Web Data Integration for E-Commerce Applications

Wilhelm Hasselbring
University of Oldenburg, Germany

To make them flexible, scalable, and useable, e-commerce applications require systematic development, including data integration. Traditionally, the integration proceeds in a bottom-up way. This article discusses the problems and proposes a top-down approach to overcome some of the problems. A combined yo-yo approach aims to exploit both strategies' benefits.

E-commerce is an application domain in which integrating Web data from different information systems is a central problem.[1] For e-commerce applications, often interorganizational processes must be supported, whereby the individual information systems are highly autonomous, making the integration process an even more challenging task. For a scalable integration of autonomous, heterogeneous information systems' data, engineers need a systematic process.

Traditionally, heterogeneous information system integrations proceed in a bottom-up way—we analyze the information models of existing (legacy) systems and subsequently integrate them. One result of the traditional bottom-up approach is that the overlaps among the component system's models—not the global applications' requirements—determine the (merged) integrated data models' structure. The maintenance of such integrated models is a problem, because the merged models rapidly become very complex; usually more complex than required for the actual integration goals. This situation can lead to severe scalability problems with respect to execution performance, usability, and maintenance.

Another way to approach heterogeneous information systems' integration is a top-down process. Starting with common models based on the requirements of new (global) applications and on domain-specific standards, we can integrate the individual component models into these common models. Still yet, another approach—called the yo-yo approach—combines bottom-up and top-down strategies in a way that exploits their benefits and serves as a migration path from the traditional bottom-up approach toward an ideal top-down approach.

Traditional bottom-up integration

One characteristic result of the bottom-up construction is that the component models' overlappings greatly determine the integrated model structure. For instance, many object-oriented integration approaches resolve semantic overlappings by introducing generalized classes so that the original inheritance hierarchies are subhierarchies of the resulting merged hierarchy (upward-inheritance principle[2]). These approaches take over the inheritance hierarchies from the component models to the integrated model and adapt them to each other. Consequently, the resulting merged hierarchies may become needlessly complex.

I illustrate these problems by examining two sample national banks using bottom-up database integration. These imagined banks aim at a joint venture to create an international virtual bank that offers banking services on the Internet marketplace based on their local services and systems. The first local component system stores account information for a Dutch bank, in which Dutch terms are used in the model (see Figure 1b). The second component system stores account information for a German bank and uses German terms (see Figure 1c). For simplicity, I only modeled basic account information in this small example; I omitted information on account holders and other related business objects.

The Dutch bank offers checking and savings accounts (*Privérekening* and *Sparrekening*, respectively). The structure of the German bank's model is similar, but it offers two kinds of savings accounts (bank books and fixed time deposits). In Germany, each bank has an identification number (*Bankleitzahl*), which is absent in the Dutch system where the bank is identified via the account number. The German bank uses a legacy system that only supports one currency (*Deutsche mark*), whereas the Dutch bank has a currency attribute (*Valuta*) to distinguish between currencies (*Dutch florin* and *Euro*). (Throughout this article, for an English translation of any German or Dutch terms, please refer to the "Banking Terminology" sidebar on p. 18.)

To integrate these local database systems as the basis for an international virtual bank, we need to

1070-986X/02/$17.00 © 2002 IEEE

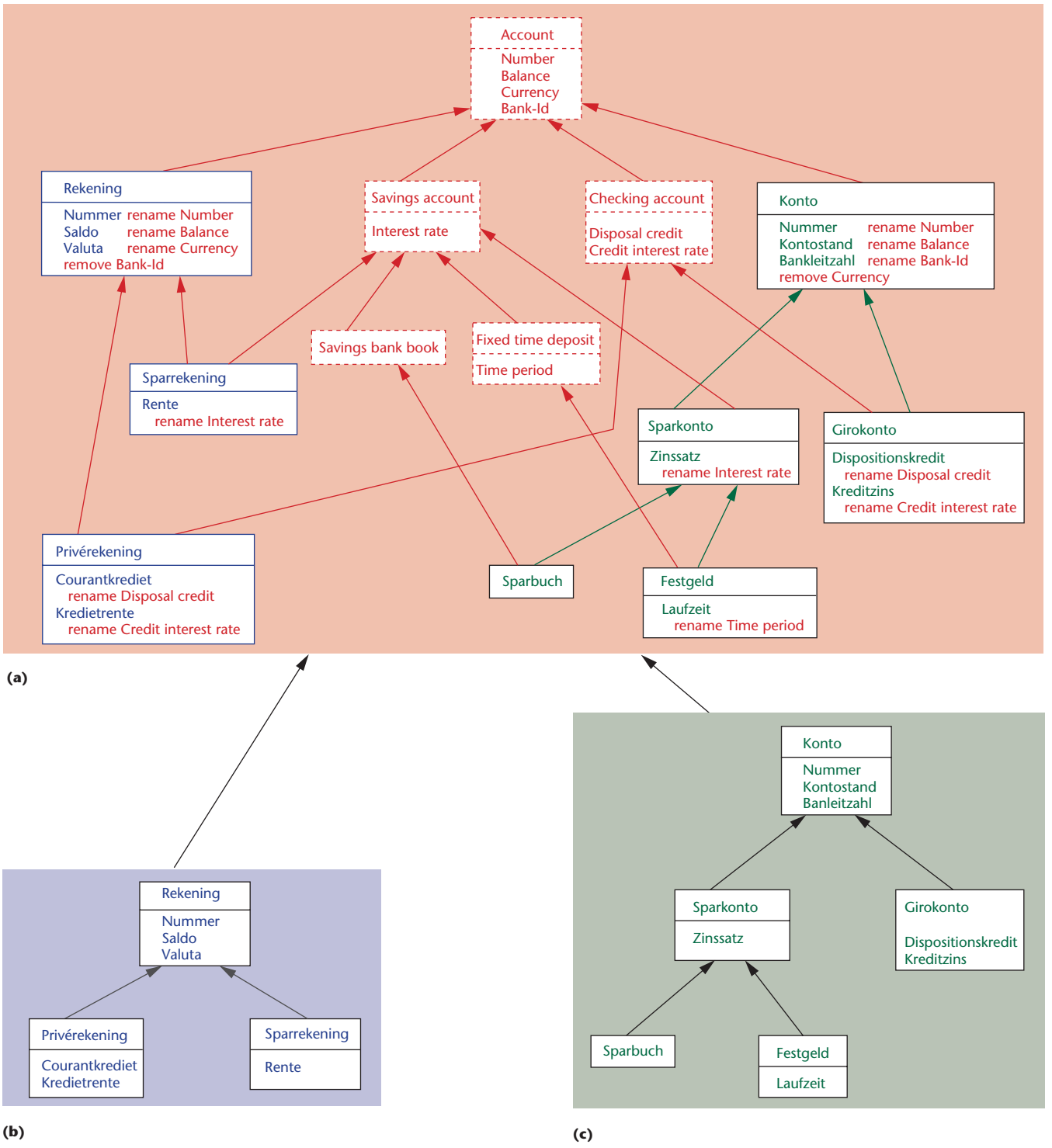

Figure 1. Component models of the local (b) Dutch and (c) German bank databases as well as the integrated model of the virtual bank as a result of (a) merging. I added the dashed elements in (a) to the component models to obtain an integrated model for the virtual bank. For this example, I assume that the Rekening and Konto extensions of both databases overlap so that the corresponding merged class Account represents the union of the component classes. Remove specifies global elements that aren't available locally. The Unified Markup Language notation specifies the models for class diagrams.[3]

analyze the overlaps among the classes we're integrating.

By applying the upward-inheritance principle,[2] we can construct an integrated model with 13 classes (Figure 1a), whereas the German database contains only five classes. The three classes for the Dutch database are almost identical to a subset of the German classes. Because of the upward-inheritance principle, the integrated model contains more classes than we need. To reduce this complexity within the integrated model, we may merge classes in the following ways:[4]

- *Vertical merging* involves merging a class with its direct superclass. In Figure 1a, for instance, we can merge the classes Account and Rekening vertically. Then we must record the renaming of attributes and classes somehow. This vertical merging changes Bank-Id into an optional attribute within the merged class (while in the local databases all attributes are mandatory—for example, Null values aren't allowed).

- *Horizontal merging* involves merging classes that have a direct specialization relation to the same superclass. In Figure 1a, the classes Rekening and Konto could be merged horizontally into a class that inherits from the class Account. This horizontal merging would change both Bank-Id and Currency into optional attributes within the merged class. Furthermore, we must introduce an additional integrity constraint, which requires at least one of the attributes Bank-Id and Currency in the merged class to hold a valid value.

We can combine the horizontal and vertical merging. Figure 2 shows the optimized model of the virtual bank with six classes. In this example, the high amount of class merging is possible

Banking Terminology

Dutch	English
Privérekening	Checking account
Sparrekening	Plain savings account
Valuta	Currency
Nummer	Number
Saldo	Balance
Rente	Interest rate
Courantkrediet	Disposal credit
Kredietrente	Credit interest rate

German	English
Bankleitzahl	Bank identification number
Festgeld	Fixed time deposit
Sparbuch	Savings bank book
Sparkonto	Generic savings account
Zinssatz	Interest rate
Girokonto	Checking account
Dispositionskredit	Disposal credit
Kreditzins	Credit interest rate
Nummer	Number
Kontostand	Balance
Laufzeit	Time period

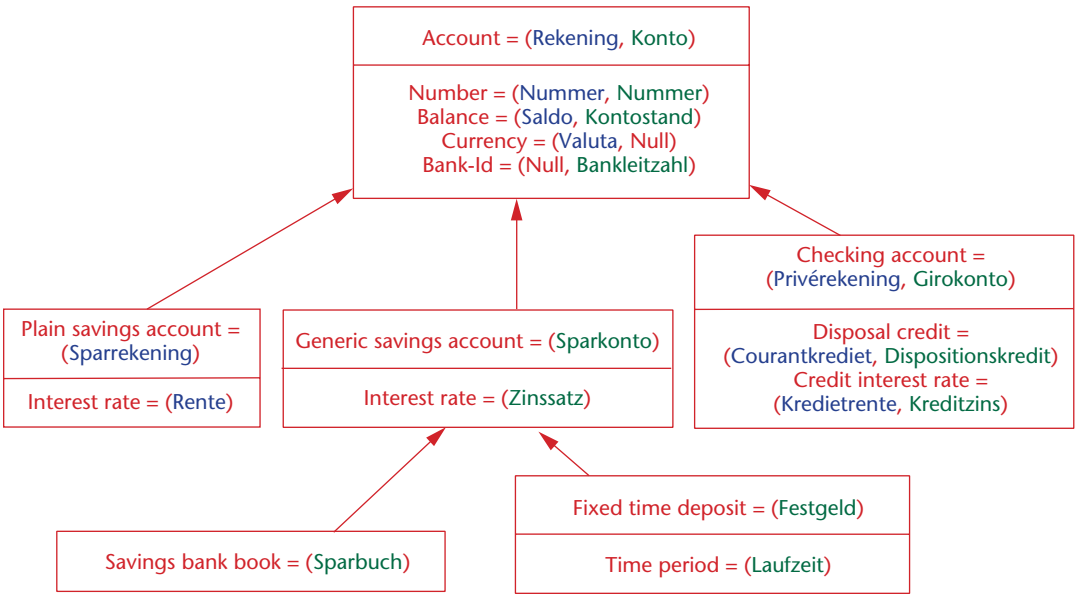

Figure 2. The optimized integrated model for the virtual bank. In Figure 1, we can't merge Sparrekening with Savings account, because the Savings bank book isn't a specialization of the Dutch savings account.

because the two integrated models cover almost the same information in a similar structure, which is unlikely in scenarios with large numbers of systems to integrate. Also, the component models' structure determine the integrated model's structure and, because of the merging, this imposes peculiar constraints on the integrated model.

Under those traditional bottom-up integration paradigms, the integrated model depends directly on the source models. An integration engineer defines the desired integrated models by examining all the existing systems to be merged. In the bottom-up approach, the system accumulates the capacity of existing information systems in one global model. As a result, the merged model's usability and maintainability can become a serious problem.

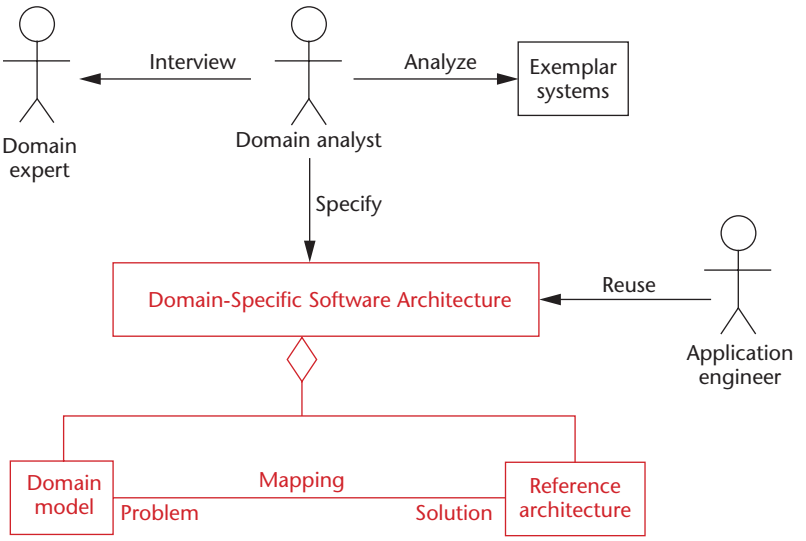

Figure 3. Relations between some roles and artifacts in the DSSA engineering process. Hollow diamonds indicate part-of relations. We use the UML notation for actors to model the roles.[3] For instance, Domain Analysts specify the DSSA, based on interviewing Domain Experts and analyzing exemplar systems in the domain. Application Engineers then reuse the DSSA, which consists of the Domain Model and the Reference Architecture.

Domain-specific top-down integration

To approach a more ideal top-down integration process, let's start with a look at domain-specific software development. Domain engineering is an activity for building reusable components, addressing the systematic creation of domain models and architectures. Domain engineering supports application engineering, which uses the domain models and architectures to build concrete systems. The emphasis is on reuse and product lines. The Domain-Specific Software Architecture (DSSA) engineering process promotes a clear distinction between domain and application requirements.[5] A DSSA consists of a domain model and reference architecture, as Figure 3 shows. The DSSA process consists of domain analysis, architecture modeling, and design and implementation stages, as Figure 4 illustrates.

Domain models represent the set of requirements common to systems within a specific domain. Usually, we can group those systems into product lines such as the insurance or banking domain. There may be many domains, or

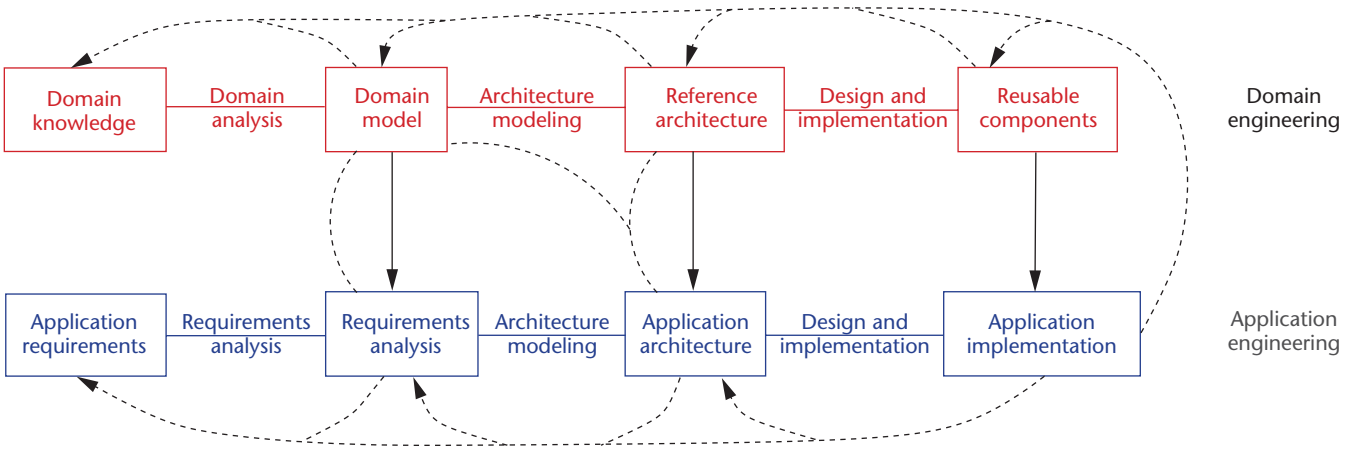

Figure 4. The DSSA engineering process. In application engineering (bottom), we develop software systems from reusable components created by a domain engineering process. Essentially, this is a classical waterfall process for software development with feedback cycles. The domain engineering process (upper part) supports the application engineering process. As indicated by the dashed arrows, various forms of feedback are possible. However, this figure doesn't address real-time dependencies. Typically, the two tracks aren't parallel but interleaved.

Standardization Initiatives

Several standardization initiatives for e-commerce exist. Electronic data interchange (EDI) provides a way to conduct structured electronic exchange of information between trading partners. Approved standards for EDI are the ANSI X.12 and UN/Edifact. One of the difficulties of the approved EDI standards is the fact that they're somewhat abstract. This led to a proliferation of custom implementations making the actual deployment expensive. Industry-specific variations emerged. For instance, EDI purchase order transactions differed between the manufacturing, pharmaceutical, and grocery industries. Meanwhile Extensible Markup Language (XML)[1] emerged as the standard for defining the syntax of data structures transferred over the Internet; thus, substantiating the concrete syntax of EDI messages. To provide interoperability across implementations, we must define the concrete syntax and semantics of standardized messages. This is the goal of some standardization initiatives for e-commerce—institutions often reexamine traditional EDI to define the meaning of transferred data (semantics) and employ XML as the practical foundation for structuring this information (syntax). Standardization initiatives for combining EDI with XML include, for instance, the XML/EDI Group (http://www.XMLEDI.org) and e-business XML defined by The United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT, http://www.EbXML.org). Here are some domain-specific standards for the financial domain:

- *FinXML*. This standard (see http://www.FinXML.org) defines XML messages for capital markets.
- *Open Financial Exchange*. This standard (see http://www.OFX.net) defines Standard Generalized Markup Language (SGML) messages for the electronic exchange of financial data. SGML is a predecessor to XML. Essentially, XML is a subset of SGML, whereby various complex features of SGML have been removed. Conversely, Hyper Text Markup Language (HTML) is an application of SGML.
- *Interactive Financial Exchange Forum.* This standard (see http://www.IFXForum.org) defines XML messages for banking services. IFX is a successor to the Open Financial Exchange (OFX).

Reference

1. S. McGrath, *XML by Example: Building E-Commerce Applications*, Prentice Hall, Upper Saddle River, N.J., 1998.

areas of expertise, represented in a single product line and a single domain may span multiple product lines. Domain analysis is the process of identifying, collecting, organizing, and representing the relevant information in a domain, based on the study of existing systems and their development histories, and knowledge captured from domain experts. Figure 3 illustrates the relations between some roles (domain experts, domain analysts, and application engineers) and the artifacts in the DSSA engineering process.

A software system's architecture defines that system in terms of components and interactions or connections among those components. The architecture isn't the detail design, which specifies smaller items (such as program classes). The architecture shows the correspondence between the requirements and the constructed system, thereby providing some rationale for the design decisions. Reference architectures build systems in a product line. As Figure 3 illustrates, the domain model characterizes the problem space, while the reference architecture addresses the solution space.

In application engineering, developers use the domain models within the product line to understand the capabilities offered by the reference architecture and specify a system for development. They then use the reusable components to build the system. Once developers build the architectural model, they can use the model for detailed design and implementation. Application engineers use the domain models to elicit the requirements for the planned software systems with users. The models cover users' needs in terms of existing models. Any needs not covered by a domain model are new requirements. The domain analysts may choose to update a domain model with the new requirements. As the dashed arrows in Figure 4 indicate, various forms of feedback are possible.

Despite the fact that engineering of (new) global applications will usually require integrating existing information sources, I would argue that the integration process should proceed in a top-down fashion, starting with models common to all involved component systems—for example, with domain models in the context of a DSSA engineering process. For such top-down integration of heterogeneous information systems, I recommend using domain-specific standards as the basis for the common data models.[6] Please see the "Standardization Initiatives" sidebar for more resources regarding e-commerce standards.

I illustrate the top-down approach with the small banking example employing the latter of these new standards, namely the Interactive Financial Exchange (IFX), standard which defines the electronic exchange of financial data between financial institutions, business, and consumers via the Internet. Figure 5a displays a small extract of the virtual bank's data model. I based the model on the IFX standard, which uses aggregation as its main structuring mechanism. I used the class IntRateInfo to store both saving and credit interest rates, then distinguish by means of the AcctType

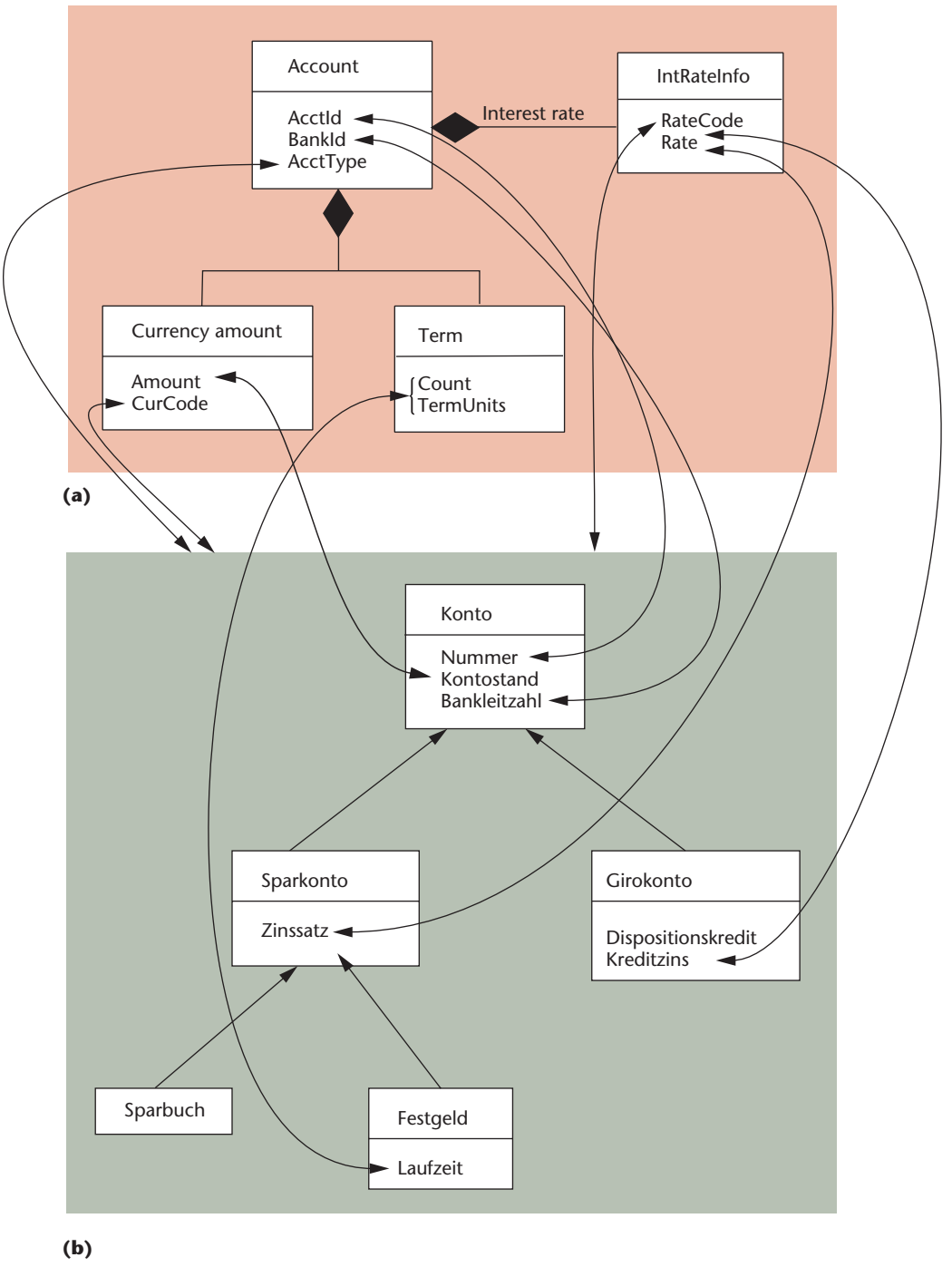

Figure 5. A small extract of the (a) IFX-based data model for the virtual bank together with the mapping to (b) the local German bank's model. In UML, composition is specified by a filled diamond at the aggregating class:[1] Accounts contains Currency Amounts and Terms.

attribute of the associated Account object.

The mapping from the local German bank's model in Figure 5b to the IFX model, as indicated by the arrows, is more complex than the mapping to the previous model of the virtual bank in Figure 2. On the global level, the bank requires the account type (AcctType), the currency code (CurCode), and the rate code (RateCode)—and we must extract them somehow from the German component system. The mapping to the Dutch bank's model (Figure 1c) is similar. An advantage of the bottom-up approach is that the mapping from local components to the common model is straightforward, but at the price of a complex common model. On the other hand, integrating some modern banks with IFX-compliant interfaces into the common IFX model with the top-down approach is also straightforward.

To hook a component information system's model into a common (domain-specific) model

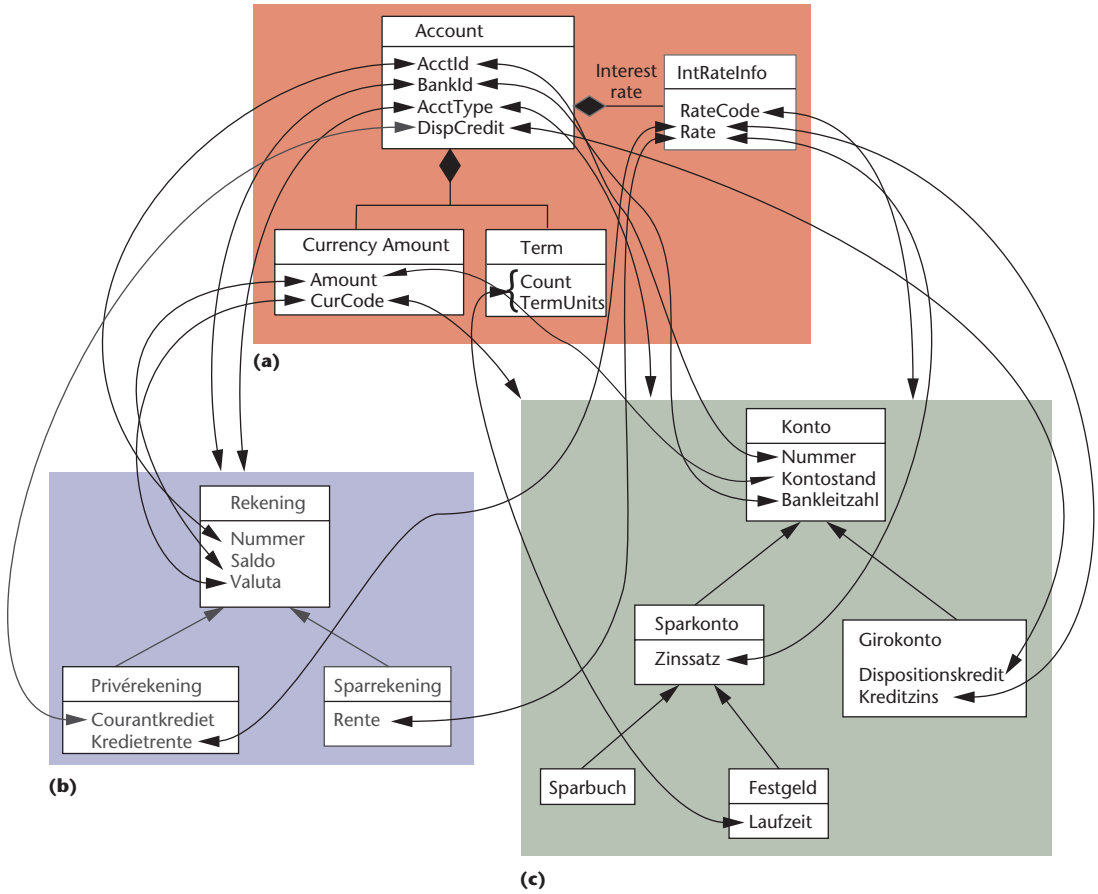

Figure 6. The yo-yo effect after one bottom-up iteration: mappings from (a) the virtual bank's extended data model to (b) the local Dutch bank's model and (c) the local German bank's model. I added the attribute DispCredit to the class Account, based on the requirements of the local Dutch and German models.

with the top-down approach, responsive integration engineers have to understand their component information system and the corresponding domain model, but they don't have to understand other component information systems and constraints that could be introduced by them into the common model using a bottom-up process. The top-down approach defines a common model based on the information we want in it. Then we map to and from the relevant bits of data in the component information systems. This approach will result in a more workable approach for Web-based e-commerce applications with a large number of involved information sources.

A combined yo-yo approach

In practice, we can also expect a yo-yo approach—that is, the integration process alternates with bottom-up and top-down steps. For instance, the bottom-up process may provide input for extending the domain models. It then requires a method for new information that resides in a local information system, but wasn't originally intended in the common model. The requirements of global applications may impose new requirements on the local component systems to achieve a usable integration. The DSSA process takes feedback on the domain artifacts into consideration. Regardless, it's important to start at the top (with the common models).

To illustrate the yo-yo approach, let's return to our banking example after the first top-down iteration: for both local banks, the disposal credit limit on checking accounts is important (Kredietrente in Figure 1a and Kreditzins in Figures 1c and 5b). The IFX standard only considers credit limits on credit cards, not on checking accounts. Thus, we don't consider disposal credit limits in the model for the virtual bank in Figure 5a. In a following bottom-up iteration, the obvious requirement for managing disposal credit limits by the virtual bank implies the corresponding extension of the virtual bank's model, as Figure 6a illustrates. Here, we add the new attribute DispCredit to the class Account. Figure 6 displays the mappings from the virtual bank's extended data model (Figure 6a), to both the local Dutch bank's model (Figure 6b) and the local German bank's model (Figure 6c).

If many applications in this domain require these extensions, it may even lead to eventually

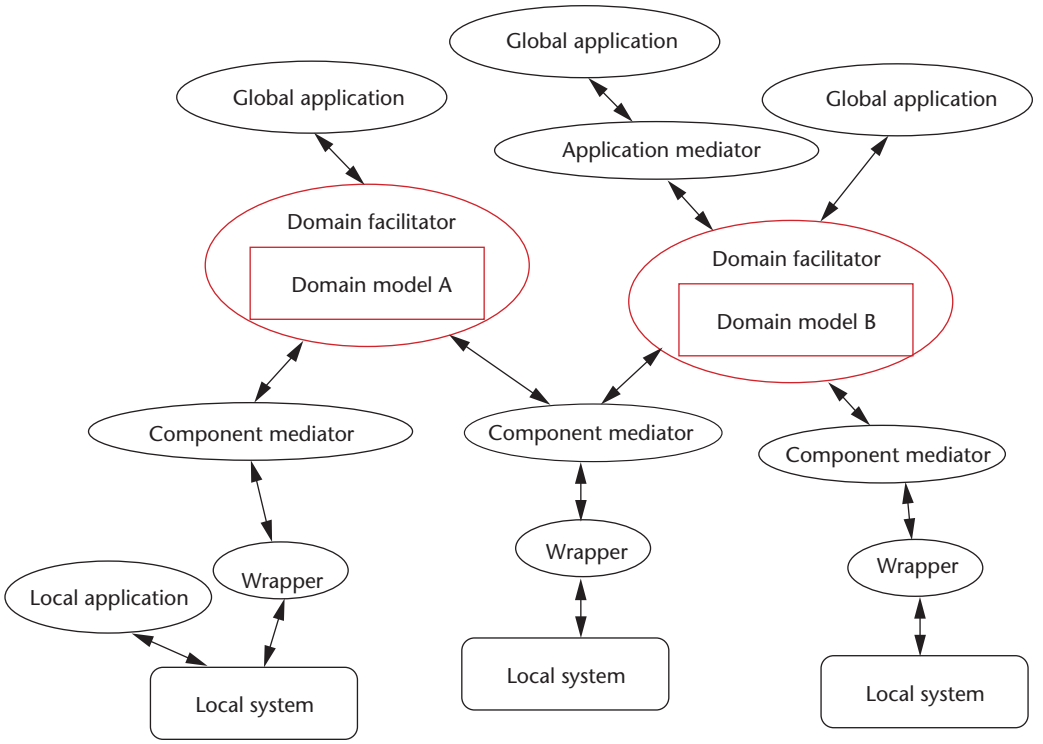

Figure 7. A mediator-based integration architecture for Web engineering providing separation of concerns between domain facilitators, component mediators, application mediators, and wrappers.

extending the IFX standard, not just extending this specific virtual bank's data model. Such bottom-up iterations in a yo-yo process result in feedback from application engineering to domain engineering, as Figure 4 illustrates.

As engineers, we should support updating the common model with a yo-yo approach. Others have used the yo-yo approach with success. Fong et al.,[7] for instance, propose a joint bottom-up and top-down methodology for schema integration in the context of federated database systems. They took the bottom-up approach to integrate existing information systems using traditional schema integration techniques and the top-down approach to develop a data model for the new applications. This methodology aims to facilitate an evolutionary approach to integrate existing information sources supporting new applications. Mutual completeness checks of the common models against the component models determine the model modification steps to be performed on the common models to meet the requirements of the new applications. The idea behind this combined methodology of conducting integration is that top-down and bottom-up approaches can complement each other by taking into account the requirements of new applications. Thus, the integrated model can readily support new applications while continuing to serve existing ones.

Software architecture issues

Researchers have proposed various approaches for integrating heterogeneous information systems—for example, federated database systems or mediator and agent architectures. I discuss the integration process diversity in the context of federated database systems elsewhere.[8] We can use the federated schema architecture to illustrate the differences of diverse integration processes. However, these issues don't only apply to federated database systems.

The yo-yo approach in particular requires a more flexible software architecture as compared to the schema architecture's somewhat static structure in federated database systems. Mediator-based architectures, for instance, offer such flexible and dynamic software architectures.[9] A mediator is a component that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications. Figure 7 illustrates a mediator-based integration architecture. The software components managing the domain-specific models are specific mediators, which we call domain facilitators. A facilitator is a component that provides coordination services.[9] Component mediators accomplish integration into standards-based domain-specific models. This approach increases scalability as it achieves a separation of concerns between the

management of the global models and the integration of component models into the domain-specific models. A wrapper is a component that provides data access into legacy systems to conform with access standards and conventions used by mediators and facilitators. Optional application mediators mediate between domain facilitators and global applications.

Shifting the responsibility for integration from the multidatabase level toward the local level with the top-down and yo-yo approaches should result in more scalable and usable system architectures. Each component mediator must hook a view of its associated local information systems into the common domain models, which the domain facilitators provide. Each local component mediator also maintains the local view on the common domain models. We obtain a decentralized responsibility for maintaining the integration. In the presence of a large amount of heterogeneous information, it's beneficial to reduce the central coordination and distribute the maintenance cost.

The domain facilitator dictates the conditions that the component mediators must satisfy if they wish to participate in some integrated system. Under this approach the domain facilitator can maintain itself in the presence of new mediators joining or dropping out of the overall system. A centralized definition of a collection of common domain models based on standards, far from hindering local autonomy, simply provides a facilitation layer that represents some guarantee to global applications and to component mediators.

Conclusions

The combined yo-yo approach aims to exploit the benefits of both bottom-up and top-down strategies and to serve as a migration path from the traditional bottom-up approach toward a more ideal top-down approach.

Particularly, using domain-specific standards as the basis for the common domain models will simplify the integration of commercial components that offer standards-compliant interfaces, because the structures of local and global data models are identical or at least similar. Using such standards within the integration layer also encourages a (smooth) migration towards modern standards-compliant systems. The common domain models serve as the starting point in the integration process with the top-down and yo-yo approaches. The overall integrated system will be more scalable than with the bottom-up approach, because the common models shouldn't grow linearly with the number of component systems. The common models don't grow linearly with the number of component systems, as it would be the case with the bottom-up approach, since we don't extend the common models with each added component system. Instead, we specify an additional mapping. The decentralized responsibility for maintaining the integration mappings reduces the central coordination needs and distributes the maintenance cost.

Starting with the common models and basing them on standards-based domain models avoids changes to the fundamental structure of these models, making integration more usable and scalable. For integrations with a small number of component systems, the top-down approach may not offer the optimal solutions, but for integrations with many connected systems, as we could imagine on the Internet, we'll obtain a more usable and maintainable overall system architecture.

To ensure correct Web data integration, we need semantic interoperability and the provider and requester of information must have a common understanding of the meaning of the requested services and data. Particularly, we emphasize the role of domain-specific standards for managing semantic heterogeneity among dissimilar information sources. Appropriate standards won't always be available, but when available, it's advisable to use them. In any case, starting with the common representation of information to be integrated for Web-based e-commerce—and not with legacy systems' data structures—is the preferable starting point for integration.

However, correspondences in top-down approaches are usually more complex than in bottom-up strategies because we design the domain model and the component models independently. We specify correspondences separately for each individual component, emphasizing the components' autonomy. In general, top-down approaches result in a more flexible integration than bottom-up approaches. The bottom-up strategy has disadvantages with respect to autonomy and evolution in the component systems; thus, it should be applied only in those scenarios where the configuration of the component systems is somewhat stable (which is unlikely on the Web), or when no model to start top-down exists. **MM**

References

1. G. Goth, "E-Commerce Gets Ready for the Mainstream," *IT Pro*, vol. 1, no. 2, Mar./Apr. 1999, pp. 12-14.

2. M. Schrefl and E.J. Neuhold, "Object Class Definition by Generalization Using Upward Inheritance," *Proc. 4th Int'l Conf. Data Eng.* (ICDE 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 4-13,
3. G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley, Reading, Mass., 1999.
4. I. Schmitt and G. Saake, "Merging Inheritance Hierarchies for Database Integration," *Proc. 3rd Int'l Conf. Cooperative Information Systems* (CoopIS 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 322-331.
5. R.N. Taylor, W.J. Tracz, and L. Coglianese, "Software Development Using Domain-Specific Software Architectures," *ACM SIGSOFT Software Eng. Notes*, vol. 20, no. 5, Dec. 1995, pp. 27-38.
6. W. Hasselbring, "The Role of Standards for Interoperating Information Systems," *Information Technology Standards and Standardization: A Global Perspective*, K. Jakobs, ed., Idea Group Publishing, Hershey, Pa., 2000, pp. 116-130.
7. J. Fong et al., "Methodology of Schema Integration for New Database Applications: A Practitioner's Approach," *J. Database Management*, vol. 10, no. 1, Jan.–Mar. 1999, pp. 3-18.
8. W. Hasselbring, "Top-Down vs. Bottom-Up Engineering of Federated Information Systems," *Proc. Engineering Federated Information Systems* (EFIS 99), Infix-Verlag, Sankt Augustin, Germany, 1999, pp. 131-138.
9. G. Wiederhold, ed., *Intelligent Integration of Information*, Kluwer Academic Publishers, Boston, 1996.

Wilhelm Hasselbring is an associate professor and head of the Software Engineering Group in the Department of Computer Science, University of Oldenburg, Germany. His current research interests include software engineering for cooperative information systems in various application domains. He received his diploma in computer science from the Technical University of Braunschweig and his PhD in computer science from the University of Dortmund. He is a member of ACM and the IEEE Computer Society.

Readers may contact Wilhelm Hasselbring at the Software Engineering Group, FB 10, Univ. of Oldenburg, PO Box 2503, D-26111 Oldenburg, Germany, email hasselbring@informatik.uni-oldenburg.de. Also visit his site at http://se.informatik.uni-oldenburg.de/.