Framework Development for Large Systems

Dirk Bäumer[1], Guido Gryczan[2], Rolf Knoll[3], Carola Lilienthal[2], Dirk Riehle[4], and Heinz Züllighoven[2]

Abstract

Frameworks are a key asset in large-scale object-oriented software development. They promise increased productivity, shorter development times, and higher quality of applications. To fulfill this, frameworks should be designed in such a way that they can evolve, be easily reused, adapted and configured. Drawing on experience with large-scale industrial banking projects, we present concepts and techniques for domain partitioning, framework layering, and framework construction. In particular, we discuss how domain aspects relate to framework structure, how frameworks are layered to accommodate domain needs, and how the resulting framework layers are integrated without tight coupling.

1 Introduction

These days, many businesses such as hospitals, banks and insurance companies tailor their services to the individual customer. Customer needs become the center of attention, the service being adapted to meet their needs. The more specific the service required, the more specialized the software solutions have to be. To achieve such flexibility, computer support is indispensable.

For some years, we have been designing and implementing such software solutions for various application domains. Using object-oriented technology, frameworks now largely support the development of new applications. However, frameworks alone don't solve all the problems. The construction and use of frameworks is so highly complex that software developers are confronted with almost insurmountable difficulties (see [4]). In this paper, we describe the problems encountered, and the concepts and techniques used to overcome them.

The work reported here is based on a series of object-oriented software projects that were conducted at the RWG[5] (cf. [2, 3]), a company providing software and computing services to a heterogeneous group of approx. 450 banks in Southern Germany. The projects have produced a family of applications covering almost the whole area of banking - tellers, loans, stocks and investment departments as well as self-service facilities. The entire Gebos[6] system, including several frameworks, consists of 2500 C++-classes and was developed over the past five years. Against this background, conclusions can be drawn about how to further develop, reuse and adapt frameworks. The presented solution could be transferred to any kind of graphic workplace system embedded in the context of human work.

First, we look at the main problems encountered, when designing large software systems. Such a large software system addresses and correlates the various tasks found in a business. It should be possible to configure and adapt the system to the requirements of individual workplaces in several different enterprises. This can only be achieved by employing framework technology. The framework layers of the Gebos system are therefore described and discussed in detail. The following section describes layering techniques and divides frameworks into a concept and an implementation part. These parts are combined to form concept and implementation libraries. Finally, we examine the Role Object design pattern, which is used to make an object play different roles in different departments while remaining an integrated component.

2 Framework Layering in Large Systems

A framework models a specific relevant domain aspect using classes and objects. Abstract classes define the model and the interaction of their instances. Concrete classes provide default behavior and implementations of the abstract classes. The abstract classes specify the flow of execution and can be tailored to specialized implementations by subclassing.

[1] Dirk Bäumer (baeumer@takefive.ch) can be reached at TakeFive Software, Eidmattstrasse 51, 8032 Zurich, Switzerland.
[2] Guido Gryczan, Carola Lilienthal, and Heinz Züllighoven (first.last@informatik.uni-hamburg.de) work at University of Hamburg, Germany.
[3] Rolf Knoll (rolf_knoll@rwg.e-mail.com) can be reached at RWG GmbH, Räpplenstrasse 17, 70191 Stuttgart, Germany.
[4] Dirk Riehle (Dirk.Riehle@ubs.com or riehle@acm.org) can be reached at Ubilab, UBS, Bahnhofstrasse 45, 8021 Zurich, Switzerland.
[5] RWG stands for "Württemberg Co-operative Bank Computer Center".
[6] Gebos stands for "Banking Co-operative Office, Communication and Organization System".

Appeared in *Communications of the ACM* 40, 10 (October 1997). Page 52-59.

We describe an object-oriented software architecture using framework layers, the integration of different frameworks and their customization making up the resulting system. Frameworks and framework layering must be rooted in the application domain in order to meet business needs. We therefore begin by discussing how to relate application domains to frameworks.

2.1 Application Domain Concepts

The services offered by a business enterprise such as a bank can be divided up into different areas of responsibility. Traditionally, banks have organized their departments according to these different areas. Each department consists of specialists, whose work is confined to their own field of expertise. We call these areas **business sections** (e.g., teller, loan, investment, see Fig. 1). Today, the division into different departments is often supplemented by so-called service centers offering customers an “all-in-one” service. Service center clerks can perform most of the common tasks encountered in the various business sections. Business reorganization, thus, creates new types of workplaces. A concrete form of work will be referred to as the **workplace context** (see Fig. 1). Examples of workplace contexts in the banking business are:

- Customer service center: A customer receives advice on loans, bonds or investments.
- Teller service: Customers should receive a fast and qualified service for frequent and standard requests. Here, a clerk has to deal with deposits, withdrawals, transfers and foreign currency.
- Automatic Teller Machines and home banking: The services of the bank are made directly available to its customers.

For every workplace context, a different **application system** may potentially be required. For a workplace in a customer service center, support is needed for the mixture of services from the different business sections. The customer consultant needs access to both the loan and investment sections when advising a customer. At the same time, the system must allow a simple money transfer.

Although each application system is tailored to the needs of a particular workplace context, all should be built on the same basis. Take the following example: Customer profiles exist in all departments of a bank. In the loans department, sureties are an essential part of the customer profile. A surety serves to minimize the bank’s losses in the case of a customer’s inability to pay. In the investments department, savings accounts form part of the customer profile. In both departments, though, the customer’s name, address and date of birth are an integral part of the profile. Software development for different workplace contexts needs to take into account these differences and similarities.

Our approach is to identify the core or common parts of the concepts and terms that are essential to running the business as a whole. We call these common parts the **business domain** (see Fig. 1). Cooperation within an organization is only possible through the existence of these core elements in the business domain. In a bank, ‘account’, ‘customer’ and ‘interest rate’ are examples of overlapping business domain concepts.

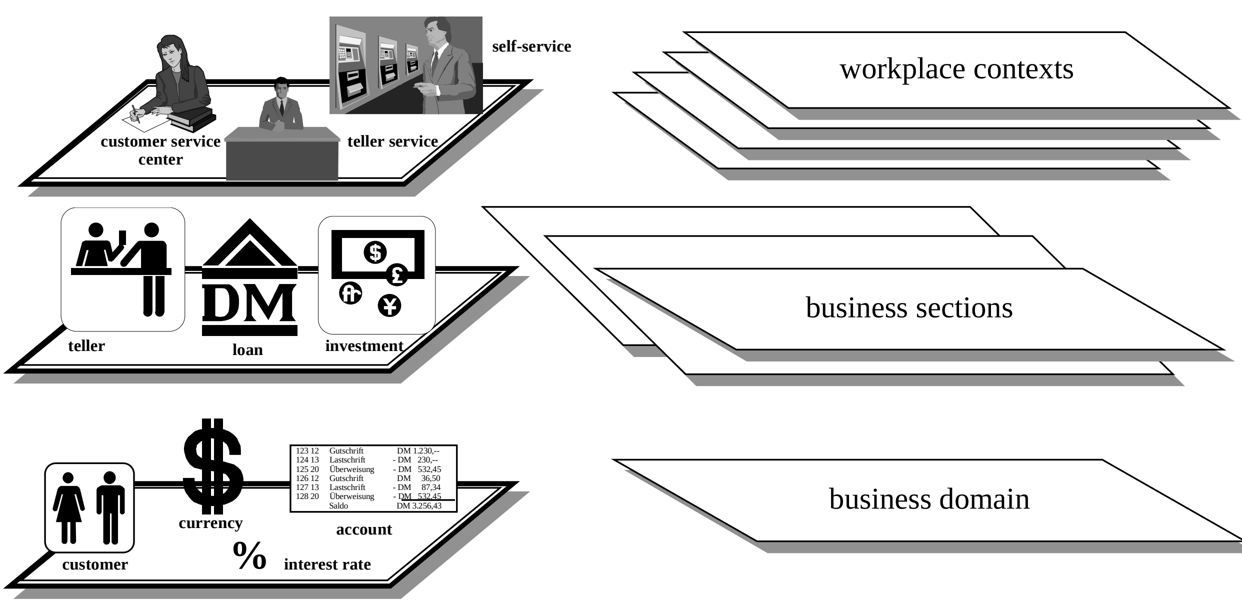

Figure 1: Organizational concepts in the application domain

Modeling of the business domain can only be done if at least two business sections are already supported by software systems. Although the business domain forms the basis for the business enterprise as a whole, it is not tangible as such: there is no place in a bank where 'account' or ' customer' can actually be seen. Considering the relevance of the business domain, we need to "reconstruct" the core concepts behind the different business sections in order to build an integrated system.

We believe that frameworks for this type of applications should be organized along business-domain, business-section and workplace-context lines. In order to avoid unnecessary duplication, frameworks should be designed so as to encourage or even enforce reuse of the business domain in the various sections. The reuse of framework components then yields the basis for uniform and coherent application systems. In the following sections, we describe the close correlation between application domain concepts and the framework architecture.

2.2 Layers of the Gebos system

The layers of the Gebos system take into account the distinction between the business section, the business domain, and the workplace context. Several application systems can be based on the different business sections, and different business sections can be based on the same business domain. Two additional framework layers offering general support complete the Gebos system (see Fig. 2):

- **Application Layers** provide the software support for the different workplace contexts.
- **Business Section Layers** consist of frameworks with specific classes for each business section.
- **Business Domain Layer** contains the core concepts for the business as a whole.
- **Desktop Layer** comprises frameworks that specify the common behavior and general characteristics of applications.
- **Technical Kernel Layer** offers middleware functionality and includes specific object-oriented concepts.

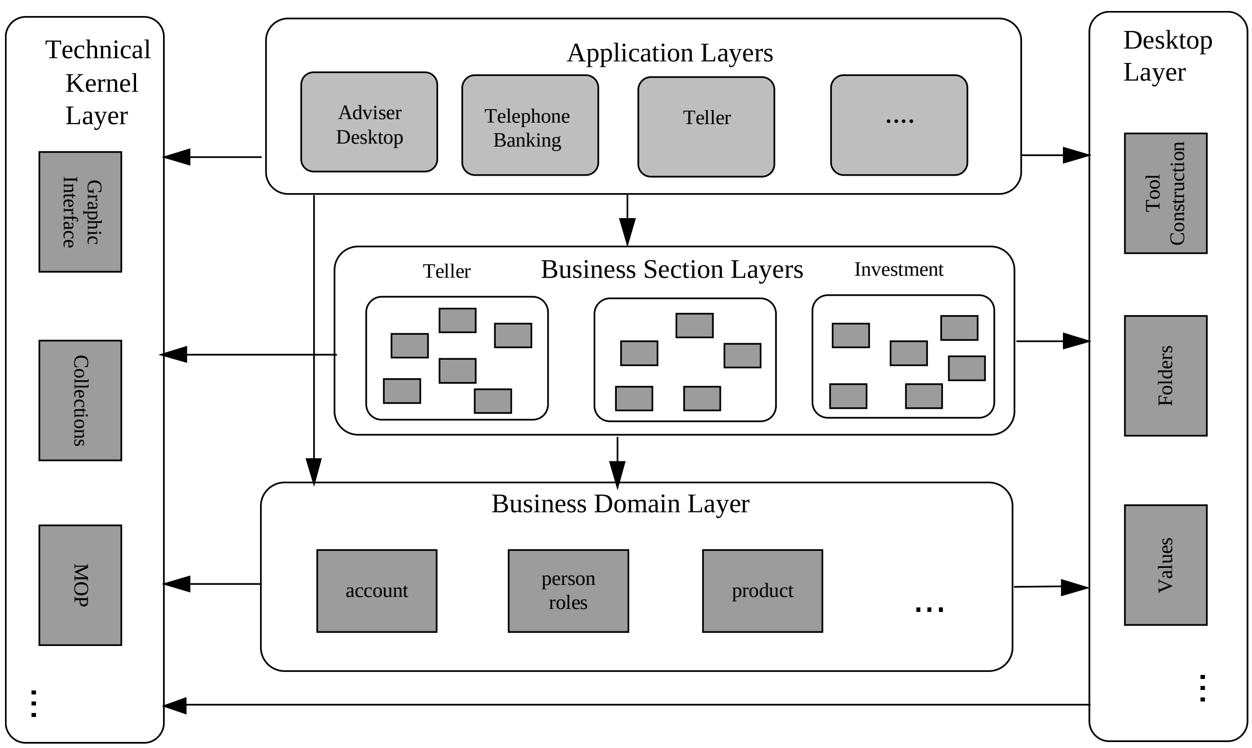

Figure 2: Layers and frameworks of the Gebos system

The framework layers in Fig. 2 are not arranged with the Application Layers at the top end and the Technical Kernel Layer at the bottom. We have, instead, chosen a U-shape consisting of the Technical Kernel Layer, the Business Domain Layer and the Desktop Layer. The U-shape of the layers represents a frame for the Business Section Layers and the Application Layers. By enforcing the integration of the Business Section Layers and the Application Layers, further development can only take place within this U-shaped frame, leading to a fast and efficient implementation of new application systems. The Gebos system makes it possible to configure and adapt new application systems for a new bank in a comparatively short time. We now go on to look at the basic functionality of each layer and the relations between them.

2.2.1 The Technical Kernel Layer

The Technical Kernel Layer provides services to other layers and is used by the other framework layers - like a class library with an API interface. Within the Gebos system, this layer encapsulates and stabilizes middleware functionality. It consists of black box frameworks (see [6]). The frameworks in this layer can be classified as:

- Wrapper frameworks that include frameworks interacting with the underlying operating system, the window system, client-server middleware, and data stores (such as relational databases, CD-ROM drives or host databases).
- Basic frameworks that comprise specific object-oriented concepts such as a meta-object protocol, late creation, garbage collection including trace tool support, and a container library.

The main idea is to reuse the functionality encapsulated by the Technical Kernel Layer. These frameworks are used by all other layers, especially the Desktop Layer. They can be largely reused, as they do not incorporate any domain-specific knowledge.

2.2.2 The Desktop Layer

The Desktop Layer comprises frameworks that specify the common behavior of applications, i.e the type of support for interactive workplaces. Examples of frameworks in this layer are (see Fig. 2):

- The Tool Construction Framework, describing the general architecture of tools and their integration into an electronic workplace (cf. [9]). This can be compared to the MVC framework (cf. [8]).
- The Folder Framework, offering classes such as File, Folder and Tray. Following the desktop metaphor, the Folder Framework provides the familiar look and feel of interactive applications pioneered by the Macintosh system.
- The Value Framework enriches the standard value types offered by programming languages like C++ (e.g., Integer, Char and Boolean). It provides classes containing the basic mechanism for domain specific value types (e.g. AccountNumber), which can only be used with value semantics.

The Desktop Layer thus defines the basic architecture of interactive application systems and their look and feel. This design decision ensures uniform behavior in the application systems as well as technical consistency. The Desktop Layer frameworks are therefore used like white box frameworks (cf. [6]) by subclassing and implementing abstract methods (see Business Section Layer). Frameworks belonging to this layer can be reused in any kind of office-like business domain with graphic workplace systems.

2.2.3 The Business Domain Layer

The Business Domain Layer defines and implements the business’s core concepts as a set of frameworks based on the Desktop and Kernel layer. It thus forms the basis for every application system in this domain. It is crucial to make an appropriate division/separation between that part of a core concept that belongs to the business domain and the parts belonging to the business sections. If the core concept in the business domain is too small, the missing parts have to be duplicated for each business section, and consistency becomes a problem. If a core concept in the business domain becomes overloaded, transporting an appropriate object between the various applications becomes cumbersome.

This layer consists of classes such as Account, Customer, Product and various domain-specific value types. Although some implementations exist in this layer, most frameworks are white box and rather “thin”. The final implementation is postponed to the Business Section Layers.

2.2.4 The Business Section Layers

The Business Section Layers are composed of separate partitions for each business section. The frameworks in these partitions are based on the Business Domain, the Technical Kernel, and the Desktop Layers. They are implemented by subclassing the Business Domain and Desktop Layer classes. Usually, each subclass only implements the abstract methods predefined in the respective superclass. In this layer, we find classes like Borrower, Investor, Guarantor, Loan, Loan Account, and tools for specific tasks within the business section.

Business section frameworks change more frequently than business domain frameworks. A business domain framework should therefore only incorporate those core concepts that are relevant to most business sections. It should, however, provide hooks that can be easily extended and customized for applications that use one or more business sections. To relate the core concepts of the Business Domain Section to their extensions in the Business Section Layers the Role Object Pattern has been developed (see below).

2.2.5 The Application Layers

The separation of the Application Layers from the Business Section Layers is motivated by the need to configure application systems corresponding to different workplace. The Gebos system has to support business activities in a wide range of banks, from those with only a few branches to large ones with over a hundred. Applications configured to meet the different workplace context requirements of an individual bank can be found in this layer.

3 Framework Construction for Large Systems

We now go on to discuss the internal structure of frameworks and layers and the relations between them. First, the internal structure has to be chosen such as to minimize the coupling between different frameworks and application systems. Reopening frameworks during application system development should also be avoided. Second, the design and implementation of business domain frameworks should be independent of any business section framework. We have employed various patterns in the Gebos system, but the Role Object Pattern is crucial for the integration of the business domain and business sections. This pattern will therefore be described in detail.

3.1 Structuring of Frameworks in Library Layers

Based on the discussion above, we group class hierarchies to form frameworks. A framework of the Gebos system does not match a specific class hierarchy, but contains parts of different class hierarchies. This contrasts with the traditional way (cf. [7]) of aligning class hierarchies and frameworks (see Fig. 3).

Each framework of the Gebos System is divided into a concept part and one or more implementation parts (see Fig. 3). The concept part is modeled according to the concepts of the application domain. The implementation part subclasses the corresponding classes of the concept part. The former is developed with technical aspects in mind. Without this separation one cannot discuss the various dependencies between classes with an explicit focus on either the technical or the conceptual part.

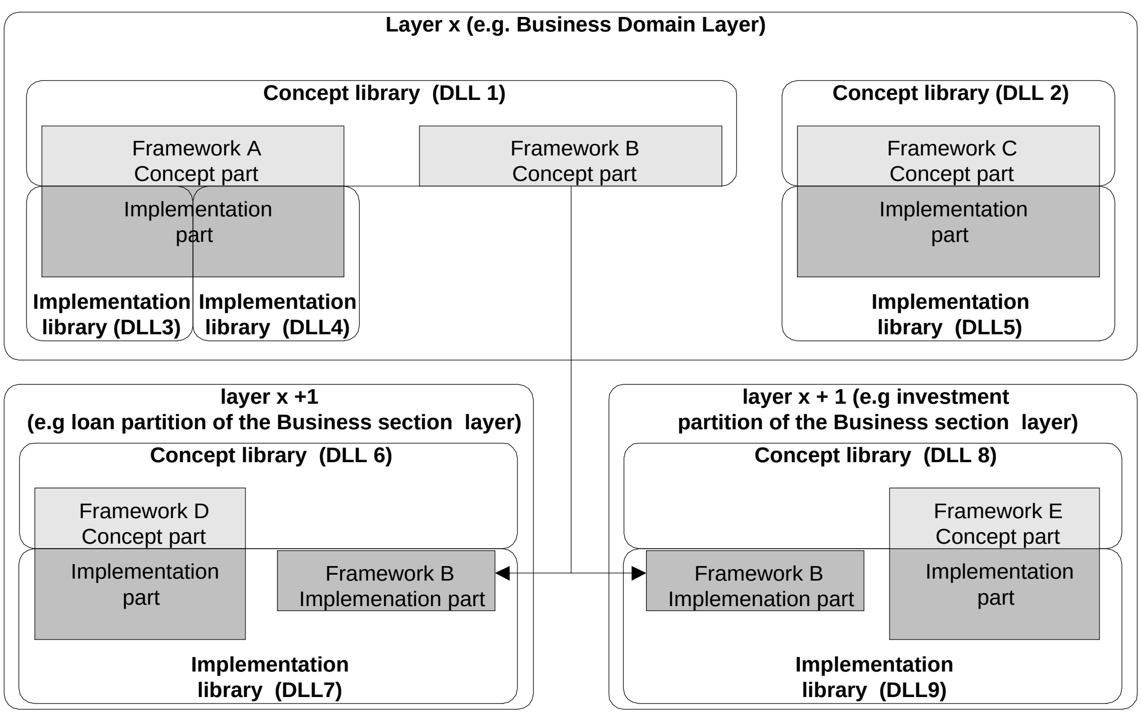

Figure 3: Division of frameworks into concept and implementation parts

After dividing each framework into a concept and implementation part, we physically package concept parts into concept libraries, and implementation parts into implementation libraries. The structure of the concept libraries need not correspond to the structure of the implementation libraries. In our example, Framework A is divided into two implementation parts, each forming a separate implementation library. The concept parts of Framework A and B are packaged into one concept library.

The framework parts themselves are organized in layers according to application domain concepts. Concept and implementation parts of one framework do not therefore necessarily belong to the same layer (see Fig. 3, Framework B). This enables us to describe a concept in the Business Domain Layer and provide different implementations according to specific business sections. The frameworks in the Business Domain Layer in Fig. 2, for example, consist only of concept parts for the core concepts. The implementation parts are located in the frameworks of the Business Section Layers.

The use of classes belonging to other layers is restricted to the concept classes of a framework. Implementation parts of frameworks can then be changed without affecting those classes that use the concept part. To avoid cyclic dependencies a framework should only be used by classes contained in frameworks of higher layers.

By offering libraries that include either several concept or implementation parts of frameworks per layer, the configuration of different application systems is facilitated. Each application system consists of only those libraries needed for operation. The different frameworks are provided in DLLs to ensure that framework classes cannot be changed at will. All application systems are thus built according to the same schema. This is extremely important for classes that predefine and partially implement the flow of control for an application system (see Desktop Layer).

To realize the decoupling between concept and implementation parts, clients should use a creational pattern, such as Factory Method or Prototype [5], to hide the selection of a particular implementation of a concept class. A pattern, that has proved to be particularly effective, is the Product Trader pattern [1].

The division into concept and implementation parts is a prerequisite for developing an integrated system because we can supply a concept part in the Business Domain Layer and several implementation parts in the Business Section Layers. In the following section, we describe how to connect the classes of the concept part with those of the various implementation parts.

3.2 The Role Object Pattern

A simple way to connect concept and implementation parts is to subclass a class from the Business Domain Layer. This, however, means that two instances of different subclasses are not identical. If they are meant to be conceptually identical, it becomes hard to use them as a consistent representation of one logical object. We apply the Role Object Pattern to make one logical object span one or more layers. The core object, which resides in the business domain layer, is extended by role objects, which reside in the business section layers.

A role is a client-specific view of an object playing that role. One object may play several roles, and the same role can be played by different objects. An instance of a core concept belonging to the business domain may play several roles in different business sections. For example, in the loan business section, 'customer' could play the role of 'borrower' or 'guarantor'. In the investment business section, 'customer' could play the role of 'investor'. These three roles could also be played by the same customer, in real life as well as in the system model.

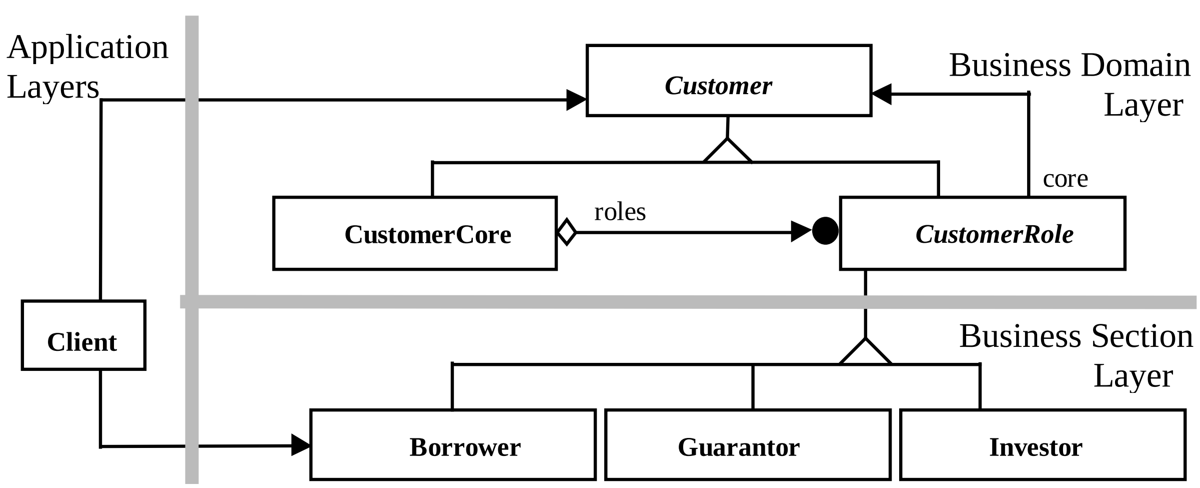

Figure 4: Example of a Role Object Pattern

We use a combination of design patterns, collectively called the Role Object Pattern, to attach business section roles to core concept objects. Figure 4 illustrates the design of the customer example. Using the Decorator pattern, a core concept such as ‘customer’ is defined as an abstract Customer superclass - as a pure interface without any implementation state. The class CustomerRole is a subclass of Customer and “decorates” a Customer object at runtime. The CustomerCore class implements the core of the Customer abstraction. We use Product Trader to create and manage role objects at runtime [1].

At runtime, instances of CustomerRole forward calls to the decorated Customer object (see Fig. 5), which is a CustomerCore instance. Clients work either with objects of the CustomerCore class, using the interface class Customer, or with instances of CustomerRole subclasses. By asking the CustomerCore object for a specific role, a client can obtain the respective role object.

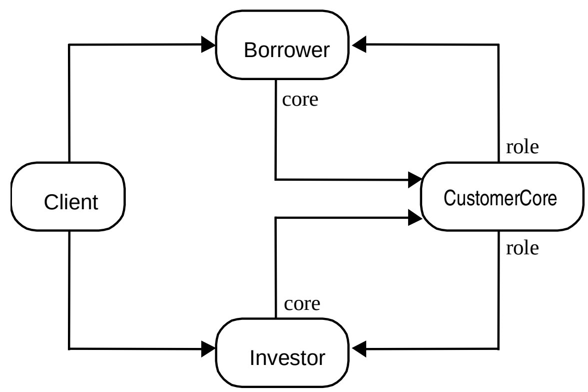

Figure 5: Role Object Pattern at runtime

The Role Object Pattern lets us handle a complex logical object spanning several layers as one coherent integrated object. Role objects from different business sections (e.g., borrower, guarantor or investor) share the same CustomerCore object. Business section specific role objects can be created on actual demand.

We view roles and classes as domain modeling constructs. Others have chosen to ban classes as pure implementation constructs (most notably [8]). For us, a class defines a domain abstraction, which includes the roles that class instances can play. A role defines a context-specific view of an object. On this level, we have to distinguish between technical and conceptual identities. The entire logical object has an identity of its own, even though its role objects maintain their own technical identity.

3.3 Combining Roles and Frameworks

Using the Role Object Pattern frequently leads to a concept part and its implementation parts being located in different layers (see Fig. 3). The concept framework of the Business Domain Layer consists of the Customer, CustomerCore and CustomerRole classes, as well as other core concepts such as Account and Product. Their implementation can be found in implementation frameworks of different Business Section Layers.

This organization of frameworks in concept and implementation parts (see Fig. 6) ensures that extensions to any business domain concepts can be made without reopening the concept part of a framework. If a new business section is to be supported (e.g., housing loans or foreign currency), a new layer with various implementation parts for the concept part of a framework is added to the system. Based on the added Business Section Layer, new applications and extensions of existing applications can be implemented without affecting other layers.

Changes to one business section concern only the classes in the corresponding Business Section Layer. If a new type of saving is to be added to the investment layer, the implementation frameworks must be extended by a new product and a new savings account. None of other Business Section Layers will have to be changed.

Changes to the core concepts in the Business Domain Layer are the only ones that affect all other layers. A change in the classes Customer, Account or Product will change all other layers (see Fig. 6). However, changes to a stable and well designed Business Domain Layer do not occur very often. Such changes can only be caused by a major reorientation of the entire business strategies and services of a company.

Fig. 6 is a simplified representation of the Gebos system’s layers and frameworks. The Business Domain Layer does not consist of one framework only, but contains several of them (see Fig. 2).

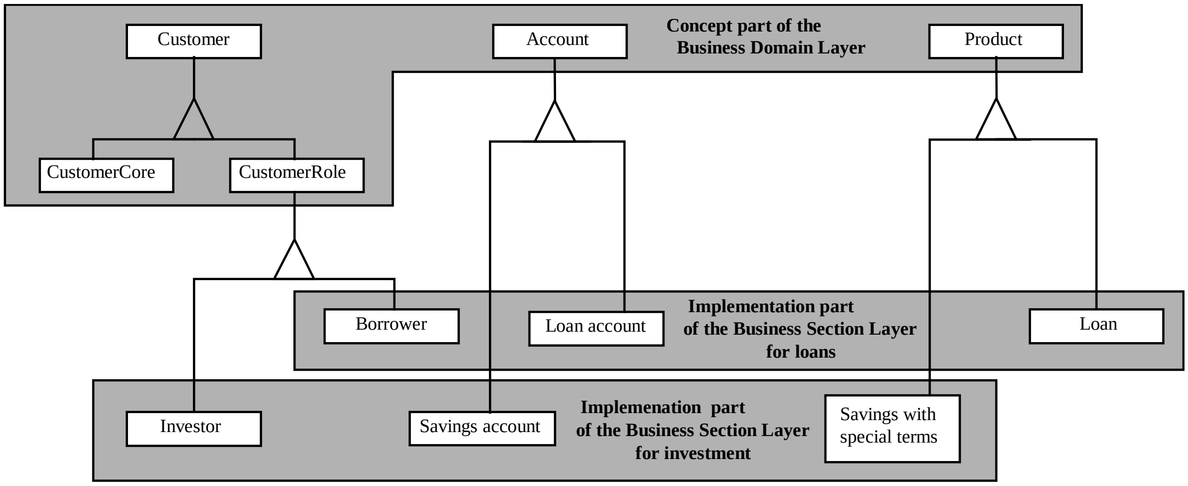

Figure 6: Classes of the Role Object Pattern spread over different frameworks

4 Conclusion

In this paper, we have shown how frameworks can be categorized and layered to match their application domains and the business's organizational structures. We defined the categories business domain, business section, and application framework. Based on these distinctions, we showed how frameworks can be layered in order to manage their dependencies and reduce their coupling. We pointed out how to split a framework into a concept part and several implementation parts, and how to support this split by the use of design patterns.

A particularly important issue is building and maintaining logical objects that span several layers. We use the Role Object Pattern to adapt a core concept from the business domain to different business sections. Using role objects allows us to extend a core concept without having to change it, and thus without having to touch the business domain frameworks.

The presented approach supports the development of frameworks and systems that are both stable and capable of evolving elegantly and at different speeds.

Bibliography

1. Bäumer, D. and Riehle, D. Product Trader. In R. Martin, D. Riehle, and F. Buschmann, Eds., *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, Mass., 1997.
2. Bürkle, U., Gryczan, G. and Züllighoven, H. Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions. *Human-Computer Interaction, Special Issue: Empirical Studies of Object-Oriented Design 2 & 3*, 10 (1995), 293-336.
3. Bäumer, D., Knoll, R., Gryczan, G. and Züllighoven, H. Large Scale Object-Oriented Software-Development in a Banking Environmen - An Experience Report. In ECOOP '96 (July 11-12, Linz, Austria). LNCS 1098, Springer Verlag, Berlin, 1996, 73-90.
4. Garlan, D., Allen, R. and Ockerbloom, J. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software* 12, 6 (November 1995), 17-26.
5. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.: 1995.
6. Johnson, R. E. and Foote, E. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (June/July 1988), 22-35.
7. Lewis, T. *Object-Oriented Application Frameworks*. Prentice-Hall, New York, 1995.
8. Reenskaug, T. *Working with Objects*. Prentice-Hall, New York, 1996.
9. Riehle, D. and Züllighoven, H. A Pattern Language for Tool Construction and Integration. In J. O. Coplien and D. C. Schmidt, Eds., *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass., 1995.