

# Modeling Business Applications with Patterns

André Zwanziger and Sebastian Herden<sup>1</sup>

## Abstract

Patterns are best-practice examples to solve recurring problems. They help to understand a problem and its solution. Moreover patterns promote a well-structured and flexible design of software applications. In the past years, such recurring problems were identified in business application development and were documented as patterns. This article shows an overview of patterns in different phases of the software development process.

## 1. General

Building business applications challenges customers and software developers to define and to create the right things together. Customers must express their needs precisely and sensitize software developers to their problem. Software developers must understand the problems of their customers and support them in technical decisions. Building business applications increase the importance of a good collaboration between customers and software developers, because every decision made in the software development directly effects the quality of the software and additionally result in economic aspects during the whole software lifecycle.

At the present time, business applications are widely no stand-alone applications, they often communicate with other existing applications and manipulate existing data sources. They must be stable, fault-tolerant and have to meet the needs of different users everytime and in future times also everywhere. Business applications must be robust, which means that they have to resist high access rates and must manage large amounts of data. Moreover it should be possible to update and upgrade the software. Last but not least, business applications must be secure, so that unauthorized users cannot manipulate the system.

This short list of general requirements gives a hint, that building business application is more than hacking functionality into computer systems. But this list also shows, that these requirements are neither customer specific wishes nor are they new. In fact, a lot of software systems have been written and it could be possible that one solution for a specific problem was very effective and efficient. Why not reuse this solution in a similar situation, instead of reinventing the wheel?

Patterns can be used to document and share these problem/solution pairs. In the field of software development many patterns were identified and documented. The goal of this article is to show, which patterns can be used in different phases of software development (see Dumke, 2001) and to give an insight of some software pattern. The paper does not cover, which patterns can be used in a special moment of a special software development process (e.g. waterfall model or extreme programming).

---

<sup>1</sup>Otto-von-Guericke University of Magdeburg, School of Computer Science  
Dep. Of Tech. And Bus. Inf. Sys., Business Information Systems I  
e-mail: {zwanzige, herden}@cs.uni-magdeburg.de

The document is organized as follows: Section 2 gives an methodical introduction into patterns and pattern languages. In section 3 a rountrip through software development phases is made and some helpful patterns are shortly named and explained. Finally a conclusion is made in section 4.

All patterns in this article are emphasized in *italics*. Unfortunately, it is impossible to discuss all patterns shown in this article in detail. This article shall only give an overview of patterns. The interested reader may refer to the literature to get more detailed information about special patterns.

## 2. Patterns and Pattern Languages

A pattern describes a problem and its general solution. They document recurring structures and provide “best-practice” solutions for a problem. A pattern language contains a set of patterns and shows different strategies to solve a more complex domain problem with its patterns. Moreover a pattern language shows the dependencies and cooperation of the included patterns.

The concept of patterns and pattern languages is not new. It was adapted from Christopher Alexanders architectural patterns. (see Alexander/Ishikawa/Silverstein, 1977) In the software development (esp. in the object-oriented software development) patterns were introduced in the early 90ies. The final break through of patterns in software development was made by the Gang-of-Four (GoF) in their book “Design Patterns”, which documents twenty-three design patterns. (see Gamma et al., 1995) Until now, many patterns and pattern languages were identified for different phases of software development.<sup>2</sup>

### 2.1 The Structure of Patterns

Each pattern and pattern language has its own structure – there is no universal accepted standard. The key point for authors is to describe patterns in a understandable fashion for the intended audience. In spite of that, Meszaros and Doble identified elements of a pattern description, which help authors to document the pattern and help readers to understand the problem and its solution. They structured these elements into mandatory elements, which should be used, and optional elements, which are helpful. (see Meszaros/Doble, 1996)

Mandatory elements include the *name* of the pattern, the *context* where the pattern was identified, the *problem* description, the *forces* which influence the solution and the *solution* itself. The *name* of the pattern should be an evocative expression of what the pattern is about. To provide a suitable name, meaningful methaphors (e.g. *Composite*) or a noun-phrase-name (e.g. *Two-Step-View*) could be used. Hence, the right name for a pattern is one of the most important thing for readers, because they should be able to remember the problem and its solution only by the name. The *context* section reveals the situation in which the problem is solved. Additionally, it determines the relative importance of the forces, which should be optimized. The *problem* is often documented as a question and should address the core of the matter. The reader should be able to understand the main issue of the pattern only by reading the problem section. The *forces* are mostly contradictionary considerations which must be taken into account to solve the problem. The *solution* proposes a way to solve the problem considering the restriction of the forces. Figure 1 shows the relationships

---

<sup>2</sup>An overview of pattern and pattern languages can be found at the website of the Hillside Group (<http://hillside.net/>), 30<sup>th</sup> August 2004.

between *context*, *forces*, *solution* and *problem*. In the scope of this article, pattern user might be the customer or the software developer.

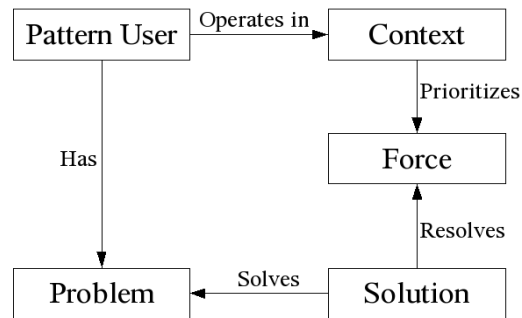


Fig. 1 Relationships Between Pattern Elements (Meszaros/Doble, 1996)

Optional elements can be used to give a more detailed description of the pattern and to refer to similar solutions. Optional elements could be *indications* (hints for readers to identify the symptoms that indicate that the problem exists), *related patterns*, the *resulting context* when the pattern was used, *(code) examples*, *aliases* (other names of the pattern), *rationales* (which explain why this solution is most appropriate for a problem within a context) and *acknowledgements*.

## 2.2 Anti Pattern

A special kind of patterns are Anti Patterns. (see Brown et al., 1998) An Anti Pattern describes a negative solution („worst-practice“) which reoccured in many practical fields and proposes a solution. These patterns can be identified in project- and software-reviews. Even though Brown's Anti Patterns are not the scope of this article three examples shall be given, because they belong to software development as well as the normal patterns (see Brown et al., 1998):

- *Analysis Paralysis* is a management Anti Pattern. It describes the striving for perfection and completeness, which leads to a project gridlock. One solution is an incremental software development process with clearly specified goals for each iteration.
- *Swiss Army Knife* is a software-architectural Anti Pattern and describes the over-design of interfaces. This leads to objects with numerous methods that try to anticipate every possible need. This makes the software very hard to understand and more difficult to maintain. A solution might be a clear structure of the software into packages and the exact definition of the functionality and responsibilities of each package.

- *Cut and Paste Programming*<sup>3</sup> is a development Anti Pattern and describes the process of copying methods and functions within one application. This leads to very hard maintainable and over-sized source code. A solution is to encapsulate this functionality properly, maybe in an extra-class.

### 3. Pattern in the Software Development Process

This section gives an overview of pattern languages and their patterns, which can be used in software development. This is not a complete list of all existing pattern languages, but it shows that there are recurring solutions in many phases of software development process.

Even though patterns are best-practice solutions, their usage does not automatically imply a good software quality. If a pattern does not apply to a special problem, it should not be used. Moreover patterns are not static solutions, they can be adapted to a special problem.

#### 3.1 Problem Definition and Requirements Analysis

The problem definition and requirements analysis are one of the first steps in the software development. The goal of this phase is to elaborate the functionality of the software system. To reach this goal, software developers have to understand the needs of a customer and capture the requirements for the software. In contradiction, customers are sometimes not able to express their needs clearly. RAPPeL and IBM's e-business pattern are two pattern languages, which address the issues of requirements analysis.

RAPPeL (Whitenack, 1995) stands for „Requirements-Analysis-Process Pattern Language“ especially for object-oriented software development. The language was designed to guide software-analysts and developers to appropriately apply a set of methods and techniques in requirements analysis and to understand a problem area. It provides a framework to define and capture requirements before and during the development process, which can also be used to evaluate, design, build, and test the software. Moreover the framework can be used to trace back the system functionality to the original business and system objectives. The patterns range from „*Managing and Meeting Customer Expectations*“, „*Defining the Requirements*“, „*Domain Analysis*“ to „*Requirements Validation*“ and „*Prototypes*“. All patterns provide deliverables (e.g. „*Requirements Specification*“, which is itself a pattern) in their solution, which supports the communication between customer and software developers.

While RAPPeL patterns focus on the methods and techniques of requirements analysis, IBM's e-business pattern pay more attention on the technical side of software development. (see Adams et al., 2002) The e-business patterns are structured into Business patterns, Integration patterns and Application patterns. Business and Integration patterns are on a higher level of abstraction and contain a set of Application patterns (see Figure 2).

---

<sup>3</sup>A better name of the pattern is „Copy and Paste Programming“. „Cut and Paste Programming“ indicates that the functionality is moved from one part of the application to another.

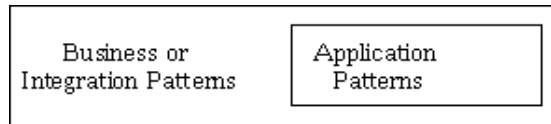


Fig. 2 Structure of E-Business Pattern (Adams et al., 2002)

Business patterns define the primary business purpose of the solution. They identify participants who interact with the software system and help to understand the interactions between the participants. The pattern language provides four business patterns (see Adams et al., 2002):

- *Self-Service* enables interested parties to interact directly with a business (e.g. an electronic shop is a Self-Service). The parties include customers, business partners, stakeholders, employees and all others who like to interact with a business. This pattern is also known as *User-To-Business* (U2B).
- *Collaboration* is also known as *User-To-User* (U2U) and addresses the interactions between users. This pattern can be identified in all applications where users work together to reach one goal (e.g. groupware systems).
- *Information Aggregation* enables the user of a software-system to get data from different sources (e.g. different databases). This pattern is also known as *User-To-Data* (U2D).
- *Extended Enterprise* connects existing applications even from other companies to the software. This pattern is also known as *Business-To-Business* (B2B).

Integration patterns help to compose Business patterns and to integrate existing software solutions in a company (e.g. databases or information systems). There are two directions of integration: The integration of users to different Business patterns (Front-End-Integration or *Access Integration*) and the integration of Business patterns to other applications (Back-End-Integration or *Application Integration*). (see Adams et al., 2002)

- *Access Integration* describes in particular designs that enable users to access Business patterns from multiple channels (e.g. different devices).
- *Application Integration* were often identified in web-based applications. The Application patterns of the *Application Integration* pattern can be mapped to Enterprise Application Integration (EAI).

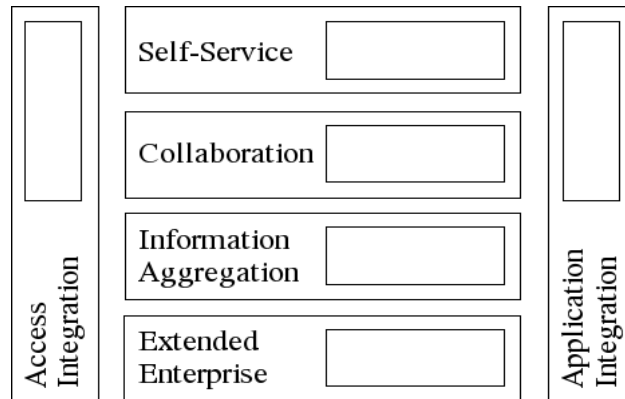


Fig. 3 Overview of Business and Integration Patterns  
(Adam. et al., 2002)

Figure 3 gives an overview of Business patterns and Integration patterns. It is possible to compose Business and Integration patterns in different ways (e.g. if *Self-Service*, *Information Aggregation*, and *Application Integration* are composed, an electronic commerce application can be realized).

Application patterns help to refine the Business and Integration patterns and therefore to make the shift from the problem and requirements phase to the software analysis phase.

### 3.2 Software Analysis and Specification

The goal of this development phase is to make coarse concept of the software. Developers have to decide which architecture should be used and which technical means apply to integrate other software systems. Additionally, a conceptual *domain model* of the software is created. The architectural decisions made in this phase are highly important, because all of them effect the design and implementation phases and the quality of the software directly – either in a good way or in a bad way. This might be a reason, why there are many patterns for this phase, which address different aspects.

Analysis Pattern (see Fowler, 1999)<sup>4</sup> describe recurring solutions in domain models of business. The pattern language provides “data-templates” that arise in numerous modeling situations. Fowler divided the view of software into two levels of abstraction: The knowledge level describes general rules, which can be used for more than one application. The operational level shows domain specific structures, within the general rules of the knowledge level. One example of this pattern language is the *Accountability* pattern, which reveals different views of organizational structures in a company. Employees of a company play different roles with various responsibilities. Fowlers intention is to provide different types of accountability on the knowledge level, which are composed to special kind of employee on the operational level. This architecture makes it easy to add and remove responsibilities to/from an employee without changing the source code of the software. Moreover this structure splits the complexity of the software design into two parts, which might lead to a better understanding of the problem.

<sup>4</sup>Some additional information and extensions about Analysis Pattern can be found on Martin Fowlers website (<http://www.martinfowler.com/>), 30<sup>th</sup> August 2004.

Application architecture patterns show fundamental concepts of the software. Each pattern structures a subsystem of the overall architecture and gives guidelines to customize these generic subsystems. Three examples are:

- *Lazy Load* (see Fowler, 2002) enables objects to load needed data separately. Sometimes the data volume for one object is very high, but is not needed for an operation. To prevent loading all data at once and therefore a bad performance, the *Lazy Load* pattern can be used.
- *Two-Step-View* (see Fowler, 2002) splits the view component into two parts. The First-Stage-View provides all data for a view in a raw format. The Second-Stage-View transforms this format into a special kind of presentation (e.g. HTML, WML or a normal application presentation). This pattern can be used for the *Access Integration* to support different devices (e.g. a mobile device or a PC) for one application.
- *MicroKernel* (see Buschmann et al., 1996) is used for software systems, which must be able to adapt in changing system requirements. Operating systems like Unix/Linux are popular *MicroKernel* implementations that manage minimal sets of core functionalities. Extended functionalities are included in pluggable modules.

Enterprise Integration patterns (see Hohpe/Woolf, 2004) focus on the integration of business applications via messaging. These patterns can be used to refine the *Application Integration* e-business pattern. The authors identified more than sixty patterns which occur in this field and additionally created a graphical syntax for their patterns. For example (see Hohpe/Woolf, 2004):

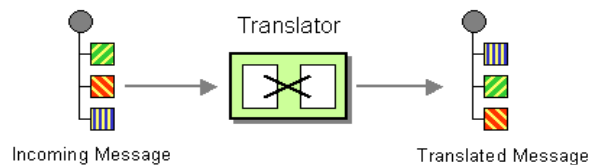


Fig. 4 Message Translator (Hohpe/Woolf, 2004)

- *Message Translator* (see Figure 4) translates a message from one format to another. With this pattern, it is possible that two different applications can communicate with each other, even if there are different data formats of the messages.
- *Canonical Data Model* can be developed to minimize the dependencies when applications with different message formats shall be integrated. Figure 5 shows an example: Both applications A and B communicate with applications C and D. Each application is able to translate its message format to the *Canonical Data Model* and reverse. Consequently only four translators must be written and all applications can communicate with each other. The worst case, that all applications have to communicate with each other, but no *Canonical Data Model* pattern is present, leads to six translators.

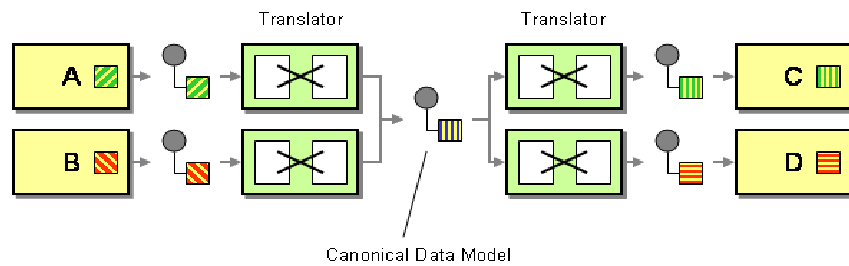


Fig. 5 Canonical Data Model (Hohpe/Woolf, 2004)

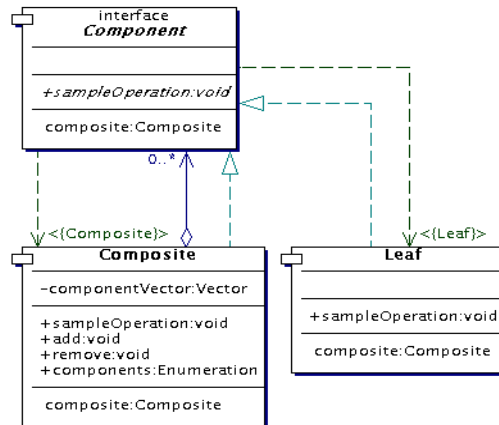
### 3.3 Software Design

The goal of the design phase is to realize the requirements and the technical specifications in an appropriate software model. All models of the Software Analysis phase are refined and completed. Developers are challenged to make a flexible design of a software, so that it is easily possible to add and remove features at a later moment. Furthermore, the software model should guide programmers, if they want to use a special functionality.

As mentioned before the Gang-of-Four wrote a book about Design Patterns. (see Gamma et al., 1995) Three examples shall give an insight of this pattern language:

- The *Singleton* is one of five creational patterns. Creational patterns hide the instantiation process of objects and therefore make a system independent of how objects are created., composed and represented. The *Singleton* ensures that only one instance of a class is present in the whole application. This pattern can be used to implement the core of a *MicroKernel*, which manages different modules. These modules inherently need the *MicroKernel* that they can work together. If module programmers need the *MicroKernel* functionality they are implicitly guided by the *Singleton* to get only the existing instance and prevent initializing another *MicroKernel*.

- The *Composite* pattern is one of seven structural patterns. Structural patterns describe how objects are composed to create a larger structure. The *Composite*



composes objects to a tree structure and represents part-whole hierarchies. Figure 6 shows an UML class diagram of the *Composite* pattern. *Composite* objects can contain *Components*, which are either type of *Leaf* or again *Composite*. A practical example of the *Composite* can be found in the Java™ Abstract Window Toolkit (AWT) architecture. The classes `java.awt.Component`

Fig. 6 The *Composite* pattern as UML class diagram

and `java.awt.Container` form the fundamental concept of the *Composite* pattern. Classes like `java.awt.Button` or `java.awt.Checkbox` are leaves.

- The *Command* pattern is one of eleven behavioral patterns. Behavioral patterns focus on algorithms and assign responsibilities between objects. The *Command* pattern encapsulates requests, which are initiated by clients. It can be used in the *Two-Step-View* pattern, where different clients are connected to the same application with different views. The requested functionality of the application may not depend from a special kind of client. So all clients may be able to call the same function.

### 3.4 Implementation

Patterns at the implementation phase of the software-development process can be identified, as well. They are called idioms. Idioms are patterns which are specialized to a specific programming language on a low abstraction level. They describe, how problems of a component or their relationships between them can be implemented with methods of special programming language. Most idioms are language-specific and reflect implementation experiences. An example is the *Counted-Pointer-Idiom* which eases the memory-management of objects in C++. (see Buschmann et al., 1998) Other examples can also be found in some technology developments like J2EE. J2EE-Patterns describe how to solve problems in the field of applications within J2EE-based application-servers. Here, idioms can be found for the presentation-, client-, business-tier and patterns for the integration-layer, as well. For example to build dynamic web-presentation, *JavaServerPages* or *Servlets* can be used. (see Bien, 2002)

### 3.5 Test

The goal of the test phase is to assure that the developed software fulfills all requirements. The “System Test Pattern Language” (see Delano/Rising, 2004) addresses especially system testers and shows some recurring problems in software testing. The pattern language is structured into four parts. The Test Organization focuses on the relationship of system testers to the rest of the organization. One revealed pattern is *Designers are our friends* and addresses the collaboration between designers and testers. The Testing Efficiency part concentrates on finding errors in a software very early. One pattern in this part is “*Unchanged*” *Interfaces*, which reveals the importance of proper defined interfaces in a very early stage of software development. Part three, Testing Strategy, shows patterns for software test environments. The *Busy System* pattern illustrates the problem, when a software runs on a system, which resources are nearly exhausted. The last part, Problem Resolution, contains patterns that help to communicate discovered errors. *Document The Problem* is a pattern that reveals some common problems on how to document the problem and how programmers and testers should behave.

### 3.6 Documentation

The patterns in the software-development process are used to solve and therefore document some aspects of problem domains. Hence all patterns are useful for documenting the software, as well. But furthermore some patterns related to documentation itself can also be found. They define on the one hand what kind of documentation can be used, like *FAQsDocumentation* or *SelfDocumentingCode* and on the other hand they define how to document, like *DocumentLast* or *Documents-FirstCodeLater*. (see Cunningham, 2004)

Moreover some programming-languages provide a set of rules how to document the source code. This patterns can be seen as documentaion-idioms, as well. For example the Java programming language provides tools to generate a Application Programming Interface out of source code comments. (see Sun, 2004)

Additionally, a general description was found on the internet. The university of Essen in Germany offers a so called „Schreibwerkstatt“ (writer workshop), which describes the kinds of available texts, like articles, reports, academic papers and explains how to write such texts, as well, like how to make an investigation or how to build an argumentation structure. (see University of Essen, 2004)

## 4. Summarize and Conclusion

This article gave a short overview about patterns and pattern languages in different phases of software development. Various patterns out of different pattern languages have been shown. Each of these patterns addresses one special issue in software development and can be reused in similar situations of modeling and building business applications. The simple structure of patterns with name, problem, solution, context, and forces leads readers, pattern users and pattern writers to an efficient way of communication.

Due to the fact, that there are patterns in different levels of abstraction and that these patterns can be combined, it is possible to develop a software system with patterns continuously. If patterns are used correctly, it is possible to build flexible business applications. Moreover there are patterns and pattern languages which guide customers and software developers to do the things right.

## Bibliography

- Adams, J., Koushik, S., Vasudeva, G., and Gambalos, G. (2002): Patterns for E-Business: A Strategy for Reuse. IBM Press.
- Alexander, C., Ishikawa, S., Silverstein, M. (1977): A Pattern Language: Towns, Buildings, Construction. Oxford University Press.
- Bien, A. (2002): J2EE Patterns. Entwurfsmuster für die J2EE, Addison-Wesley
- Brown, W.H. (et al.) (1998): Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1998): Pattern-orientierte Softwarearchitektur – Ein Pattern-System, Addison-Wesley, originally published by John Wiley & Sons, Ltd. 1996 under: Pattern-Oriented Software Architecture. A System of Patterns
- Cunningham & Cunningham, Inc (2004): internet-site: <http://c2.com/cgi/wiki?DocumentationPatterns>, Date: 31.08.2004
- Delano, D. and Rising, L. (2004): internet-site: <http://www.agcs.com/supportv2/techpapers/patterns/papers/systestp.htm>, Date: 31.08.2004
- Dumke, R. (2001): Software-Engineering. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden; 3rd Edition.
- Fowler, M. (1999): Analysis Pattern. Addison-Wesley.
- Fowler, M. (2002): Patterns of Enterprise Application Architecture. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): Design Patterns: Elements of reusable Object oriented Software. Addison-Wesley.
- Hohpe, G., Woolf, B. (2004): Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.
- Meszaros, G., Doble, J.(1996): Metapattern: A Pattern Language for Pattern Writing. The 3rd Pattern Language of Program Congress. Monticello, Illinois.
- Sun Microsystems (2004): internet-site: <http://java.sun.com/j2se/javadoc/index.jsp>, Date: 31.08.2004
- University of Essen (2004): internet-site: <http://www.uni-essen.de/schreibwerkstatt/trainer/>, Date: 31.08.2004
- Whithenack, B. (1995): RAPPeL: A Requirements-Analysis-Process Pattern Language for Object-Oriented Software Development. In: Pattern Languages of Program Design, pp. 259—291, Addison-Wesley.