Form-Oriented Analysis

Dirk Draheim · Gerald Weber

Form-Oriented Analysis

A New Methodology to Model Form-Based Applications

With 83 Figures

Springer

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
Takustr. 9
14195 Berlin, Germany
draheim@acm.org

Gerald Weber
Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland 1020, New Zealand
g.weber@cs.auckland.ac.nz

http://www.formcharts.org/

Library of Congress Control Number: 2004112522

ACM Computing Classification (1998): D.2.1, D.2.2

ISBN 3-540-20593-4 Springer Berlin Heidelberg New York

Cover design: KünkelLopka, Heidelberg
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Typesetting: by the Authors
Printed on acid-free paper 45/3142/YL - 5 4 3 2 1 0

Preface

This book presents a special purpose modeling technique for the analysis and design of an important system class, namely form-based enterprise systems. Recent discussions on modeling languages emphasize that there is a strong demand for such domain-specific modeling languages. The class of form-based enterprise systems includes, for example, web shops as well as ERP and B2B solutions and can be said to be paradigmatic for enterprise computing. This book was motivated by the widespread interest in this type of business application from professionals as well as from scientists. The book adapts well-established basic modeling techniques in a novel way in order to achieve a modeling framework optimized for the indicated application domain.

Besides its practical parts the book details theoretical achievements, which lead to real improvements in the application domain of the book. It explains how to model a form-based enterprise system during the analysis and specification phase, and how these models translate into good design. Typical form-based applications have common properties that can be molded into specialized diagram types for such applications. Such a diagram type is the formchart, the central artifact that is described in the book. The formchart is a good example of customized diagrams according to the most recent proposed profiling techniques.

If form-based enterprise systems are modeled with typical general purpose modeling approaches without such customizations, there are a number of obstacles the modeler will face. For example, if one employs use case modeling together with interaction diagrams, the analyst will be confronted with three problems. First of all, the method has to be adapted to the form-based application, since no specific guidance for this special application type is part of the general method. Secondly, the model will become complex for even small-sized problems, since every diagram has to repeat common properties of this very specific application class. Hence the model tends to become highly redundant, and the important distinguishing information is diluted. Furthermore, a third problematic aspect is that current analysis methods are traditionally rather oriented towards event-driven and complex GUI-based applications but not towards form-based applications. Hence the customization demand for form-based applications is particularly high.

Conversely, there are certain benefits the reader of the book can reap by employing the new customized artifacts presented in this book. The reader can obtain faster results and more significant models, because the common properties of enterprise systems are already incorporated in the semantics of the modeling method. The new artifact types presented in this book incorporate the results of studying form-based systems in general, knowledge that a software engineer can hardly obtain in the limited setting of a running project. Our method provides a separation of concerns by splitting the general semantic structure of such applications from the specific information about business logic in the concrete single project. The foundation of the new techniques is fully elaborated in the book for the working developer confronted with everyday problems in professional IT projects. In the same way, the scientist interested in performing novel research on enterprise systems can use this formal reference.

The book is divided into four parts. The first part is a detailed discussion of the new modeling method for form-based systems from a practitioner's viewpoint and explains how the proposed techniques can actually be employed in a project. The second part is about tool support and exemplifies how the concepts introduced in the first part can be exploited by several different implementing technologies. The third part provides the semantic foundation of the different kinds of diagrams and tools introduced in the first and second parts. The fourth part serves as a summary and provides a discussion of related work.

After the introduction the book starts with an in-depth motivation for the new techniques. It is shown that the considered system class encompasses a wide range of important enterprise systems from mainframe/terminal systems through ubiquitous COTS software to modern web applications. The explanations in the book are deliberately based on a realistic running example in order to make a difference. Throughout the book the concepts are exemplified with an online bookshop. This example is not an arbitrary choice of the authors – the important TPC-W benchmark, for example, also uses a standardized online bookshop as a representative example of typical business functionality. The form-oriented information system model is introduced. Different kinds of diagrams for these models, i.e., screen diagrams, page diagrams, form storyboards, and formcharts are introduced for the user interface state part of these models. All of these, and the further model components, i.e., dialogue constraints and the layered data models, are introduced immediately with unambiguous semantics and are used in modeling the running example. Then, techniques for decomposition and refinement are discussed. A parsimonious data modeling language is elaborated. A message approach to the modeling of data interchange is outlined. The book is not primarily about software engineering processes; however, it provides a discussion on how the proposed artifacts can be exploited in an entire software engineering life cycle. The interplay of some proposed best practices that are centered around descriptiveness, artifact orientation, feature orientation, and reuse are discussed. For each concept we show how it can be used to add sustainable value to the respective software engineering activities.

The second part discusses issues of architecture, design, and implementing technology. From this discussion concepts and concrete prototypical technologies for forward engineering, reverse engineering, and the implementation of web presentation layers are derived.

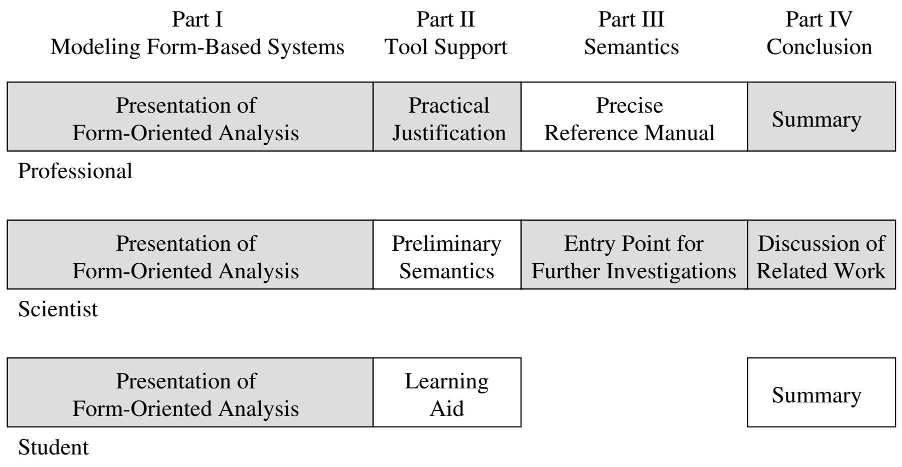
The third part of the book presents the semantic foundation of form-oriented analysis. First, several alternatives for tool support are discussed and given a conceptual basis using an integrated source code model. Then, precise semantics for the form-oriented diagram types are given. For this purpose, a new, lightweight, semantics framework approach is introduced as an alternative to current multi-level metamodeling techniques. Along the lines of the framework approach precise semantics of formcharts, layered data modeling, and the dialogue constraint language are given. This is followed by a discussion of the semantic of the proposed parsimonious data modeling approach. A formal type system for the interplay of server actions and pages of submit/response style systems is provided.

The fourth part provides a focused description of the widely accepted modeling approaches in use. The discussion shows the differences between these approaches and the new method, but it also shows how our method is integrated with standard modeling techniques. For each related method we discuss how it could be applied to enterprise systems and how form-oriented analysis provides a more convenient solution. Therefore this chapter provides a different view on the benefits of form-oriented analysis to the reader. Finally, a summary of the main contributions is provided.

The reader should have some experience with object-oriented programming languages. First-hand experience with visual modeling languages and the graphical tools for them is helpful. Basic knowledge of SQL is also desired for some advanced excursions, but this can be postponed until needed. Related approaches are comprehensively introduced, so that even a reader who is new to these other approaches can follow the arguments.

The book targets professionals, i.e., working software engineers and decision makers, researchers in computer science, and upper-level graduate students who are interested in enterprise systems. Care must be taken, because professionals, researchers, and students typically have different objectives, different dispositions, and different opinions with respect to software engineering topics. This is due to the fact that goals and driving forces are different in industry and academia. Consequently, readers may have different attitudes towards the several parts of the book; see the figure below for a guess. In the figure, supposed main interests are shaded gray, whereas minor interests are left blank.

Professionals actually working on enterprise software will gain a deepened understanding of form-based systems from the abstract system viewpoint pro-


Part I
Modeling Form-Based Systems
Part II
Tool Support
Part III
Semantics
Part IV
Conclusion
Presentation of Form-Oriented Analysis
Practical Justification
Precise Reference Manual
Summary
Professional
Presentation of Form-Oriented Analysis
Preliminary Semantics
Entry Point for Further Investigations
Discussion of Related Work
Scientist
Presentation of Form-Oriented Analysis
Learning Aid
Summary
Student

vided by form-oriented analysis. Many developers already use ad hoc techniques tailored to form-based systems like naive page diagrams or click dummies. These ad hoc techniques arise naturally when developing form-based systems but lack an elaborated conceptual basis. The book allows these developers to strengthen these techniques in practice. Readers can employ the approach directly in projects, because every concept introduced comes with precise semantics and the mutual dependencies between the concepts are elaborated, too. The prototypical forward and reverse engineering tools are suitable for convincing the professional about the potential practical impact of the form-oriented approach. The third part of the book is less important for the professional. If a semantical clarification is needed, this part can serve as a precise reference manual. The professional can use the fourth part as a detailed summary.

Researchers might be especially interested in the third part of the book as an entry point for further investigations. Upper-level graduate students will benefit from the presentation of state-of-the-art knowledge about the development, architecture, and design of enterprise systems in the organizing framework of form-oriented analysis. The second part of the book will help students to grasp more easily the concepts of form-oriented analysis.

Since enterprise applications are a particularly important class of software, almost every IT professional, computer scientist, or computer science student may have some interest in gaining at least an overview of the fundamentals of enterprise computing. The book is written with the different objectives of professionals, researchers, and students in mind. In industry productivity eventually targets return on investment. Product quality and product quantity are limited by productivity. Productivity is limited by the availability of resources. Knowledge acquisition is needed to improve productivity. Academic activity spans two areas that have to be integrated: research and education. While academic research has a subtle target, i.e., the construction of knowledge, higher education has the tangible responsibility to produce well-prepared professionals. Academic research is driven by the pressure to get contributions published in the scientific peer community. Higher education is driven by the demands of the yet uneducated. Altogether these differences result in the following: one and the same concept can be perceived totally differently by individuals in industry and in academia. We encourage all who try to keep an open mind and hope that this book provides valuable information or inspiration.

We are indebted to Martin Große-Rhode for his encouragement and advice. We want to thank our editor Ralf Gerstner for his support and guidance. We would also like to thank the reviewers who made many helpful comments.

Berlin, August 2004 *Dirk Draheim*
Auckland, August 2004 *Gerald Weber*

Contents

List of Figures

Part I

Modeling Form-Based Systems

1 Introduction

What is the business logic of an enterprise system? How do I specify it in such a way that I know how to transform it into a running system, by skill and by automated tool support? This book gives a self-contained introduction to the modeling and development of business logic for enterprise systems.

Enterprise systems are a distinct and highly complex class of systems. They are characterized by their importance for enterprises themselves, making them mission critical, by their extreme multi-user capability, by their tolerance of heavy loads, and by their tight integration with the business processes, which makes every enterprise system installation unique. In short, they are one of the most fascinating yet most demanding disciplines in software engineering. This book is in the first instance intended to be a conceptual introduction to and comprehensive overview of enterprise systems modeling.

Enterprise applications can be singled out within the domain of database applications in general through a number of typical characteristics, which are usually not discriminating if seen in isolation, but which together characterize enterprise applications. An example of a database application very different from enterprise applications is a typical genome database. Enterprise applications store amongst other things data describing events in time. The enterprise system is critical for ensuring data consistency and allowing action consistent with those data in particular. An enterprise system installation controls a certain defined part of an organization. Installing and using the system a second time in parallel for the same business processes leads to serious problems. The correct assignment of the installation to a part of an organization is itself a basic element of this consistency effort, otherwise outdated data is used. This does not mean that multiple installations with correct information about their domain cannot cooperate. It just means that enterprise system installations are in a certain sense territorial. This can be pinned down to the data they manage.

This book explains enterprise systems in the novel form-oriented framework, which is easy to learn and practically applicable yet theoretically powerful. It introduces the form-based system specification as the key notion for the conceptual definition of business logic. It explains state-of-the-art modeling languages and their usage for specifying business logic and for the subsequent building of enterprise systems. The techniques presented here are applicable to enterprise systems of all sizes.

1.1 Enterprise Systems

Enterprise systems are a type of system where so-called business rules are automated. The term business rule refers to a high-level data manipulation rule, which is executable manually or mechanically. But business rules can contain straightforward computations, which we expect today to be executed by a system, or which we even require to be done, not only for convenience, but also for security and data integrity, like the tax on a purchase computed by a cash desk.

There is an intricate relationship between business rules and enterprise systems. Enterprise systems have a distinctive look and feel, because they are mainly systems for managing business rules. On the other hand, if you have a business rule, it is at least conceivable that it will be executed or supported by an enterprise system. Take as an example your mortgage. If you ask your banker about it, the banker should and probably will draw up a financing plan in which you can see what you have to pay each month and what you still owe afterwards. The banker will get this perhaps from the intranet web portal of the bank created by a part of the bank's enterprise application.

In this book we will encounter a uniform viewpoint on business rules, the form-based metaphor. We will learn a general method for describing business rules as functionalities, or *features* of a form-based system. The motivation is not in any way the assumption that business rules must be executed with form-based systems. It is rather the goal to have a common notation, which allows comparisons of different business rules in a common language. If we therefore model a business rule like the mortgage computer above as a part of a form-based system, it is based solely on the assumption that it is *possible* to model the system in such a way, but not on the belief that it is necessary to do so.

One interesting advantage of this uniform modeling approach for form-based systems is, indeed, that by subsuming business applications under this model we single out a well-defined category of system properties in the following, slightly tricky sense. By viewing every business rule as performed with a form-based application, we abstract from the properties of the application, which make no difference in this type of interface. For example, whether a system is used from a touch screen on dedicated terminals, or as a web application over the Internet, might not make a difference with respect to the description of the form-based interface. These differences nevertheless can and should be recognized, but they somehow belong to a different kind of property. The key advantage is to gain a useful classification of different types of system requirements. We therefore introduce the term *business logic requirement* in order to refer to such requirements, which we can express as requirements of the system within our form-based metaphor. In other words, the subsumption of applications under the form-based metaphor is our operative method to separate the business logic requirements from the whole requirement mix. Business logic requirements form the part of an organization's business rules that are to be mechanically executed by the enterprise system.

A full requirements description of a system will consist of the business logic requirements gained in this way as well as all more general requirements. In this book we focus solely on the domain of business logic requirements. This does not reduce the importance of the other requirements in any way. Making the modeling and implementation of business logic requirements more efficient is an encouragement that more care should be taken for these other requirements as well.

1.2 Modeling Enterprise Systems

In this book a semantically precise notion of form-based systems is explained, and called the notion of *submit/response style applications.* This gives us an exactly defined concept of form-based systems.

This notion of submit/response style systems is therefore the framework in which we can model business rules, and it can be seen as a novel concept of a *virtual business machine*, i.e., a machine, which is directly able to execute business logic, regardless of whether it includes human interaction or not. We can now forge the important notion with which we want to refer to business logic requirements in their form as executable programs of this virtual machine. We call the program of this virtual machine the *business logic* of the application.

These considerations lead to another key concept in our considerations, namely *executable specification.* If our business logic can be conceived as the program of a virtual business machine, then our specification method is nothing less than a high-level programming language. In fact, some decades ago, the concept of such languages for data-centric applications appeared so natural that the terminology "fourth generation languages" was coined for these types of languages. This term "fourth generation language" can now be said to be tied to the even more specialized look and feel of such a language, like ABAP-4 in the SAP system. Like this example, fourth generation languages are typically tightly integrated into single-vendor platforms. The modeling method presented here can nevertheless be seen as an integration of practical experiences from fourth generation languages into a state-of-the-art modeling style. In principle it is therefore conceivable that a future business platform running as a virtual machine on common platforms can interpret business logic directly.

1.3 High-Level Transactional Programming

A central topic in this book is the distinction between high-level programming and specification. High-level programming is a programming paradigm, which is on the abstraction level of specifications. Yet a high-level programming paradigm must be effectively translatable in efficient code on standard platforms.

The vision of high-level programming is certainly not restricted to the idea of a fully automated translation. This idealized concept of the translation of the functional specification into a running system on a current platform is neither state-of-the art, nor the most desirable concept. The high-level programming vision rather exploits the broad range of programming notions. Consider higher programming languages, which indeed started as a kind of abstract notation, but which were often not translated automatically, but manually. The task of the programming language was to encode knowledge about good program design.

The most intuitive use of a high-level program is perhaps as an input for a generator. The generator translates the system description into a target code, i.e., it can be compared to a compiler approach. However, the translation process is more transparent, in that the target code can be customized. The difference between a high-level programming paradigm and a general specification paradigm is that the former allows only those specifications that are known to be translatable into code. This is not a contradiction of the previous statement that the translation can also be transformed manually. However, one must be aware of that the code generated from the high-level program may not be a complete implementation: it may well be that there are still system parts that have to be developed manually on the target platform. One could also say that there are also tasks of tuning, which have to be done manually, since the problem is not to generate *some* code that fulfills the specification and is executable, but to generate efficient code.

1.4 A Parsimonious Relational Notation

Throughout the book we use class diagrams for data modeling. We start our examples with some currently discussed modeling approaches, e.g., UML and OCL. From the problems encountered there we will advance to a new, simple data modeling notion, the PD models. It has not just a simple format, but also a precisely defined meaning, its *semantics.* This notation is especially good for training and educational purposes, being an excellent primer for the sometimes more abstruse and feature-laden notations used in industry. It is particularly designed as a preparation of the diverse modeling landscape in industry and research. In industry projects as well as in scholarly discussions one might encounter a whole range of different data modeling notations, like ER diagrams and UML diagrams, but also database schemes, which all turn out to be related approaches to data modeling. The PD model notation in this book is an introduction to the commonalities of all these notations. The PD notation provides a good preparation for the very important skill nowadays to be able to quickly grasp new notations.

You will be well prepared if you keep your mind open, and if you are trained to identify the common concept behind these notations.

Industry notations, like UML, have the property that they must possess deliberately ambiguous semantics, in order to capture a number of differing notions under the umbrella of a unified notation – hence the name. In contrast, the notion used in this book has a fixed format as well as semantics. Of course you are encouraged to use the PD modeling notation presented here in your projects. Hopefully you will experience the advantages of a lightweight yet feature-rich approach that is tailored for today's enterprise applications.

1.5 A Descriptive Approach to the Software Development Process

Form-oriented analysis proposes artifacts for modeling. These artifacts are accompanied by recommendations on when and how to produce these artifacts. However, these recommendations must not be misunderstood as prescriptions – form-oriented analysis is a descriptive approach to software development. This means, first and foremost, that it should not be used to restrict the working software developer in any way. For example, we propose a couple of artifacts for the visualization of our form-oriented information system models, i.e., screen diagrams, page diagrams, form storyboards, formcharts, which differ with respect to granularity, comprehensibility, and preciseness.

Coarse grain modeling is motivated by certain demands of the requirements elicitation process, e.g., the need for lightweight communication with the domain expert, or a desired jump start to modeling. Then, for the sake of a convenient presentation the different kinds of artifacts are presented in this book in the manner of a strict stage-wise process, as if we were proceeding in a project from informal to more and more formal documents. However, this should not convey the impression that the proposed artifacts can only be exploited in a defined proceedings.

The arguments of form-oriented analysis aim at empowering the developer, they aim at improving the modeling of enterprise systems under the overall umbrella of *artifact orientation.* Ideally, form-oriented analysis provides the conceptual underpinning for some personal best working practices that the developer has already discovered by him- or herself, though in an ad hoc manner, or that he or she has learned from a colleague – typically from a senior developer.

In a wider sense, the descriptive approach has several properties, which are not mutually exclusive:

- Orthogonality with respect to process models.
- Focus on conceptual insight.
- Free approach.
- Holistic approach.

Software development methods can have a product model aspect and a process model aspect. Different software development methods can put different foci on these aspects. Actually, we argue that a concrete software development method can be advantageous with respect to product modeling even without reference to any process model. As we have already mentioned, form-oriented analysis techniques and tools are *orthogonal* to process model aspects, too.

Form-oriented analysis focuses on *conceptual insight*. It tries to foster a precise understanding of the certain widespread class of enterprise applications. This does by no means imply that we do not propose concrete formats, concrete tools, and concrete activities. It just means that concrete formats, concrete tools, and concrete activities are subject to concrete elaboration and must not be overemphasized, i.e., they must not be considered more important than the understanding of the system semantics.

The form-oriented analysis approach does not strictly follow any other dominating paradigm. This does not mean that it is completely decoupled from proven techniques and concepts defined so far in the software development community. On the contrary, we believe that form-oriented concepts can be also used in scenarios where other techniques are already successfully established. It just means that form-oriented analysis is *free* from the dictates of other paradigms and metaphors.

Similarly, form-oriented analysis is not restricted to a certain modeling level or process stage, say, requirements specification, analysis or design. As a *holistic* approach form-oriented analysis is open for equal discussions of all kinds of problems the working software engineer is faced with. The only restriction of form-oriented analysis is a self-restriction: it does not aim at being a general purpose approach, but utilizes assumptions about the systems that it is designed for.

2 The Form-Based System Paradigm

Enterprise systems encompass online transaction processing systems, enterprise resource planning systems, electronic data interchange, and e-commerce. This means the system class of interest can contain a small web shop as well as a huge system like the SABRE flight reservation system, which connected 59,000 travel agents in the year 2002 [155].

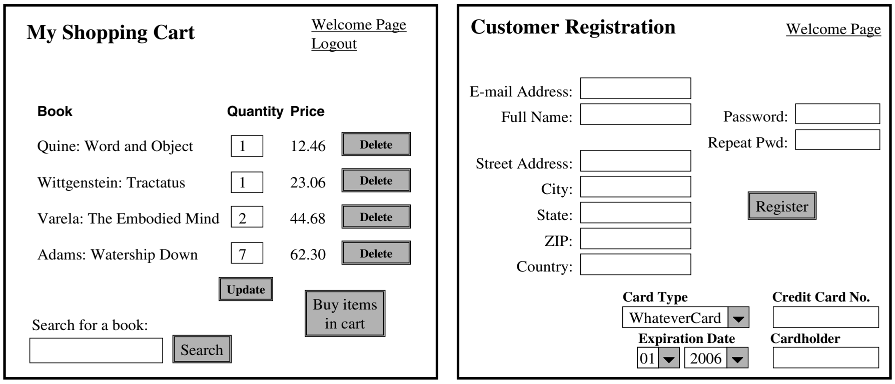
In this chapter we give an outline of our model for enterprise applications. An enterprise system can be seen as an installed and running enterprise application. The basic type of enterprise system we call a *unit system*: that is, a system which we consider as a single unit for our purposes. From the user's perspective a unit system is a black box. It is characterized by the interfaces through which it is accessed. Each unit system is a single unit of abstraction, it is a single *abstract data object*. The interface of a unit system which is directly accessible for us is the *human–computer interface* for interaction with the user. A unit system can also have an interface to other unit systems, and we will call this a *service interface* . Of the two kinds of interfaces of a unit system the human–computer interface is the more tangible one, therefore we begin our outline of the system modeling approach with this type of interface. In our method, the human–computer interface for communication with one unit system is session based, and we call it the *submit/response* style interface. One can conceive of another kind of interface which is sessionless and resembles a mail client with its mailbox. We will discuss such an interface designed for communication with multiple unit systems later in the book. But here we concentrate on the session-based interface type. It captures the key concepts behind several widespread interface types for enterprise applications, e.g., web interfaces. In one sentence one can say that the submit/response style interface models the human–computer interaction as an alternating exchange of messages between the user and the computer. But before we try to understand submit/response style systems in this way we look at them solely from the perspective of the user.

2.1 The Submit/Response Style Interface

We introduce the class of submit/response style interfaces by using a familiar application as an example, namely an online bookshop as can be found frequently in a similar form on the Web. Chapter 3 is devoted solely to an informal description of this example bookstore.

We have designed the following considerations in such a way that the reader can participate in the development of the ideas about the interface types. This is intended to be neither a historic line of development nor a necessary argument; it is just considered to be helpful, instructive, and easy to follow.

Submit/response style interfaces show at each point in time a page to the user, the *current page*. Two such pages, which are taken from our example bookstore, are shown in Fig. 2.1, i.e., a page showing the contents of the user's shopping cart and a page for gathering personal data.



Book	Quantity	Price
Quine: Word and Object	1	12.46
Wittgenstein: Tractatus	1	23.06
Varela: The Embodied Mind	2	44.68
Adams: Watership Down	7	62.30

Card Type: WhateverCard
Expiration Date: 01 2006

Fig. 2.1. Example pages of the online bookshop

A submit/response style interface allows the user to perform two kinds of interactions with the interface: we call them *page edits* and *page change*. Page changes are singular interactions which change the page, i.e., the current page is replaced by a new page. Page edits are interactions with the current page, namely the filling out of a form or resetting a form. Forms are the only editable parts of the page, and are made of input elements. These input elements can be quite sophisticated by themselves. A very sophisticated form element is a text field that allows the input of formatted text, as can be found in some interface technologies.

There is a hierarchy in these two kinds of interaction. Take the search option as an example. First you enter keywords by page interaction. Then you press the search button and the system shows the page with the search results by performing a page change. The page edit is always a preparation for the page change in this style. We call this the *two-staged interaction* paradigm of submit/response systems.

During the heyday of GUI-based client/server programming such interfaces were often considered as bare metal legacy technology. The advent of the web browser as a new thin client has shown many reasons why submit/response style interfaces are here to stay. On the one hand there are proven system architectures for submit/response style systems. Classical mainframe architectures like CICS are still in use and being constantly improved. Some ubiquitous commercial off-the-shelf (COTS) products are successful because they have a mature system architecture. They provide working solutions for enterprise applications, and they take into account the substantial non-functional requirements of enterprise applications. New vendor-neutral and platform-independent enterprise computing approaches like J2EE are emerging, targeting the same driving forces such as the classical approaches.

But submit/response style systems do not just have proven software architectures. Surprisingly, submit/response style interfaces can have cognitive advantages, too. This means that submit/response style interaction can foster usability in many cases, simply because it is often the natural solution with respect to an automated enterprise functionality.

2.1.1 Proven System Architecture for Submit/Response Style Systems

Enterprise applications are data-centric and transaction-based. The submit/response style interface is not tied to any specific technology. On the contrary, the same characteristics can be found in many technologies, e.g., HTML-based clients and mainframe terminals. Even the screens of a GUI-based COTS system follow the submit/response style interface metaphor.

An important class of systems with submit/response style interfaces are systems with ultra-thin clients, encompassing terminals and HTML browsers, see Fig. 2.2. Ultra-thin clients are used for creating an interface tier that does not contain business logic in itself. Ultra-thin clients cache the user interaction on one page in the client layer. The page sequence control logic – or workflow controller – is also not hosted by the client layer but rather by the server layer. Ultra-thin clients fit neatly into the transactional system architecture, be it one of the classical proposals [23, 130] or a more recent proposal [181]. Transactional system architectures successfully target many problems: system load, performance maintainability, scalability, security, and others.

The interaction with a system/response style system is a repeated alternation between data processing and the presentation of a new screen. The dialogue appears to the user as a sequence of editable screens: the dialogue steps are *screen transactions*. The presentation layer of a system is responsible for a preprocessing of data submitted by the user, the triggering of appropriate business rules, and the presentation of the correct new screen. Given a multi-tier system architecture, there is no requirement that this logic be hosted by

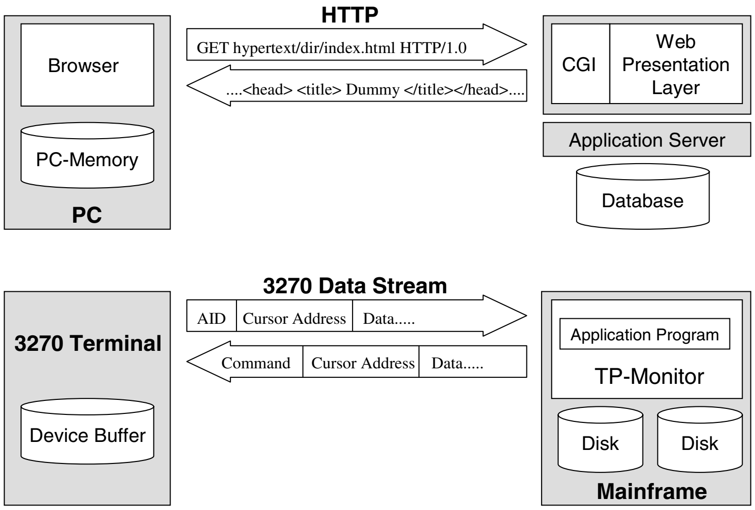


Fig. 2.2. Examples of ultra-thin client based submit/response style systems

the application server tier. In the SAP R/3 system [208], see Fig. 2.3, it is actually hosted by the client tier. The SAP R/3 system architecture is optimized with respect to the notion of commercial off-the-shelf software. In a full version of the SAP R/3 system the vertical architecture depicted in Fig. 2.3 is completed by a horizontal architecture consisting of a production system, a consolidation system, and a development system: the necessary customization of the system is only possible in a defined safe way by deploying new modules via a special transportation system.

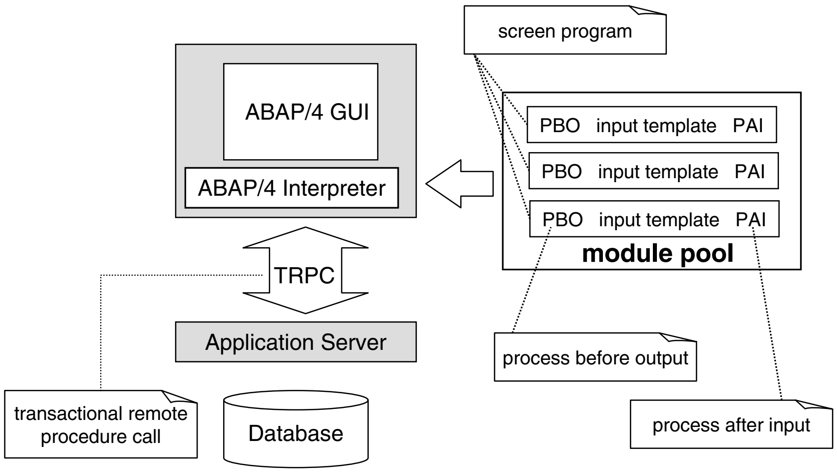


Fig. 2.3. SAP R/3 architecture – a client/server submit/response style system

2.1.2 Cognitive Advantages of Submit/Response Style Systems

Form-based interfaces have clear advantages for the self-explanatory character of a system. The usage of the system is intuitive, since it is guided by a paper form metaphor. However, the importance of the submission process is notable; therefore we want to characterize the metaphor as a *submission form metaphor*. The difference between temporary input and submission, or "sending," is intuitive and fosters the user's understanding of the system. The form-based metaphor has a multi-tier structure of its own, without being fixed to an implementation. The two classes of interactions structure the work of the user into the work-intensive frequent page interactions and the punctual and atomic interactions of the "serious" kind, namely the page changes which also happen to be the conclusion and separation of logically disjoint bunches of work. The submit/response style character puts the user in command of the timing of system usage. It protects the user from irritating disruption of his or her work by incoming information.

In form-based interfaces the submission of a form is an operation that has exactly the semantics indicated by the metaphor. In computer science terms we have compared the submission of an actual parameter list with a method name. The submission form metaphor views interaction with the system as filling out virtual paper forms and submitting them to a processing instance, which represents the core system.

The metaphor has the qualified name submission form metaphor, because other form interface types can be found as well. For example, desktop databases as found in office suites allow form style interfaces, which possess page navigation buttons. Input in this form immediately changes the model. We call such a form style interface a *formlike view*. Applications using formlike views are in principle required to have synchronous views of the data: if two formlike views currently show the same data, and the data are changed in one formlike view, then the other formlike view has to be immediately updated. Many implementations, however, have to stick to polling mechanisms, which leads to latency effects in the update process. Well-known and even worse examples are file managers, which recognize state changes frequently only after manual refresh. It is important to recognize that the necessary refresh in this case is a bad implementation, while the reload mechanism of submit/response style applications is a logically necessary feature.

In desktop databases the model state is the persistent state. Other applications with formlike views have non-persistent states, e.g., spreadsheets.

The submission form metaphor has the advantage of possessing a clear semantics. The two-staged state change due to the two-tiered model is an integral part of the metaphor. This is quite in contrast to, for example, the important desktop metaphor. Consider the important drag and drop feature, which is at the very heart of the desktop metaphor. Drag and drop means regularly either copy or move, and hence can lead to two different effects.

The submission form metaphor is accompanied by the *response page principle* for showing responses of the core system. The submission of a form is a page change, i.e., the page that hosted the form is hidden and a new page is shown. This new page is the response from the server. The response page has three important functions:

- Notifying the user of the immediate status of the submitted form.
- Showing new information to the user.
- Offering new interaction options.

The immediate status of the submitted form is the system's immediate response to the form. Depending on the business logic this may or may not be the completion of the form processing.

- Consider the entry of a new date in a web calendar tool. The response page is the new calendar view with a short notification message. The form has been completely processed.
- Consider the submission of an order in an online brokerage system. The response page is a notification of reception. The execution of the order, however, takes place asynchronously.
- Consider the submission of an e-mail in a mail account on the Web. The response page logically is only a notification of some overall validity of the submitted data, e.g., the recipient's address contains an at-sign. The completion of the intended effect, i.e., the delivery of the e-mail, is not acknowledged at all.

2.1.3 Semantics of Page Change

Each page can contain different forms. A form connects input elements to a page change option, the submit button. Of course the intuition is that only the page edit in the form that belongs to a submit button gives the intended meaning of this command.

This meaning is captured in a message-based model, which we use for the submit/response system. The user interface is considered as a distinct system, which we call the *conceptual session terminal*, or terminal for short. It is very much an abstraction of today's web browser used as a client for an enterprise application. As the name terminal suggests, the terminal is considered to be connected with the unit system, which means that it can communicate by messages and only by messages. The conceptual session terminal has a state, namely the current page shown to the user, optionally including some invisible information within that page, as well as the page edit the user has performed so far on this page. The page edit is kept in the terminal until the user performs a page change, which belongs to a form. Then the contents of the input elements of this form are transmitted as a data record to the unit system. The input elements of this form can contain the page edit of the user, or data which was pre-filled on the page, so-called default data. The transmitted data record is tagged with the name of the form, and together this message is like a remote procedure call. The name of the form is something like the name of the procedure. It leads to an action on the unit system, and this action always produces a result, given as a page description. This page description is a message that is transmitted back to the terminal, and the described page replaces the current page the user has seen before. This gives in effect a page change. We call the new current page the *response page* to the submission, the received message the response message, and if we do not want to distinguish it, we call it the *response.* The page change is therefore the submission of a parameterized command, and the new page is the result. The terminal is locked between send and receive. The remote method call is therefore a synchronous procedure call. This alternation of submissions and responses has of course given rise to the term submit/response.

The parameter of the submission can of course be empty. There is only one type of page change. Each page change can, however, transmit data, which were not rendered on the page.

All page edits that have been performed on input elements which do not belong to the submitted form are therefore lost; the state of the conceptual session terminal after a page change is exclusively the response page.

Of course there are many other possible types of interfaces than the submit/response interface explained so far, e.g., interfaces which support several pages at once. However, the submit/response style system has its advantages in that it is quite expressive yet simple and primarily it is very regular. The strict alternation between the user and system messages yields many advantages for modeling. Therefore it is very suitable for the high abstraction level, on which we want to focus during analysis. A key concept here, which contributes to the whole method's characteristics, is the notion that the user can submit a whole compound data object with each message.

2.1.4 Dialogue Types

We have explained submit/response style interaction as the alternating interchange of messages. We now want to introduce static typing to these messages, and this step alone will lead to a plethora of interesting new properties of our interfaces.

First we want to introduce static types for the response pages. This means that only a finite number of page types are allowed for each interface. It allows us to give a natural yet rigorous meaning to the finite number of pages depicted in screenshot diagrams (sometimes called non-executable GUI prototypes); they simply represent the page types. Furthermore it allows us to give precise semantics to the arrows in these diagrams representing possible page change in the following way.

The current page has a type from a finite number of page types. We conceive the type of the current page as a *finite state aspect* of the terminal. (A finite state aspect is a reduction of the state of a system, which is of interest for the modeler. This is known from finite state modeling in many domains. Consider the finite states a process can have in an operating system. Of course each process can in principle have infinite states, but the finite states are the reduction of interest for the modeler.) The terminal can then be seen as a finite state machine. The arrows are naturally characterized as transitions.

We now turn to the user messages. They are statically typed as well. Therefore there can be only a finite number of possible user messages. Each form on a page must be assigned a single user message type. The page edit on this form prepares an instance of this type. The page change is then used for sending this instance as a method parameter. We identify the concept or the type with the concept of the procedure name. Therefore the type of the message already determines the processing action of the unit system. We call this procedure of the unit system the *server action.* A form on a page is therefore an editable message instance.

For each page type the number of page changes is constant or bound by a constant. Consider a catalogue page which contains a list of books. Each book can be put into the shopping cart with a single click. If we model these interaction options as separate page changes, then the number of page changes is not bound by a constant. We therefore conceive all these interaction options as addressing the same page change, but providing a different parameter every time. In this way the list of interaction options forms a single conceptual interaction.

2.1.5 Conditional System Response

If a message is sent to the unit system, the system's response is conditional, depending on the message and on the system's internal state. Of course the system's response is conditional with respect to the content of the page, e.g., in the case of selecting a book, the shopping cart as the system's response depends on the previous cart state as well as on the chosen book. But the system's response can be conditional with respect to the page type as well. Take a system login dialogue as an example. The business logic says that if a user has never bought anything, then after six months the username will be deactivated and can be taken by another user. The submission of username and password can therefore have a number of different effects.

- If the username belongs to a valid account and the password is valid, then the welcome screen for registered users is shown.
- If the username belongs to a deactivated account and the password is valid, then the user gets a screen informing him or her that the account has been reactivated.
- If the username belongs to a valid account which has been taken over by a user, and the password is the last password of an old user, then the user gets a notification that his or her account has been collected and redistributed. A new account is offered to him or her.