business logic is "documented." Often no other documentation exists and programmers may have left the company. Hence, a complete re-implementation of business logic is usually not feasible with reasonable effort.

- Developing a new system will take considerable time. During this period the legacy system must be maintained to meet the ever changing business requirements, which implies parallel costs for both the new system and the legacy system.

Consequently, smooth transition paths and integration of legacy systems are not just nice properties of software development projects, but essential for most real-world projects in the domain of information system integration [9, 20].

In this paper, we present an architectural pattern describing the integration of typical monolithic legacy systems into modern multi-tier architectures. This pattern, named *Dublo* pattern for DUal Business LOgic, implements business logic at two places: the legacy code and new middle-tiers (usually realized on application servers). While this (partial) duplication seems to contradict a clear separation of concerns, it allows for a smooth migration path.

The pattern generalizes our specific experience and design discussions in a legacy integration project for municipal information systems. We discuss the legacy architecture as deployed before this migration project started and the involved problems such as technology selection and migration alternatives. Although any project has its peculiarities, we generalize from the presented project by discussing its re-occurring issues. Furthermore, the system architecture we started with (a database and a single 4GL-layer intermixing presentation, business logic, and data management) is somewhat representative for many existing systems to be migrated.

The main contribution of this paper is the presentation and discussion of the Dublo pattern and the exemplary description of a migration project with its typical constraints and issues to be resolved.

The paper is organized as follows. We first describe the project's context in Section 2 by discussing the domain of municipal information systems and the initial two-tier (legacy) architecture. In Section 3 we deal with typical issues of migration projects, such as technology selection and the process of architecture selection we went through. After having set the scene in these sections, in Section 4 we present the emerged Dublo pattern together with its design rationale. We describe the migration paths as enabled by the Dublo pattern. As important trade-offs we also discuss limitations and costs associated with this pattern. The process of identifying appropriate web services in our context is briefly discussed in Section 5. Section 6 presents related work, such as patterns for enterprise information systems and patterns for legacy system integration in general. Finally, Sections 7 and 8 conclude and identify some open research problems in the intersection of legacy system integration and enterprise information system architectures.

2. Project Context

This experience report discusses an architecture selection process together with the emerged architecture pattern for smooth migration of business information systems.

KDO is a software house that offers domain-specific software solutions for municipalities such as town councils and rural districts, with customers in the north-west of Germany (http://www.kdo.de). So far, their client-server information systems are mostly based on Informix databases (server) with Informix 4GL [21] and the 4Js development tool for user clients [15]. KDO supports both application service providing for their customers and installation of individual server systems at the customers' location.

KDO decided to migrate from this (monolithic) two-tier architecture towards a more flexible, standards-based, and future-oriented multi-tier architecture employing modern software engineering techniques such as component-based software development. When moving from traditional development of packaged applications towards state-of-the-art component-based software development, special care has to be taken to manage resulting risks and new challenges [31, 45].

To manage the transition towards new technology, KDO collaborates with OFFIS, the Oldenburg Research and Development Institute for Informatics Tools and Systems. OFFIS is a technology transfer institute associated with the Department of Computing Science at the University of Oldenburg, Germany (http://www.offis.de). The work presented here is a collaborative effort of the Software Engineering Group at the University, the Business Information and Knowledge Management division of OFFIS, and KDO. This cooperation is accompanied by on-the-job training of KDO employees in object-oriented modeling and Enterprise Java technologies, provided by OFFIS. This way, new software engineering methods are transferred from university research via an associated technology-transfer institute towards industrial practice.

Below, the application domain and the present client-server architecture are briefly introduced. The present paper focuses in this project setting on the migration process and the emerged migration pattern that will be discussed in Sections 3 and 4.

2.1. Municipal Information Systems

Municipal information systems support town councils, rural districts and other local authorities to accomplish the duties that are assigned to them by national law. In Germany, these duties cover approximately 9000 tasks of varying complexity which can be classified as follows, to give a glimpse of the application domain's complexity:

- Self-government of activities that are voluntarily taken by the authorities. Examples are culture and sports.
- Formal liabilities for which the way of fulfillment is not defined by specific directives. Examples are social welfare, school building, road development and construction, rescue systems, and fire protection.
- Formal liabilities for which the way of fulfillment is defined by specific directives. Examples are resident registration and all kinds of elections.

Municipal information systems are meant to improve the associated workflows, i.e. to serve the citizens and public servants in a cost-effective way more effectively, efficiently, and reliably.

2.2. Present 4GL Architecture

The present client-server architecture which is based on Informix products is displayed in Figure 1. 4GL is the language employed for coding the business logic on the server side [21]. 4Js [15] is the development tool, which allows to generate graphical user interfaces based on the Tcl/Tk Toolkit [30]. With the 4Js/4GL programming approach [49], the business logic is tightly coupled with the database management system; thus, the server side is conceived as one tier. 4GL and 4Js are proprietary development tools.

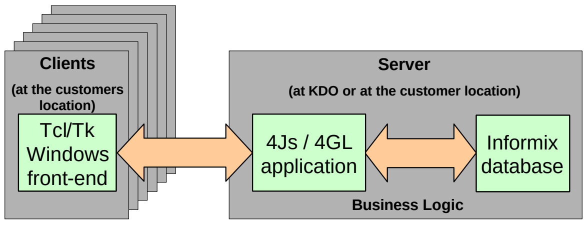

Figure 1. Present Informix two-tier architecture.

In addition to these client-server systems, Visual Basic 6.0 [27] is used for developing specific stand-alone desktop applications. This paper is only concerned with the client-server systems.

3. Architecture and Migration Process

3.1. Technology Selection

One task was a selection of the component technology for realizing the municipal informations systems. The capabilities and properties of the underlying technology are an important concern.

At present, there are only two (competing) component development platforms available that have considerable support in the market: J2EE [36] from Sun Microsystems and .NET [26] from Microsoft. The criteria that were defined for selecting among J2EE and .NET in our context are displayed in Table 1. The evaluation of each criterion is based on our specific project context, and may be different for other settings.

Criterion	.NET	J2EE
platform independence	–	+
vendor independence	– –	+
database independence	+	++
training costs	–	– –
hardware/software costs	–	– –
application service providing	○	+
zero administration of clients	○	+
documented usage patterns	–	+

Table 1. Criteria for selecting among J2EE and .NET in our project context.

Independence from specific vendors, platforms and database systems is an important concern for a guaranteed future of the new system architecture. In particular, being independent of license policies of vendors is desirable, as far as possible.

Training costs for developers are a one-time investment (continuing education is required in any case). Costs for hardware and software are not the primary issue for an application service provider such as KDO. For instance, the BEA Weblogic application server is licensed on the basis of CPUs utilized by the application server, not the number of application clients connected to the server.

Zero administration of client programs is achieved by Java Webstart [38]. So far, a similar technology is not available for .NET, but we expect that the ActiveX Controls technology [25] will become available for .NET in the future.

Standardized and established patterns are documented for J2EE systems [2, 5, 6, 10, 24, 39, 46]. This is considered an important prerequisite for designing the right architectures for our application domain. However, it can be expected that we will see patterns for .NET usage in the future [43].

In the end, J2EE was selected. At this time, J2EE is the more mature and proven technology that has broad support from major vendors such as IBM, Oracle, SAP, and Sun. Web services [23] take a loosely-coupled approach, which also allows for cooperation between J2EE and .NET implementations. Note, however, that the architecture pattern which will be presented in Section 4 is independent of the specific middleware technology that is employed for implementation.

As concrete tools BEA Weblogic [3] was selected as application server, Together [8] for modeling, and JBuilder [7] as IDE. In addition to commercial tools and system software, various open-source systems are employed, for instance CVS for version control and Linux as a server operating system.

3.2. The Process of Architecture Selection

As discussed in the previous subsection, J2EE has been selected as target technology. The J2EE standard also suggests a multi-tier architecture. A typical instance is illustrated in Figure 2 with four tiers. In this example, the client tier contains Java application clients that access the middle tier via Remote Method Invocation (RMI) [13]. This middle tier, which is an application server with a container for Enterprise JavaBeans (EJB) [36], could also be accessed by some web container, whereby the web container serves web browser requests via HTTP; for simplicity in the presentation, we ignore this alternative. This paper focuses on server-side component technology.

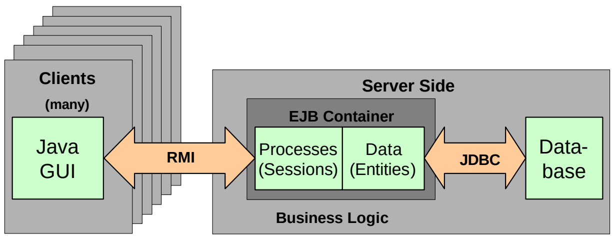

Figure 2. Four-tier J2EE target architecture.

The EJB container manages session and entity beans. Entity beans represent (passive) data objects that are stored persistently in databases. Sessions beans represent (small) business processes that access the entity beans, but do not contain persistent data themselves. These two categories of EJBs constitute two logical tiers in our target architecture: business processes and business objects. EJBs are further categorized as stateful/stateless session beans and container/bean managed entity beans, as well as the additional category of message-driven beans. This additional classification is not relevant for our discussion.

The benefits of multi-tier architectures are well presented in the literature (e.g., [11, 48]). Historically, multi-tier architectures arose from client-server systems. With widely available low priced PCs, system architectures moved from monolithic mainframe systems to client-server systems. However, often the client presentation layer simply used a terminal emulation without changing the server (mainframe) side. However, the benefits of separating a dedicated presentation tier are convincing, in particular as various (remote) client systems could be developed more or less independently from the mainframe development process. As a next step, one realized that similar benefits arise from decoupling data management and business logic. This resulted in the state-of-the-art three-tier architectures. By the identification of different business-logic layers (i.e., further structuring the middle tier) and/or introducing additional data access layers (i.e., identifying additional layers in the data tier), multi-tier architectures emerged.

In particular, modern middleware platforms with their provision of application servers for server side components (such as J2EE and .NET) and pre-defined data access methods through containers and standardized data access techniques (ODBC, JDBC, etc) support multi-tier architectures as an underlying technology.

Now, the question arises of *how* to migrate from the two-tier legacy architecture in Figure 1 towards the four-tier target architecture in Figure 2. Figure 3 illustrates one of our first approaches for a migration architecture, exemplary for our motor vehicle registration system, which is an Informix 4GL/4Js application, as discussed in Section 2.2. During the migration phase, both the old 4GL/4Js Informix user interface clients and the newly developed Java clients are able to co-exist.

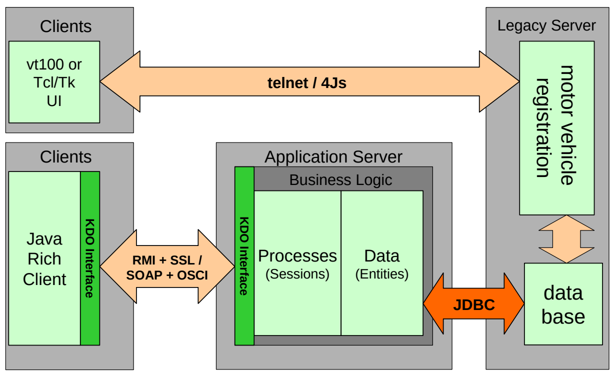

Figure 3. First approach for a migration architecture.

The communication between Java clients and application server is further refined with specific “KDO interfaces” to support different forms of technical communication: RMI for fast communication on secured channels and Web Services over SOAP with OSCI (Online Services Computer Interface) [29] for a certified secure communication over insecure channels such as the Internet. A detailed discussion of these security measures is beyond the scope of the present paper and not really relevant for our architecture discussion. It is just important to note that security measures are relevant on the architectural level, and that they are resolved in our specific architecture as far as required in the application domain of municipal information systems according to German law [22].

The communication between application server and the legacy server is the critical issue in this architecture. The approach of Figure 3 to access the legacy database directly via JDBC poses various problems. The grown database structures in legacy systems do not reveal all relevant semantics of the stored data objects that are relevant to use them correctly. Additionally, those structures usually do not fit the newly defined business objects on the middle tiers.

So, our second approach was to store those data objects, which are implemented with the new technology, in a new database, as illustrated in Figure 4. This approach has the obvious, and highly critical disadvantage of requiring consistency mechanisms for data that is replicated in both the old and the new database [18, 19].

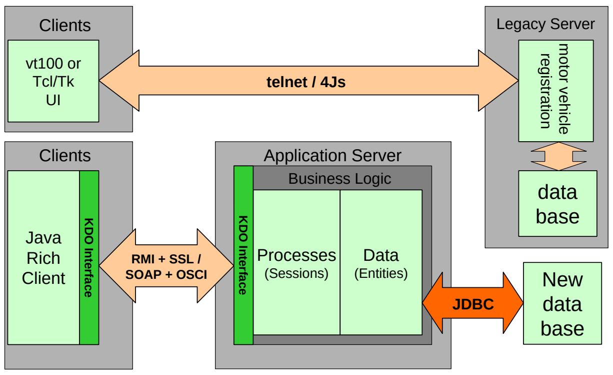

Figure 4. Second approach for a migration architecture.

These considerations lead us to the finally selected migration architecture that is displayed in Figure 5. For legacy information systems it becomes apparent that their internal databases should never be accessed directly. It is advisable to access them via some kind of application programming interface that “knows” their business logic.

We decided to use SOAP as a communication protocol which consequently requires specific adapters at both the EJB and 4GL sides. The Simple Object Access Protocol (SOAP) [23] is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. The web service technology is based on SOAP as communication protocol, the Web Service Definition Language (WSDL) for interface description, and the Universal Description, Discovery and Integration (UDDI) protocol to dynamically find and use web services over the Internet [23].

Other communication mechanisms would be possible in this context, such as JCA [32].

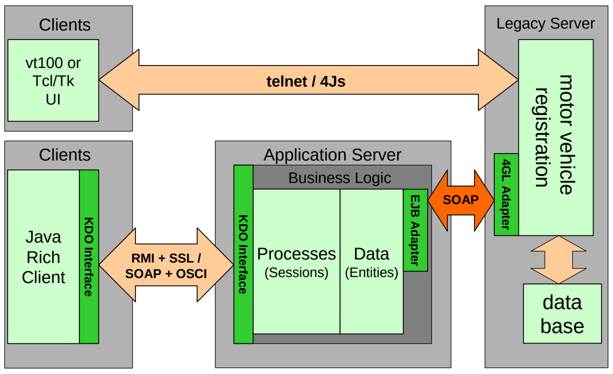

Figure 5. Finally selected approach for a migration architecture.

This preferred approach of accessing legacy information systems via application programming interface is well known from many other projects where legacy systems had to be integrated [28, 48].

4. The Dublo Pattern: DUal Business LOgic

The presentation of the Dublo pattern follows the way architecture patterns are described in [12]: first a definition of problem and context is given, followed by a description of the solution structure, and completed with a discussion of limitations.

4.1 Problem and Context

Legacy systems often differ from state-of-the-art enterprise architectures as they do not distinguish different tiers. Hence, presentation logic is often intermixed with business logic and database access code. With the advent of multitier architectures, separating these different concerns became common practice. However, one of the main reasons for the fact that many legacy systems do not separate these layers is that the system or language used to implement the business logic also offers powerful data access functions as well as the possibility to implement user interfaces (either textual interfaces or graphical user interfaces generated by tools). Remarkably, neither COBOL nor more current fourth generation programming languages [49] differ in this respect (besides differences in user interface design). We discussed such an example architecture in Section 2.2.

These programming systems allow for the integration of presentation, business logic and data logic in one single layer. As discussed in Section 3, one is now interested in migrating these systems into multi-tier architectures. This migration process strongly depends on the way (and whether) the legacy system is integrated into the new system. Different solutions for integration and migration exist:

- “Big-bang” replacement of the old monolithic system by a multi-tier architecture: Obviously this “strategy” is only applicable for very small systems which operate in a well documented fashion and in a well understood domain. As this is rarely the case for most legacy systems, this strategy is usually not considered for good reasons. However, if its preconditions apply, it has the benefit that it is simple to manage and requires no redundant code to write.

- Replacement of client and business logic at one point in time, direct access of the newly introduced middle tier to the database: This strategy keeps the old database and replaces the old combined presentation / business / data access layers by separate presentation and business logic layers. On the positive side, this strategy immediately results in a three-tier architecture with well separated concerns for presentation, business logic and data access. However, on the negative side, this approach necessitates the full replacement of all the business and presentation logic. As the business logic in the majority of cases represents a major cost factor and constitutes the kernel of enterprise information systems, a complete substitution in one step is hardly achievable.

- Keep business logic in legacy code, add new business logic to the new middle tier, access database via adapter to the legacy code: this strategy allows the reuse of the existing legacy business logic to some degree. This approach separates concerns in multi-tier architectures to a lesser extent than the previous approaches, since it keeps business logic at two places: the old legacy system and the new business logic tier. Anyway, this approach offers the smoothest migration strategy, as it decouples the development of new business logic and the new presentation layer from the operation of the existing system. This is the Dublo pattern as presented in the following subsection.

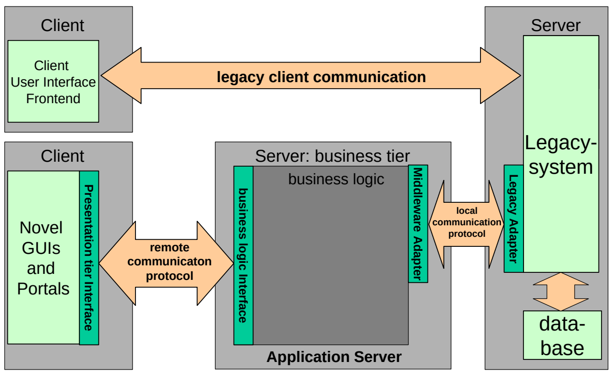

Figure 6. Structural View on the Dublo Pattern

4.2 Solution

4.2.1 Structure

The Dublo solution structure is illustrated in Figure 6. The basic idea is: Formulate business logic in a new business logic tier; write a legacy adapter for access by the new business logic to the existing legacy business logic; use this adapter for database access. Consequently, the database is accessed only via the existing legacy (business) code. This existing code acts as a functional access layer to the database. Functionality implemented in the new business logic tier is managed by a new presentation tier.

4.2.2 Migration

In the Dublo pattern old business logic and existing user interfaces can be reused as long as they provide functionality useful in the new application context. The old logic can be replaced step-by-step by a the new business logic tier. In many cases, the replacement of old user interfaces by integrated new client technology is at least as important as new business logic. The Dublo pattern supports the fast update of client user interfaces by wrapping legacy business logic by means of an adapter. Hence, the new business tier can simply pass through requests of a new presentation tier to the legacy system without implementing the business logic itself. This bypass to the old legacy code by the new business logic tier decouples the development of the new presentation tier from porting legacy code to the business logic tier. Since the transition from legacy textual user interfaces to new graphical user interfaces in a presentation tier is often of high relevance, the migration towards a new presentation tier is discussed below in greater detail.

4.2.3 Benefits

The application of the Dublo pattern is reasonable, if the following benefits provided by this pattern are of relevance:

- Smooth migration: incremental replacement of old business logic and client software by new middle-tier business logic. In particular, if the migration cannot be performed in a short time interval, this issue becomes essential.
- Database consistency: as no additional database is introduced, no consistency or updating problems between new and legacy databases arise.
- Database independence: a new DBMS can be introduced without changing the middle tier. However, this may not be possible with all legacy systems.
- Reuse of existing legacy business logic by accessing it through an adapter and possibly forwarding it to middle tier.

The consequences of the latter advantage deserve some attention: Forwarding access to the old legacy system through the new business tier results in transparency for modernized clients on whether business logic is already implemented in the new middle tier or still in old legacy code. Therefore, the transition of old user interfaces (implemented in legacy code and intertwined with data access code and business logic) toward a new presentation layer can be done in three steps:

1. Retain less frequently used old user interfaces: When considering typical usage profiles, only a fraction of all user interface forms is used frequently. As many business processes are executed in a regular, but infrequent manner (such as inventory updates, annual or quarterly financial statements, etc), the corresponding user interface forms are also used infrequently. In the Dublo pattern these user interface forms do not have to be changed as long as the underlying legacy business logic is valid.
2. Replace old user interfaces by new user interfaces in the presentation tier without re-implementing the business logic. The new interfaces can access a proxy of business logic in the business logic tier which simply forwards the calls to the existing legacy code.
3. Replace legacy business code by new business logic. As the new business logic adheres to the interface between business logic tier and presentation tier, this transition is transparent to the user interface code.

Note that the two latter steps do not have to take place in parallel. This is a result of the aforementioned decoupling of business code and presentation code development, gained by keeping the legacy code. This decoupling gives a new degree of freedom in the migration process.

The existence of business logic at two places (the old legacy system and the new business logic tier) does not allow for the clear separation of concerns as promised by multi-tier architectures. Alternatively, and in contrast to the Dublo pattern, one could access the legacy database directly. While this still preserves the benefit of not duplicating databases (and hence avoiding consistency problems), in the following we discuss why we argue for the functional access layer and the presence of legacy code:

- Possibility of exchanging old DBMS (while keeping the legacy layer). Even when directly accessing the database, it is reasonable to use an adapter, since adding new code to a legacy database is usually problematic.
- Reuse of existing business logic implemented in the legacy system. As motivated before, the complete immediate transition to a new system is in any large system impossible, hence keeping parts of the old code and following a migration path recommended by a pattern is beneficial.
- If the legacy system makes use of a DBMS which allows for direct access, the possibility to access the database directly without using legacy code still remains as an option. This is the case because the Dublo pattern uses an adapter between the new business logic tier and the legacy layer.

4.3 Comments and Limitation

The application of the Dublo pattern is greatly simplified by the existence of a functional access layer to databases in legacy code. If this functional access layer is absent, the effort of its addition has to be justified by the benefit of reusing legacy code.

In general, the degree of reuse is application specific and underlies many different concerns. However, it is an advantage of this pattern to be applicable with various levels of legacy business logic reuse.

Application of the Dublo pattern will typically result in some redundant code and extra effort compared to an immediate transition of all legacy business logic:

- The adapter between legacy layer and new business logic tier.
- The functional access layer within the legacy system for database access (if not present in the legacy system).

- Performance. When adding layers, you usually reduce the performance of a system. So far, the experience shows that the overhead is acceptable in our context. The gained flexibility is more important.
- Possible duplication of legacy code. As new system requirements (in particular non-functional requirements) may have a radical influence on the business logic design, it may happen that old legacy code cannot be reused and has to reimplemented immediately in the business logic tier.

Of course, the latter issue is a well-known problem for many legacy integration projects. Hence, the costs imposed by the previous factors limit the use of the Dublo pattern.

5. Identifying Legacy Web Services

When integrating legacy systems it is a challenging problem to identify chunks in the business logic of the legacy system that can become the components or the objects in the new business logic. The alternative approaches for identifying (web) services may be categorized as follows [47]:

1. *Data driven*: Identifying services based on the legacy data structures.
2. *Function driven*: Identifying services based on the (business) functions realized by the legacy system.
3. *Object driven*: Specification of a new object-oriented model of the legacy's data and functions.

In our context, data and object driven approaches [44] are not really appropriate: the legacy data structures have degenerated over time. Instead we selected a function-driven approach [35], whereby use cases in the legacy system are identified based on user interaction with the systems. For details, we refer to [42].

6. Related Work

All aspects of software systems, their development and their deployment are suitable topics of individual patterns or comprehensive pattern languages. The design patterns of Gamma et al [17] are patterns for object-oriented design and programming, often relying on inheritance from abstract classes. Our Duplo pattern is an architecture pattern, whereby relationships among components, not among classes, are described. Inheritance is a problematical relationship for components [40].

Fowler et al [16] present a set of patterns for enterprise application architectures organized around the presentation, business logic, and data tier. These patterns present a very good overview of different design alternatives and their solutions with their benefits and disadvantages as well as descriptions of their contexts of applicability. The integration of legacy code is not treated specifically.

The process and patterns described in Adams et al [1] for e-business applications are also applicable to our domain of enterprise application systems in general, but do not focus on legacy system integration. However, their "application integration pattern" could be used for integrating legacy applications with new code. Anyhow, legacy integration aspects, such as migration paths, etc. are not of major concern.

Emmerich et al [14] present the TIGRA pattern for enterprise application integration. This pattern is suitable for integrating one or several legacy application(s) with one or several new application(s). Its major concern is the minimization of the number of adapters by introducing an intermediate common data format. This work is related to our context, since their pattern (called architectural style [33]) also emerged in an industrial project context, but the solved problem is different.

Stevens and Pooley [34] coin the term "re-engineering patterns" emphasizing on *process* patterns for migration paths. Their patterns concern modularization, compiler-motivated restructurings (as externalization of internal representations or backend abstraction) and interface changes. The description of a migration path instead of a target architecture reflects the importance of the migration. However, in this paper we prefer the other way around: the description of the structural architecture pattern emphasizes the degrees of freedom for migration paths. In contrast to the re-engineering patterns, we focus on the problem of making the transition from a typical monolithic information system to a modern multi-tier architecture.

Bergey et al [4] describe a framework for the "disciplined evolution of legacy systems." This framework provides guidelines (mainly in the form of checklists) on how to transform a legacy system into a modern target architecture and discusses the various driving forces to be resolved during system evolution. The framework takes a comprehensive view on legacy system evolution, but does not concentrate on specific legacy or target architectures.

7. Summary

The Dublo pattern was presented as an architectural pattern easing the migration from monolithic legacy systems towards modern multi-tier architectures in the domain of enterprise information systems, exemplary for municipal information systems. The benefits as well as the limitations, and the application domain are discussed. The pattern was applied in several legacy system integration projects, of which one was described in greater detail. Besides the pattern and the related migration strategies, we report also on technology and architecture selection processes.

To summarize: The emerged Dublo pattern does not really describe a complex or astonishing architectural style. We think that this is a desirable property of any pattern. However, the presented rationale for the pattern is not as simple as the pattern itself.

8. Future Work

After migrating to a multi-tier architecture the development of an enterprise information system does not end. New requirements emerge in our project context, such as:

Portal technology: Currently, the municipal information systems at KDO are operated by the public servants on behalf of the citizens. Many tasks, such as registering for dog license fees, could be done directly by the citizens via the Internet without moving to the town halls. Our middle tier, which is an application server, could then be accessed by some web container, whereby the web container serves a web browser at the citizen's home. The employed OSCI (Online Services Computer Interface) [29] allows for a certified secure communication. The advantage of our flexible architecture is that it allows for such extensions.

Automatic generation of GUIs: Porting user interfaces is challenging for several reasons. Firstly, users need GUIs offering ease of use and an integrated view on different information services. Furthermore, the sheer number of forms to be translated into modern GUIs causes problems. For example, at KDO approximately 2000 user interfaces have to be migrated form 4GL/4Js to Java. Any tool support for generating user interfaces from an abstract specification would be of great help. Portal technology could also allow for displaying the legacy Tcl/Tk GUIs of 4Js applications in a web browser, similar to embedding Java applets in web pages [41].

Migration from Visual Basic for desktop applications: J2EE is a server-side component technology. An alternative for stand-alone desktop applications is a transition towards the J2SE [37] with the advantage of employing Java throughout the product portfolio. However, we expect that both Java and .NET will co-exist and interoperate (and compete) in the medium-term future such that we do not need to commit ourselves immediately to only one technology.

During the course of our projects, we identified the following areas requiring further research:

Architecture selection: The selection of an architecture can be seen as a multi-dimensional optimization problem. Several factors have to be balanced. In addition to functional requirements, one also has to balance non-functional requirements, such as maintainability (in particular, extensibility), performance and availability. While modern development processes offer a more or less systematic way of refining functional requirements into system designs, methods for systematically dealing with non-functional properties are still missing.

Architecture migration: Meanwhile, designing new systems according to functional requirements is well understood. In practice, most software development projects have to face constraints caused by the existence of legacy systems to be integrated. Therefore, development strategies have not only to deal with new requirements but also need to take legacy systems into account by offering migration strategies.

Adapter construction and generation: On the technical side, many adapters are required to integrate different data formats of streams and files, for bridging technical incompatibilities caused by proprietary programming systems and platforms. Incompatibilities have to be bridged by adapters. Adapters are also used for encapsulating design decisions. However, most adapters consist of relatively simple code or are just parameterizations of templates. Although template parameterization as an adapter generation mechanism can be supported by tools, additional research for (semi-) automated adapter generation is expected to reduce the costs for adaptation. Standards-based adapters may help, but legacy systems usually do not follow standards.

References

[1] J. Adams, S. Koushik, G. Vasudeva, and G. Galambos. *Patterns for e-business: A Strategy for Reuse*. IBM-Press, 2001.

[2] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2nd edition, 2003.

[3] BEA Systems, http://www.bea.com/products/weblogic/. *BEA Weblogic*. Retrieved 2003-08-25.

[4] J. K. Bergey, L. M. Northrop, and D. B. Smith. Enterprise framework for the disciplined evolution of legacy systems. Technical Report CMU/SEI-97-TR-007, Carnegie Mellon University/Software Engineering Institute, Oct. 1997.

[5] C. Berry, J. Carnell, M. Juric, M. Kunnumpurath, N. Nashi, and S. Romanosky. *J2EE Design Patterns Applied*. Wrox Press, 2002.

[6] A. Bien. *J2EE Patterns*. Addison-Wesley, 2002.

[7] Borland Software Corporation, http://www.borland.com/jbuilder/. *JBuilder*. Retrieved 2003-08-25.

[8] Borland Software Corporation, http://www.borland.com/ together/. *Together*. Retrieved 2003-08-25.

[9] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and The Incremental Approach*. Morgan Kaufmann, San Francisco, 1995.

[10] D. Broemmer. *J2EE Best Practices: Java Design Patterns, Automation, and Performance*. Wiley, 2002.

[11] A. W. Brown. *Large-Scale Component-Based Development*. Prentice Hall, Englewood Cliffs, NJ, USA, 2000.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley & Sons, New York, NY, USA, 1996.

[13] T. Downing. *Java RMI: Remote Method Invocation*. Hungry Minds, 1998.

[14] W. Emmerich, E. Ellmer, and H. Fieglein. TIGRA: An architectural style for enterprise application integration. In *Proc. 23rd International Conference on Software Engeneering (ICSE-01)*, pages 567–576, May 2001.

[15] Four J's, http://www.4js.com. *Four J's Development Tools*. Retrieved 2003-08-25.

[16] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, editors. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, 1996.

[19] W. Hasselbring. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208, Nov. 1997.

[20] W. Hasselbring. Information system integration. *Communications of the ACM*, 43(6):33–38, 2000.

[21] IBM, http://www-3.ibm.com/software/data/informix/tools/4gl/. *Informix 4GL product family*. Retrieved 2003-08-25.

[22] KBSt, http://www.kbst.bund.de/saga. *Standards und Architekturen für eGovernment-Anwendungen (in German)*. Retrieved 2003-08-25.

[23] D. Linthicum. *Next Generation Application Integration: From Simple Information to Web Services*. Addison-Wesley, 2003.

[24] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Wiley, 2002.

[25] Microsoft Corporation, http://www.microsoft.com/com/tech/activex.asp. *ActiveX Controls*. Retrieved 2003-08-25.

[26] Microsoft Corporation, http://www.microsoft.com/net/. *Getting startet with .NET*. Retrieved 2003-08-25.

[27] Microsoft Corporation, http://msdn.microsoft.com/vbasic/. *Visual Basic*. Retrieved 2003-08-25.

[28] H. Niemann, W. Hasselbring, T. Wendt, A.Winter, and M. Meierhofer. Kopplungsstrategien für Anwendungssysteme im Krankenhaus (Coupling strategies for application systems in hospitals). *Wirtschaftsinformatik*, 44(5):425–434, 2002.

[29] OSCI, http://www.osci.de. *Online Services Computer Interface*. Retrieved 2003-08-25.

[30] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[31] T. Ravichandran and M. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, Aug. 2003.

[32] R. Sharma, B. Stearns, and T. Ng. *J2EE Connector Architecture and Enterprise Application Integration*. Addison-Wesley, 2002.

[33] M. Shaw. Comparing architectural design styles. *IEEE Software*, 12(6):27–41, Nov. 1995.

[34] P. Stevens and R. Pooley. Systems reengineering patterns. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 17–23, New York, Nov. 1998. ACM Press.

[35] E. Stroulia, M. El-Ramly, and P. Sorenson. From legacy to web through interaction modeling. In *Proceedings of the International Conference on Software Maintenance (ICSM '02)*, pages 320–329. IEEE Press, Oct. 2002.

[36] Sun Microsystems, http://java.sun.com/j2ee/. *Java 2 Platform, Enterprise Edition (J2EE)*. Retrieved 2003-08-25.

[37] Sun Microsystems, http://java.sun.com/j2se/. *Java 2 Platform, Standard Edition (J2EE)*. Retrieved 2003-08-25.

[38] Sun Microsystems, http://java.sun.com/products/javawebstart/. *Java Webstart*. Retrieved 2003-08-25.

[39] Sun Microsystems, http://java.sun.com/blueprints/patterns/. *Sun Java Center J2EE Patterns*. Retrieved 2003-08-25.

[40] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition edition, 2002.

[41] Tcl Developer Exchange, http://www.tcl.tk/software/ plugin/. *Tcl/Tk Web Browser Plug-in*. Retrieved 2003-08-25.

[42] T. Teschke, H. Jaekel, S. Krieghoff, M. Langnickel, W. Hasselbring, and R. Reussner. Funktionsgetriebene Integration von Legacy-Systemen mit Web Services (Function-driven integration of legacy systems with web services). In *Proc. Workshop Enterprise Application Integration (EAI 2004)*, pages 19–28. GITO Verlag, Feb. 2004.

[43] C. Thilmany. *.NET Patterns: Architecture, Design, and Process*. Addison-Wesley, 2003.

[44] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE '99)*, pages 246–255, Los Angeles, USA, May 1999. ACM.

[45] P. Vitharana. Risks and challenges of component-based software development. *Commun. ACM*, 46(8):67–72, Aug. 2003.

[46] M. Völter, A. Schmid, and E. Wolff. *Server Component Patterns: Component Infrastuctures illustrated with EJB*. Wiley, 2002.

[47] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97)*, pages 24–32, Oct. 1997.

[48] I. Wijegunaratne and G. Fernandez. *Distributed Applications Engineering: Building new applications and managing legacy applications with distributed technologies*. Springer-Verlag, London, 1998.

[49] W. Wojtkowski and W. Wojtkowski. *Applications Software Programming With Fourth-Generation Languages*. Wadsworth Publishing, 1990.