T U M

I N S T I T U T F Ü R I N F O R M A T I K

Software Architectures and Design Patterns in Business Applications

Manfred Broy
Ernst Denert
Klaus Renzel
Monika Schmidt
(Editors)

TUM–I9746
November 97

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I9746-200/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

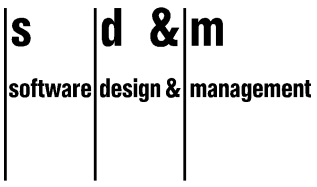
©1997

Druck: drucken + binden gmbh
Schellingstraße 23
80799 München

Software Architectures and Design Patterns in Business Applications

Manfred Broy
Monika Schmidt
Ernst Denert
Klaus Renzel

Editors



Institut für Informatik
Technische Universität München
D-80290 München

sd&m GmbH & Co. KG
Thomas-Dehler-Straße 27
D-81737 München

Table of Contents

Preface V

Part I: Introduction 1

Approaches to Software Architecture 3
Christoph Hofmann, Eckart Horn, Wolfgang Keller, Klaus Renzel, Monika Schmidt

Wege zu objektorientierten Software-Architekturen 47
Peter Brössler, Wolfgang Keller

Part II: Architecture for Business Information Systems: A Pattern Approach 61

Three Layer Architecture 63
Klaus Renzel, Wolfgang Keller

Form-Based User Interface - The Architectural Patterns 69
Jens Coldewey, Ingolf Krüger

Decoupling of Object-Oriented Systems - A Collection of Patterns 91
Jens Coldewey

Client/Server Distribution - A Pattern Language 117
Klaus Renzel, Wolfgang Keller

Relational Database Access Layers - A Pattern Language 139
Wolfgang Keller, Jens Coldewey

Error Handling - A Pattern Language 165
Klaus Renzel

Part III: Formal Description Techniques 199

Beschreibungssprachen für Software-Architekturen 201
Christoph Hofmann, Klaus Renzel

Semantic Concepts for Software Architectures 241
Manfred Broy

Towards a Mathematical Concept of a Component and its Use 261
Manfred Broy

A Graphical Description Technique for Communication in Software Architectures 283
Manfred Broy, Christoph Hofmann, Ingolf Krüger, Monika Schmidt

Case Study: Describing the Interaction Architecture of a Relational Database Access Layer using EETs 311
Monika Schmidt

Exemplary and Complete Object Interaction Descriptions 339
Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, Wolfgang Schwerin

Stepwise Refinement of Data Flow Architectures 349
Jan Philipps, Bernhard Rumpe

Preface

It is common knowledge among software developers that good design is more important for large software systems than high programming skills. Beside project management aspects it is the design ability of a software project team that makes the difference:

1. Are new requirements easy to implement or have large parts of the system to be changed in order to implement minor requirements?
2. How comprehensive is the design of the software system and its documentation, for instance, to phase in new team members?
3. How much effort does it take to port the software system to a different environment (for instance, another operating or database system)?
4. Can a different user interface be implemented without having to adapt major parts of the system that are not directly involved in the human-computer interface (for instance, systems often require different user interfaces to support specialists, semi-trained users, or clients who access the system via WWW)?
5. Is it possible to integrate the system in a new structure, for instance, as part of a workflow?
6. Can the system be tested component by component or have system integration tests be run as the only chance for quality assurance?

The keywords to the above questions are, among others, flexibility, clarity, maintainability, adaptability, modularity, testability. The only way to build these qualities into software systems is to spend a major portion of the overall development time and effort in developing a good design for the software system. Unfortunately, a large number of today's software does not meet these goals.

sd&m software design & management GmbH Co. KG has been founded with the idea of good software system design (and project management) in mind. The business mission of this company is to build large individual business information systems for customers who cannot (or do not want to) use software off-the-shelf. Obviously, software system design has played a large and important role at sd&m ever since its foundation 1982. A comprehensive presentation of the sd&m approach to the design and construction of large software systems can be found in [Den91].

The research of the team of Professor Broy at the Faculty of Informatics at the Technical University of Munich concentrates on a scientific foundation of software engineering.

In 1994 Ernst Denert and Manfred Broy decided to start the research project ENTSTAND[1] (development of a standard architecture for business information systems) which was later on renamed to ARCUS. This project was a co-operative effort of the Technical University Munich and sd&m to capture the essentials of good business information systems design. This report presents the results of this project.

---

[1] The project ENTSTAND was funded by the BMBF under the numbers 01 IS 508 A 0 and 01 IS 508 B 2.

sd&m has made a lot of experiences with design of business information systems. The book Software Engineering [Den91] reflects some of this experience, but its scope is not to deliver detailed design knowledge. Rather it describes the complete software development process, leaving only a few pages for design issues. Prof. Broy and his group have a long and well-reputed record of research in the specification of software systems. The project brought these two groups together to produce precise and sound descriptions of good design for business information systems.

The two groups spent considerable time to investigate the fundamentals of software architectures (see "*Approaches to software architecture*", "*Wege zu objekt-orientierten Software-Architekturen*" in Part I and "*Beschreibungssprachen für Software-Architekturen*" in Part III). Soon it was clear that a standard notation for software design was needed, a notation that would be

a) precise enough to avoid misunderstandings and

b) intuitively understandable to software designers.

Design Patterns have been introduced into the field of Computer Science by Bruce Anderson, Kent Beck and War Cunningham in 1989, and had their breakthrough in 1994, when Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published [GOF95]. Patterns have originally been used by the building architect Christopher Alexander to capture the "quality without a name" he observed in certain cities, towns, and buildings [Ale79]. This style of describing a proven solution to frequently occurring problems in a context has been chosen by the ARCUS team to describe the design for business information systems.

Part II contains design patterns (in shortened versions), which have been derived from sd&m projects. The introductory three-layer-architecture serves as a kind of roadmap. To fill this roadmap completely, many more design patterns are needed. The collection in this book (form-based user interface, decoupling, client/server-distribution, relational database access, and error handling) is a starting point. In fact, the list of necessary design patterns will never be completed because new technology always drives architectural developments as well.

The work of Part III complements the pattern view on architecture by more formal approaches for the description of software architectures. The paper "*Beschreibungssprachen für Software-Architekturen*" provides an overview and comparison of some selected description techniques. The paper confirms that many architectural description languages either lack a semantic basis or they are embedded into a semantic framework chosen from existing specification or programming languages. The semantic concepts needed for the specification of software architectures have not been studied systematically yet. Therefore, "*Semantic Concepts for Software Architecture*" and "*Towards a Mathematical Concept of a Component and its Use*" explores the mathematical concepts on which a formal notion of software architecture can be based. Of course, the use of a mathematical based description technique can include graphical description techniques. Many graphical design notations use a form of interaction diagrams to model the component interaction in software architectures. Most of them are suited for exemplary interaction scenarios but not for a precise description of the complete interaction architecture. An approach to model the complete interaction behaviour is presented in "*A Graphical Description Technique for Communication in Software Architectures*". The remaining articles of this chapter concentrate on improving description techniques for design patterns and software architectures.

A good research project ends with more ideas for future research than it started with. The use of design patterns together with more formalised descriptions of architectural designs will have to be investigated in practical software design studies. Design patterns should be complemented by a design methodology that derives a software architecture instead of (re)inventing it. The derivation process should be fed by the functional and non-functional requirements and by available design patterns (naturally from the large scope of world-wide available design patterns, such as [BMR+96, CS95, Fow97, MM97, MRB+97, VCK96]). We do not state that patterns may substitute creativity in the software design process in the future. There will still be enough room for creativity when transforming the requirements to an architecture, but standard requirements can be fulfilled in a standardised way using existing design patterns. Last but not least the design methodology should be supported by pattern-based tools. Some early examples of tools are available [BFV96, HRS+97, PW96, PHP97] but they are by far not satisfactory yet: instead of supporting a pattern-based design phase, these approaches mainly focus on code generation from design patterns.

References

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobsen, I. Fisksdahl-King, S. Angel: *A Pattern Language - Towns, Buildings, Construction*, Oxford University Press, New York, 1977

[Ale79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, New York, 1979

[BFV96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides: *Automatic code generation from design patterns*, IBM Systems Journal, Vol.35, No. 2, 1996

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern Oriented Software Architecture, A System of Patterns*, Wiley, 1996

[CS95] J.O. Coplien, D.C. Schmidt (Eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995

[Den91] E. Denert: *Software-Engineering*, Springer Verlag, 1991

[Fow97] M. Fowler: *Analysis Patterns - Reusable Object Models*, Addison-Wesley, 1997

[GOF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995

[HRS+97] F. Heister, J.P. Riegel, M. Schuetze, S. Schulz, G. Zimmermann: Pattern-Based Code Generation for Well-Defined Application Domains. In Frank Buschmann, Dirk Riehle (Eds.): *Proceedings of the 1997 European Pattern Languages of Programming Conference*, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997

[MM97] T. Mowbray, R. Malveau: *CORBA Design Patterns*, Wiley, 1997

[MRB+97] R.C. Martin, D. Riehle, F. Buschmann (Eds.): *Pattern Languages of Program Design 3*, The Software Patterns Series, Addison-Wesley, 1997

[PHP97] Eric Huss: Patterns Home Page - Tools and Sample Code, [http://st-www.cs.uiuc.edu/users/patterns/tools/]

[PW96] B.U. Pagel, M. Winter: *Towards Pattern-Based Tools*. In: Preliminary Conference Proceedings EuroPLoP, First European Conference on Pattern Languages of Programming, Irrsee, Germany, 1996

[VCK96] J.M. Vlissides, J.O. Coplien, N.L. Kerth (Eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996

Part I

Introduction

Growing complexity, distribution, and networking are the most important reasons why a well-organized structure of software systems is indispensable. A good structure is key for making it easier to build and maintain a software system. Therefore, in recent years software architecture has attracted a lot of attention in computer science practice and research both in industry and academia. To date there is no satisfactory formal definition of the term software architecture. It is often used to denote both the gross structure of a software system and the discipline of how to structure a software system.

While the efforts within the basic research community concentrate on formal descriptions of software architectures, present industrial approaches focus on the reuse of pragmatic structuring principles that have been applied successfully in various projects. The article *'Approaches to Software Architecture'* contains a brief discussion of the role of software architecture within the software development process and gives an overview of several theoretical and pragmatic approaches. Due to the dynamics of this research area, the article can only provide an overview of the ongoing work in the field and does not claim to be complete.

Because complex software systems have to fulfil many requirements, building a good software architecture is a very ambitious task. Nowadays, software systems are rarely developed from scratch. In most cases existing software is modified in order to fulfil additional requirements and completely new subsystems have to be integrated into a legacy system. The latter tend to be monolithic, host based, and transaction oriented. Because a complete replacement of these systems at a certain deadline would induce too much risk, strategies for step-by-step migration are very important. The article *'Wege zu objektorientierten Software-Architekturen'* describes several strategy patterns for migration of legacy systems to systems with a distributed object-oriented software architecture.

Part II

Architecture for Business Information Systems: A Pattern Approach

Today's pattern literature can be compared with dictionaries, which define a vocabulary designers can learn and work with. To study and document patterns on the level of architectural design means to talk about more complex structures than atomic patterns. Therefore, architectural patterns must address the "grammar" of a pattern language. They should illustrate the rules by which a designer can build concrete "phrases" knowing the basic vocabulary. Hence, software design knowledge comprises a number of basic design elements as well as rules how these elements can be composed to form larger software structures.

One goal of the ARCUS project was to capture the design knowledge of sd&m in building large business information systems, which can be roughly characterized by: database oriented (mostly relational), mission critical, on-line transaction as well as batch processing.

This part presents the results of our approach to apply the pattern concept to this particular application domain: First, a top-level view on the architecture of business information systems is given by a "Three Layer Architecture". The layers within this architecture are described as black-box components, which are then refined by pattern languages presented in the following papers. These pattern languages cover the main technical aspects in the design of the individual layers: user interface design, decoupling of components, distribution, relational database access, and error handling.

Future work may continue and complete this approach by investigating further patterns for refinements and extensions of the Three-Layer-Architecture (e.g. workflow, object-oriented user interface). Additional work is necessary for the integration of patterns in the design process, which requires a methodology and tool support.

Part III

Formal Description Techniques

Software architecture as a scientific and engineering discipline aims at providing mechanisms and tools that support all phases of software development for large software systems, in particular business information systems. Given the sheer size of the latter, and the diversity of tasks that have to be carried out during the development process, trying to develop and understand a system's architecture from a single point of view seems an unmanageable task.

A much more practicable approach is to deal with different perspectives ("views") of a system, such as (among many others) the data-structure, component distribution, interaction, and behaviour/automata view. Each of these focuses on some important aspect of the system while abstracting away from others. This helps to reduce the amount of complexity a developer has to cope with when reasoning about a system. One reason why the use of patterns as a means of presenting software architectures has attracted so much attention over the past few years is that a pattern description directly addresses various different perspectives of a system's structure.

This "separation of concerns" on the architecture-level immediately leads to a number of questions that have to be answered: What does an adequate set of views look like for a given architecture? What description techniques should be used for the presentation of the views? How expressive should these description techniques be? How are the different system views related? How to obtain a refined view from an existing one, thus yielding a more detailed system description? Providing precise mathematical models for the various system aspects significantly increases our potential for selecting the right set of views, for defining the consistency among different perspectives, and for introducing the notion of architectural refinement to support the construction of large software systems within and across all development phases.

The contributions presented in this part address formal approaches at dealing with software architecture and try to answer some of the questions given above. The authors deal with various approaches at defining description languages for (certain aspects of) software architecture, and aim at clarifying and formalizing important terms of software architecture as well as at providing a formal treatment of the latter. Together, these texts clearly demonstrate that using formal mathematical models for the description of (parts of) software architectures helps both to clarify the underlying concepts of "programming in the large", and to increase our ability of coping with the architecture of complex systems by means of "separation of concerns" and abstraction techniques.