Domain Services for Multichannel Application Software

Martin Lippert, Henning Wolf, Heinz Züllighoven
Computer Science Department, SE group
University of Hamburg
&
APCON Workplace Solutions GmbH
Vogt-Kölln-Straße 30
22527 Hamburg
Germany
{lippert, hewo, zuellighoven}@acm.org

Abstract

Companies have to adapt to changing environments and business requirements in short cycles. They seek to address their customers via various channels involving new (net-based) technologies. This poses an interesting question: How can we support multiple channels with various user front ends without at the same time duplicating business functionality?

We present an elegant solution in the form of an architectural approach based on so-called domain services. Domain services are related to core business functions or concepts and they abstract from any type of UI (user interface) or interaction style. Domain services also are independent of database systems, concrete work flows and technical prerequisites like host systems.

When designing domain services, we focus on the tasks of an application domain because they are the most stable elements in enterprises. The core tasks of an enterprise remain the same as long as an organization stays in its business domain. However, the concrete work flows at the different workplaces and the (technical) front ends of the application systems will change. Consequently, it is important to identify the underlying tasks and model them as "faceless" services. Thus, one domain service can be combined with various channels, their user front ends and interaction styles may differ. Such a service may even be used by other software applications of business partners within the net of a virtual company.

This paper discusses business-oriented as well as technical implications and solutions. We present our architectural design and concrete experiences gained from professional software projects.

Keywords

Multichanneling, Domain Services, Client-Server Architecture, Task Orientation

1. Introduction

Enterprises have to adapt to changing environments and business requirements in short cycles. They seek to address their customers via various channels. New net-based technologies evolve in short cycles (often several times a year) and companies would like to use these new technologies to enhance their customer service. Various examples for such new channels have emerged over the past few years: growing Internet availability, mobile devices and, last but not least, the wireless application protocol (WAP) or mobile phones, to name the most prominent. Today, banks provide Internet banking or WAP-based stock-market information.

From what we see and read in the media, it would appear that many companies have the know-how to handle these new technologies. But, behind the scenes, they still have to invest a great deal of time and effort, not only to understand new technologies, but also to implement the new channel containing the required business functionality.

As consultants to companies in the financing sector, we have had ample opportunity to observe the reality behind the glossy ads of e-commerce and new technologies. Some of the companies simply transfer the data coming in through new channels to their transaction host systems by hand. Others design and implement entire new systems for their e-commerce applications. At best, they consolidate the data of these e-commerce systems with their host databases. In any case, they have to maintain two different implementations of the same business functionality. This duplicated functionality sometimes leads to slightly different results, e.g. when calculating the premiums on life-insurance-policies.

The interesting question here is: How can we design software to support multiple channels with various user interfaces, without at the same time duplicating business functionality?

2. The Container Terminal business case

We use below an example business case to demonstrate our point. The example is based on an application for a company working with overseas containers at a large German port. The company employees register full or empty containers taken to the port by ship, train or truck. They manage these containers until they are shipped or driven to their next destination.

The company is developing a completely new software application for a leading-edge container terminal. They wish to realize a flexible architecture for multiple channels.

For example: A shipping company announces via the Internet a container transportation to the container-terminal company. The container-terminal company uses this advance notification to organize their container warehousing area so as to ensure a high throughput as well as to initiate other organizational activities. An agent of the terminal company prepares documents for customs clearance using a PC-based desktop application.

When the truck driver arrives, he enters the terminal area with a set of containers, registering these containers with the container-terminal company via a computer self-service terminal. This container registration is not only used by the truck-driver. Terminal employees also can book a container using the system. This is done when normal registration by the truck-driver is unsuccessful or if certain unusual business problems occur.

On registration, the terminal company collects several additional papers and forms (additional customs documents, shipping documents, etc.) relating to the containers.

From the moment of advance notification to the actual shipping of a container, the shipping company should be able to monitor the state and location of the container.

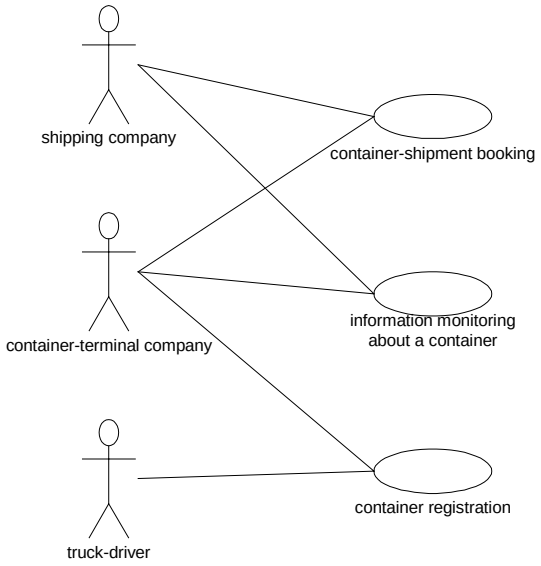

Figure 1: Use case for the container-terminal example

This short business use case can be sketched as in figure 1:

1. The shipping company makes a container-shipment booking via a website or via their own software system which is connected to the container-terminal system (b2b).
2. The container-terminal company receives the booking information and is able to plan its container storage area and other workflows in accordance with the bookings. This task is done flexibly by the terminal company employees on their desktop machines.
3. A trucker delivers containers to the terminal and/or picks up containers. He registers these operations using a self-service computer terminal that provides a specialized tool for this purpose.
4. The terminal company collects further information for the container shipment, such as documents for customs clearance or shipping documents. They perform this collection via a desktop system or with handheld devices.
5. The shipping company requests information on the current state of the container shipment. They display the process-tracking information via the Web on a special Internet site.

Obviously, several channels are needed to manage the process and access information about it. For example, the shipping company uses the Internet and a web browser to book a container shipment and gather information on the current state of the shipment process at the container terminal. The trucker uses a self-service terminal specially designed for this purpose. The terminal-company agents use normal PCs to deal with the shipment documents and hand-held devices to register containers in the truck yard.

Different companies are involved in this business use case. First, there is the container-terminal company. The shipping company cooperates with the container-terminal company and uses their services. The trucker may be working for a transport company that is a sub-contractor to the shipping company. This business use case is thus characterized by a business-to-business relationship. We assume that this b2b connection is supported by software, not only at the level of transferring data via web browsers, but also by sharing common application logic.

This small example should help to illustrate various aspects of building advanced multichannel application software in a distributed environment.

3. Challenges for the Software Architecture

Let us first consider the problem of different channels within a more traditional software architecture. The classical two-tier architecture (see [12]) divides the software system into the database server and clients

working on top the database. As a result of this, the client application contains business logic as well as the complete interaction. There is no separation of functionality and interaction. If programmers need to add a new and different channel to this kind of software system, they have to implement the business functionality twice, the chunk of software per channel containing the complete application: interaction as well as functionality.

The more recent three-tier architecture (see [3]) goes one step further. By dividing the application into two tiers, most of the three-tier systems separate the application logic from the user interface. This does not mean that they separate the interaction from the functionality. In most cases, the application-logic part of the software on the server side is specially designed for a specific type of user interface. The reason for this is that the user-interface front end simply displays data and passes UI events to the application server.

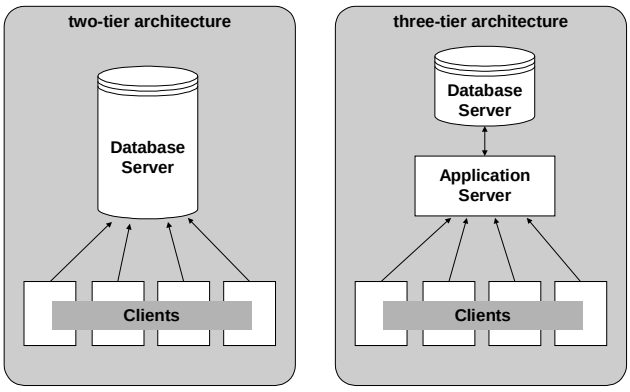

Figure 2: Two-tier and Three-tier Architectures of Distributed Software Systems

In both of these cases, the programmer has to (more or less) duplicate application logic to support new or different channels of various types. Even with the three-tier architecture, the programmer is unable to reuse one application-server part of the system to support various channels because the channels may vary in the way they interact with the user. The type of the user interface may be totally different. Imagine a desktop application versus an e-mail gateway. Even if you wished to provide similar services to the users (for example, the container-shipment booking) you would have to deal with two different interaction models. An e-mail system cannot be handled like a desktop application. This makes it difficult to add new channels to an existing software system and to respond to changing environments flexibly.

Another difficulty is the issue of consistency within a business process. In most cases where the functionality is not separated from the interaction, consistency checks are only possible on top of the business data. This is the case because several different applications and tools are used to take care of one task. The consistency checks cannot be done within one application because the application deals with one part of the task and not the whole business process. And one application is not able to take care of a task done by another application. The consequence is the implementation of expensive plausibility checks on top of the database. This means that a part of the business logic is implemented once again, this time into the plausibility checks.

Summarizing these architectures, we see a duplication of business-functionality implementations on the one hand, and a simple data-centered view of the system on the other. It is hard and complicated to add new channels to this kind of software. If we take a look at the example, we would have to implement many different components to enable each of the different user interfaces. And we would have to implement the same part of the business functionality in each of these components.

In order to minimize the duplication of business functionality and business knowledge, we would have to move one step closer to a flexible architecture. This would make it easier for the developer to add new user channels to the system and respond to new and changing technologies.

4. Our Multichanneling Approach: Domain Services

The first step toward solving the problem outlined above is to separate business logic from interaction. This is not a new idea. The classical model-view-controller paradigm (see [8]) was designed to solve precisely this problem. The main difference is that model-view-controller separates interaction from functionality within one application or process. It is useful for building interactive tools.

A recent, different approach is the three-tier architecture. Here, the client front end only implements the UI of an application. The second tier usually contains the business logic plus the process control, while the third tier encapsulates the database. The three-tier architecture is a step toward a more distributed solution to the multichannel problem. It still has some severe shortcomings though:

- The UI tier is designed as a thin client with no specific functionality apart from displaying data and gathering input.
- The second tier contains both the business logic and the process control. It thus mixes functionality with interaction.
- The UI tier actually displays the data provided by the third tier and controlled by the second tier. Changing user requirements normally affect all three tiers: new functionality (second tier) will frequently access additional data (third tier) which then has to be displayed (first tier).

Our approach of allowing flexible multichanneling is in line with model-view-controller because it separates functionality from interaction. Based on the idea of different layers from the three-tier architectures, it separates business logic more clearly from different interaction styles and from user front ends.

The first design principle is to encapsulate pure domain-specific functionality into a component without any assumptions of concrete user-interface type or specific workplace. They are implemented in a manner that is neutral with respect to the specific runtime architecture. We call these components *Domain Services*. Allen and Frost have a similar approach. They call this kind of service component a business service (see [1]).

A Domain Service provides the functionality needed to support a small coherent set of users tasks, not the whole application system.

Let us take a look at the business-case example. We can identify a Domain Service covering the different aspects of booking a container. This service would apply in different situations of our business use case. The shipping company could give its advance notification of containers via the Internet; the truck-driver could register

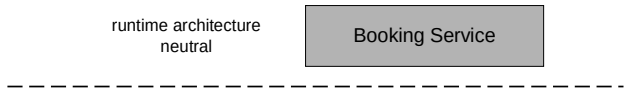

Figure 3: A Single Domain Service for Container Booking

a container using this service as well as the terminal company employee. This service need not be changed when the terminal company decides to offer a call-center service to announce containers and allow registering of containers via mobile phone. Once we have identified the Domain Service for booking containers, we can design and implement this service component (see figure 3).

We can also identify a Domain Service for container process tracking for the shipping companies. This service collects information on the current status of a container shipment. Again, this information is to be presented via various channels (Internet, WAP, ...) to shipping companies, call-center agents or truckers. It is important that this service be designed not only as a standalone application with a complete user interface. Especially for desktop applications like that of a terminal company agent, both the tracking and the booking service should be accessible via flexible tools. The user should not have to change the application or start an isolated web browser plug-in. This would lead to various different and specialized systems, each responsible for one specific task and

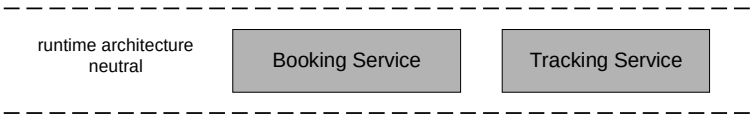

Figure 4: Different Domain Services can exist side by side

with no means of working on a common set of documents or other materials (see figure 4).

With the same idea in mind we could identify more Domain Services for our example. But these two Domain Services should suffice to illustrate the architectural concept underlying this paper.

We now address the issue of linking a Domain Service to the user interface. Adding a graphical UI to a Domain Service at the client is not enough. Depending on the type of channel used, the different GUIs have different interactive means and different "interaction styles".

For example, a Java Servlet always transfers an entire browser page, whereas a normal desktop application requests only those services and data needed to respond to a specific user action. A WAP application has the same interaction style as the Servlet, but it has to take care of the very limited display and interaction options.

This leads to so-called *Interaction Services* between Domain Services and user front ends. These take care of the different interaction styles of the various channels linked to one service. There may even be front ends that do not need adapting. For example, a rich client desktop tool could use the Domain Service directly to provide interactive access to the functionality of the Domain Service (see figure 5).

The top layer in the figure shows the front ends of the different channels, like a web browser, a desktop application or whatever channel is used to support the users in performing their tasks.

The second level contains the different interaction services for adapting the front ends and the interaction styles of the channels to the Domain Service used.

A Domain Service itself can work on top of different

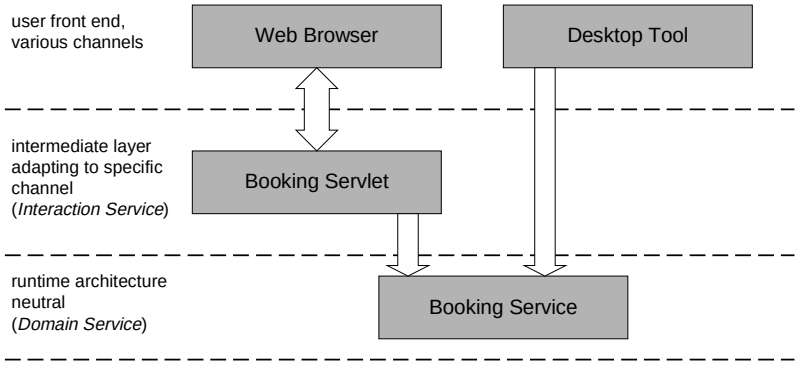

Figure 5: Domain Services can be used for different front ends via Interaction Services or directly

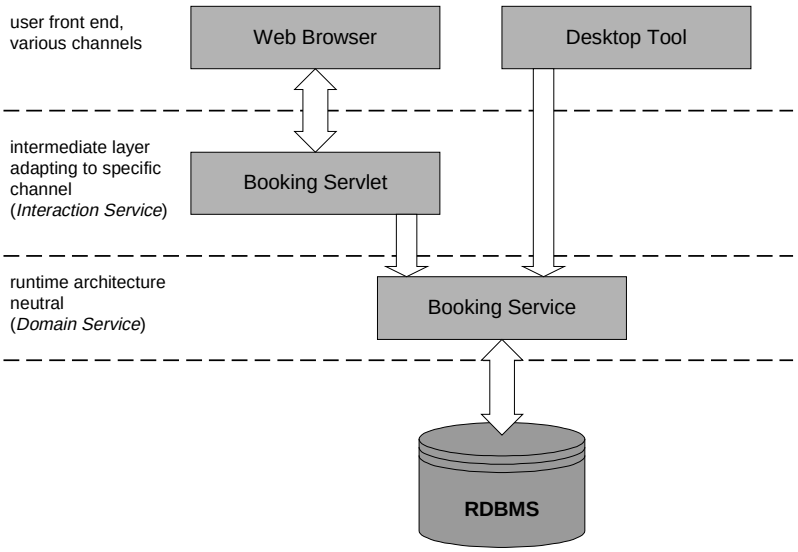

Figure 6: A Domain Service can be realized on top of a database

basic systems. It could, for example, be connected to a database system or a host-transaction system or an ERP system like SAP R/3. Since a Domain Service encapsulates its implementation and only offers business logic, the concrete type of back end has no influence on the Domain Service's interface (see figure 6).

Of course, if the underlying technology of the basic system changes – for example, from a relational database system to an object-oriented database system – the Domain Service has to be changed, too. But this change only affects the implementation of the Domain Service, not the applications running on top of it and not even the Interaction Services. Incidentally, this change in a basic system could be anticipated by a persistency layer between Domain Service and basic system, but this is beyond the scope of the present paper.

We have now outlined the basic architecture of a Domain Service and shown how it enables multi-channeling. This architecture is not confined to an in-house system. It opens the doors to b2b applications. The separation of front end and services is a natural process border. We can thus use a Domain Service not only within one company to allow multichanneling for addressing different users, but also let other companies access the Domain Service directly. The shipping company in our example has, of course, it's own computer system to manage its container transportations. Via a WAN, they can now integrate the booking service for container shipping from the container terminal company. With a Domain Service plus interaction services, there are different ways of realizing this integration. The shipping company could link up to an existing interaction service, thus running the same type of application as the container-terminal company. Or they might decide to access the booking service directly, combining it with their own interaction service to form a new type of in-house application (see figure 7). Since the Domain Service is neutral with respect to the runtime architecture, both design options are possible.

Certainly, a lot of security issues have to be kept in mind when offering your services to other companies. We do not deal with authentication or other security issues in this paper. Please refer to the relevant literature on security.

5. How to Design Domain Services

A Domain Service may be designed in different ways. The choice depends on the domain specific tasks that are to be supported by the Domain Service. First of all, most Domain Services are implemented for distributed systems. They therefore provide an interface that could be used remotely. And a number of client applications and/or

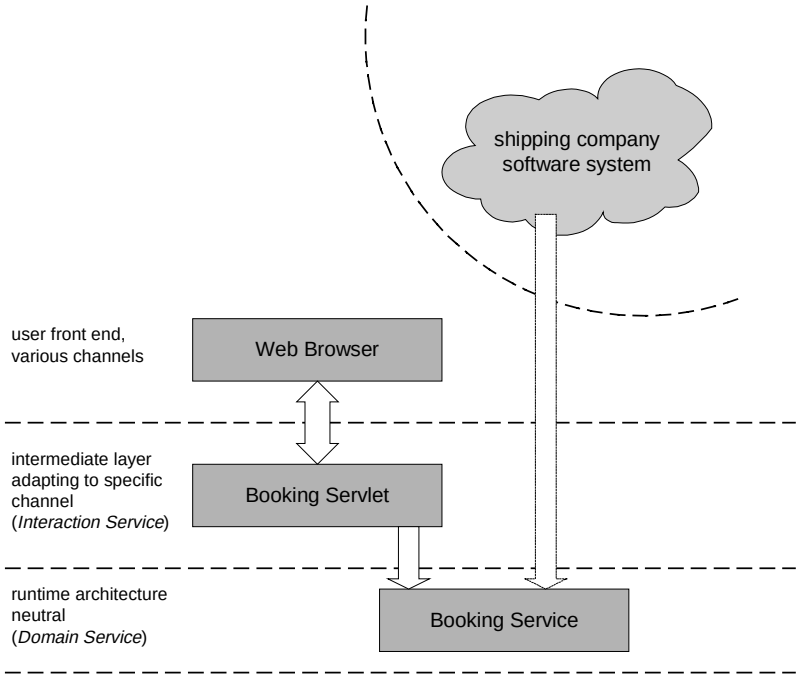

Figure 7: Domain Service used by the software system of another company. Elegant solution for b2b relationships and cooperation between different software systems

interaction services use one instance of the Domain Service (there may exist a number of instances of a specific Domain Service to allow scalability, but that is irrelevant to our discussion here). This raises the question of user sessions and whether or not we need them in Domain Services.

Implementing session handling is by no means an easy matter. The developer of a Domain Service supporting user sessions must invest a great deal of extra time to implement session handling. These kinds of Domain Services are stateful, which goes ahead with a more complicated implementation and a more complicated usage model of the Service.

This is the reason why we try to design the Domain Services to be stateless. In this context, stateless means that the Service does not provide a session concept or any other session-based features. Materials (in this context: domain-specific data records) that are created, modified or deleted by the Domain Service in response to client requests are stateful. For example, the Service might take a registration form out of the database, modify it to suit user needs and put the modified form back into the database. In this case, the registration form has a state, and the Service changes the state of the registration form before putting it back into the database. After this atomic operation, the Service itself does not contain any information about the changed form or the user that changed the material.

This stateless design makes it extremely easy to implement Domain Services, and we prefer to keep things as easy as possible. Still, the decision about stateless vs. stateful depends on the application system supported and the users' tasks. There are a number of situations in which the application system or the interaction service need a session concept to provide appropriate user interaction. A login dialogue in a web-based front end is a simple but good example. In this situation, the software architect has two ways of realizing the sessions in cooperation with the interaction services and the Domain Services:

- The first is: the Interaction Service deals with the session handling, and the Domain Service is stateless (see figure 8). Using the web login example, the servlet deals with the user session and keeps the user information in session objects. In this case, the Interaction Service, implemented as a servlet in this example, has to put the user information into every method invocation of the Domain Service. This type of session handling inflates the interface of the Domain Service because every operation (in our login example) must have a user parameter, for the Domain Service has to know the user, but it has no session concept. This design concept is suitable for Domain Services with a small number of session information like the user identification. If additional session-based information is needed, the interface of the Domain Service would be overloaded with the additional parameters.
- The second way is to integrate a session concept into the Domain Service itself. In this case, the Domain Service requires a lot more implementation to handle the sessions, but on the other hand, the parameter lists of the application-oriented methods of the Service are short and clear. We prefer this design in situations where the session-based information is more than simple user information, etc.

In addition to session handling, we should take a look at the interface of a Domain Service. The Domain Service is responsible for consistency checks and changed materials or business data. This is our desired design. The

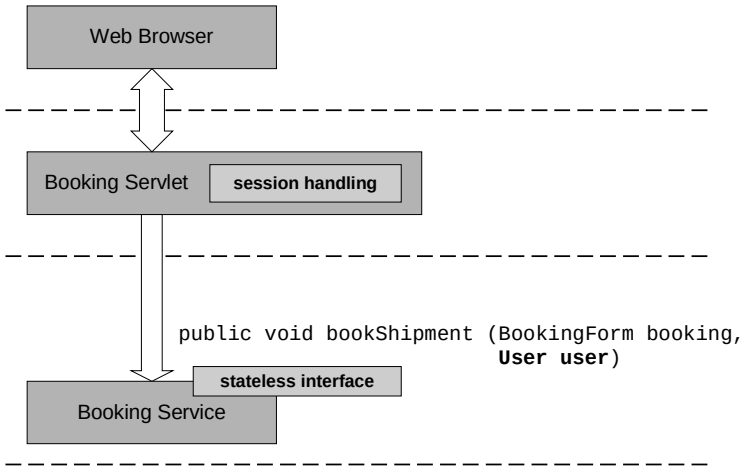

Figure 8: Session handling inside the Interaction Service dealing with a stateless Domain Service

Domain Service updates business data into the database or another persistency medium. This means the application components built on top of the Domain Service deal either with copies of the business objects provided by the Domain Service, or the Domain Service itself fails to provide business objects at the interface to the application components. Otherwise, the application built on top of the Domain Service (the client in this case) could change a business object without giving the business object back to the Domain Service responsible for persistency and consistency.

Two possible solutions should be mentioned here: a copy-based interface and a value-based interface. A Domain Service offering a copy-based interface provides only copies of business objects to the client. If the client sends a changed copy of a business object back to the Domain Service, the service is responsible for replacing the old business object with the new changed one.

A value-based interface of a Domain Service does not offer complete business objects but single business values.

This kind of interface benefits from the advantages of value semantics. The Domain Service can be sure that no client will change a business object directly, changes only being possible through the Domain Service. Implementing such a value-based Domain Service is quite easy using a framework supporting the implementation of domain-specific values. The JWAM framework, for example – an application framework for large-scale object-oriented software architectures – provides support for domain values (see [7]).

6. Implementation Variants

Domain Services are often designed to be server-side components. This means that they are provided by server machines and are potentially used by a lot of clients.

In our case, we use Java to implement the Domain Service, but the architecture of Domain Services may be realized using any other language.

For distribution of the service component, different techniques can be used. To begin with we used Java RMI (Remote Method Invocation, see [5]) to distribute the service component. This was during the phase in which we were experimenting with the architecture and the interfaces of the services. In this case, a remote interface for the service component is defined and this remote interface is used on the client side to access the Domain Service residing on a server (see figure 9).

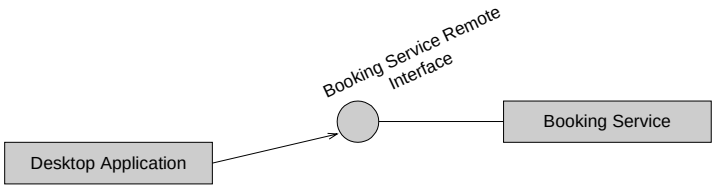

Figure 9: Booking service with remote interface for client applications

If the Interaction Service is running on the server side – for example, inside a web server as a servlet – the Interaction Service is able to use the Domain Service directly without any remote interfaces (see figure 10).

This is possible if the web server or the servlet engine runs within the same process as the Domain Service. If not, the servlet has to use the Domain Service via the remote interface, too.

We then implemented the services as Enterprise JavaBeans (see [9]) session beans, which is also possible. As is evident, almost any distribution mechanism can be used to implement the Domain Services. The architecture of Domain Services does not include a specific implementation technique to be used for the service components. This is yet another feature offering flexibility.

Combinations of implementations for Domain Services are also conceivable. You might want to implement the service independently of a specific distribution technique. Then, you would have to implement special distribution wrapper components that distribute the service to clients via RMI, EJB, Jini (see [4]), etc.

Even if you wish to support mobile devices that work off-line, like notebooks or palmtops, you can use the same service implementation at the office and on the mobile devices, doing the data synchronization later on when the mobile devices are online. In this case, there would be a special implementation of the Domain Service used supporting an off-line work mode.

7. Using Domain Services for Legacy Software Encapsulation

An interesting application for Domain Services is the smooth encapsulation of legacy systems. We used this technique in different projects to ensure a smooth migration from a legacy host-based system to an object-oriented application system. The idea here is to choose a small task to be supported by the new object-oriented application. A Domain Service is then designed for this specific task. The object-oriented system uses this Domain Service to provide functionality to the users. The important thing is: the Domain Service can be used independently of its implementation.

The Domain Service first realizes the service using the legacy system. This enables the tools to be realized using the object- and task-oriented Domain Service interface where the Domain Service uses the legacy system to realize the functionality, providing materials filled with data from the legacy system.

Using this approach, more and more Domain Services can be added encapsulating more and more legacy system parts. And, after a period of time, the Domain Services

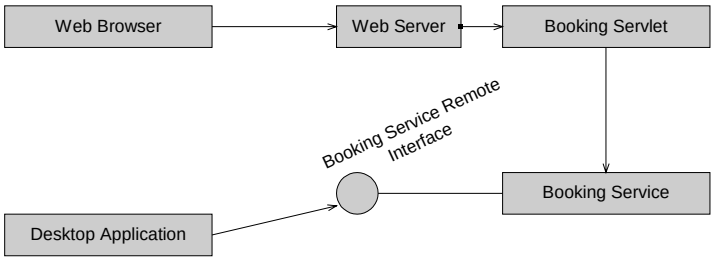

Figure 10: Booking service used by booking servlet on the server side as well as through a remote interface

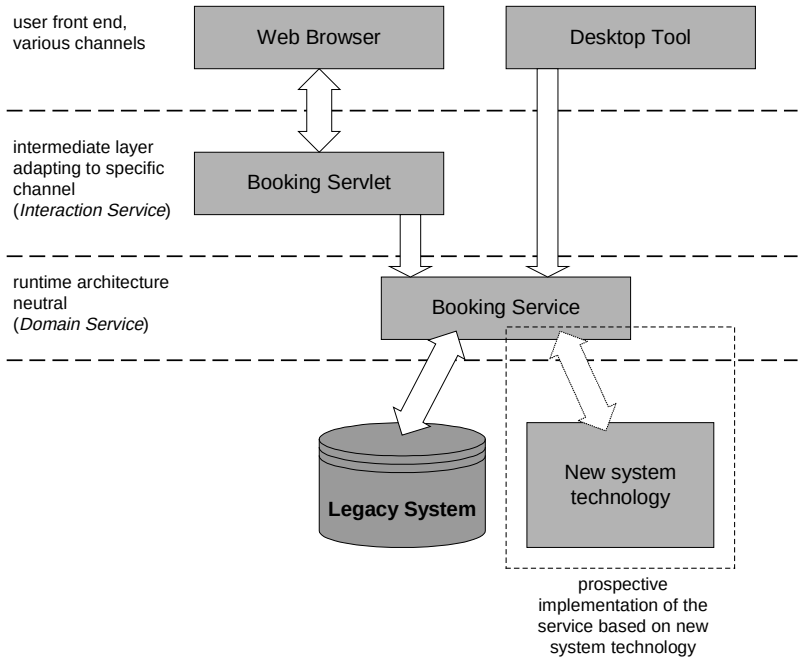

Figure 11: Domain Service encapsulates the legacy system. This implementation detail may change in the future to switch to new technology replacing the legacy system.

can be re-implemented using other, newer technologies. The tools implemented on top of the Domain Service are not affected (see figure 11).

8. Related Work

The approach discussed in this paper can be related to work in other areas. First of all, there appears to be some confusion over the term "service". We outline the similarities and differences to a number of related discussions in software technology.

Workflow business components are a recent topic in the area of connecting business objects to front-ends (see [11]). They are frequently implemented on top of database or legacy systems. The main difference between workflow components and Domain Services is the way they support users in dealing with their tasks. A workflow controls normally the sequence of activities. Domain Services offer a coherent set of domain-related operations that can be combined in many ways to deal with changing situations. Services do not specify a fixed sequence of actions; they are based on a number of related tasks within a domain. The Domain Service normally comprises both a collection of business objects – or, as we would say, materials – and operations on these business objects. The "services" are realized separately from the business objects as part of the Domain Service that encapsulates the business objects.

A very similar approach to ours is the Proposal Pattern architecture (see[2]). We share several of their principal ideas. A proposal encapsulates a task within an object, called a proposal object. The proposal object resides between the user front-end technology and the back-end system, as with Domain Services. It connects the front end to the back end in a mediator-like role, acting in a manner similar to the command pattern (see [6]). As in a Domain Service, a proposal encapsulates transactions on domain objects. One main difference is that a Domain Service is not merely a container for proposal data and does not act as a transient encapsulation of exactly one request. A Domain Service is (mostly) a long-lived single instance or component handling numerous requests from various front ends. It is an abstraction of the concrete runtime-architecture.

Since we are dealing with a kind of service architecture, we must consider the OMG standard for CORBA services (see [10]). It provides a set of system-level services built to generally support the development of CORBA systems. We focus on the design and implementation of domain-specific services. A Domain Service realizes domain-specific functionality, which is combined with other Domain Services to meet the current needs of the user tasks in the domain.

9. Conclusion

We have analyzed the multichanneling approach and identified the difficulties encountered when supporting multichanneling with common software architectures. To solve these problems and obtain highly flexible software systems, we introduced Domain Services. They enable the interaction to be separated from the functionality and concentrate functionality in a user-task-oriented manner. A Domain Service can either be used directly from a front-end implementation or adapted for various front-end technologies via Interaction Services.

Ongoing projects show, that these kinds of services provide a more flexible way of implementing business functionality independently of the user interface. Another application we implemented has shown, that we can easily integrate new channels into an existing multichanneling Domain-Service-based architecture without duplicating business functionality. A helpdesk system built on top of the JWAM framework (see [7]) and designed to support framework development (as well as other systems) is based on the concept of Domain Services and supports desktop applications, thin-client applications with

application servers, HTML websites via Java Servlets, WAP via Servlets, SMS messaging and an e-mail connection. At least half of the channels listed were not under consideration when we started implementing the helpdesk Domain Service. However, we had no problems integrating the new channels.

10. Acknowledgements

We would like to thank Michael Otto and Norbert Schuler for their significant contribution to the design and implementation of Domain Services and to identify the underlying concepts. We also wish to thank the JWAM framework architecture group for their support. Their work formed the basis for the Domain Services and this paper.

11. References

1. P. Allen, S. Frost: *Component-Based Development for Enterprise Systems. Applying The SELECT Perspective*. Cambridge University Press, Cambridge, 1998.
2. B. Bernstein, J. Tibbetts: Intermediary Objects for Transactional Systems: Introducing "Proposals", in *Object-Magazine*, November 1997.
3. S. Cook, J. Daniels: *Designing Object Systems. Object-Oriented modeling with Synthropy*. New York, London: Prentice-Hall, 1994.
4. K. W. Edwards: *Core Jini*, Prentice Hall, July 1999.
5. J. Farley: *Java Distributed Computing*, O'Reilly & Associates, January 1998.
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
7. The JWAM framework web site at: http://www.jwam.org/.
8. G. E. Krasner, S. T. Pope: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, August/September 1988.
9. V. Matena, M. Hapner, B. Stearns: *Applying The Enterprise Javabeans Architecture: Programmer's Guide and Specification*, Addison-Wesley, June 2000.
10. R. Orfali, D. Harkey, J. Edwards: *Client/Server Survival Guide, Third Edition*, John Wiley & Sons, January 1999.
11. M.-T. Schmidt: Building Workflow Business Objects, OOPSLA'98 Business Object Workshop IV, ACM press, 1998.
12. H. Singh: *Processing to distributed multiprocessing*. Prentice Hall, Upper Slade River, 1999.