Using Patterns for Design and Documentation

Georg Odenthal and Klaus Quibeldey-Cirkel

Department of Electrical Engineering and Computer Science
University of Siegen, D-57068 Siegen, Germany
{odenthal | quibeldey}@ti.et-inf.uni-siegen.de

Abstract: The dovetailing of design and documentation is characteristic for many mature engineering disciplines. In electrical engineering, for example, a circuit diagram is a means and technique for both designing and documenting. Software engineering falls short in this respect, especially when it comes to *architectural* issues. Design patterns can help here. Using both *form* and *content* of design patterns promotes the principle of *documenting by designing*. Our experience report presents some examples of this principle taken from an evaluation project at SAP, Germany.

Keywords: object-oriented design patterns, pattern form, software documentation

Overview: In Section 1, we clarify what we mean by the principle of documenting by designing. In Section 2, we outline the aim of the evaluation project. We then generalize the steps of *instantiating* a pattern to solve a particular design problem and of *identifying* a pattern candidate in a given design to gain some flexibility. To demonstrate these pattern-related *activities*, we discuss two examples in more detail. Section 3 enlarges on documentation using patterns. We discuss the central role of hypertext and give two forms (templates): one for the documentation of pattern instances, and an extension for documenting frameworks. In Section 4, we summarize our experiences in using patterns for design and documentation contrasting them to experiences from literature. We conclude our report with some remarks on further research activities.

---

1 Introduction

Design patterns are generally welcome for their pragmatism:

- capture and reuse of design expertise and experience [1, 9, 10, 12, 13, 15],
- design and documentation of frameworks [3, 14, 16, 17, 22, 24],
- economy and clarity of expression [4, 27].

In this experience report, we stress the *dual* nature of the pattern approach: it is both *generative* and *descriptive*.[1] Kent Beck and Ralph Johnson give a reliable definition [3]:

> *Alexander's patterns are both a description of a recurring pattern of architectural elements and a rule for how and when to create that pattern. [...] We call patterns like Alexander's that describe when a pattern should be applied 'generative patterns'.*

---

[1] See the mailing discussion on "Generative vs. Descriptive Patterns" and "Designs Documented as Patterns?", the latter initiated by Robert S. Hanmer:
http://iamwww.unibe.ch/~fcglib/WWW/OnlineDoku/archive/DesignPatterns/1171.html

In ontological terms, the attribute *generative* refers to a pattern's *content*, that is the recurring thing itself (classes constituting a micro-architecture). In epistemological terms, *descriptive* refers to a pattern's *form*, that is the way we capture and articulate this thing (by formats like problem-context-forces-solution). It is the dual function of design patterns, the interplay between form and content, that we experienced worthwhile in the context of documenting software.

1.1 The Guiding Principle of "Documenting by Designing"

In mature engineering disciplines, the design of an artefact is dovetailed into its documentation, and vice versa. Architects and electrical engineers, for example, get a great deal of their product documents *in passing*, that is as a by-product of the design process. The main reason for this is that circuit diagrams or blueprints describe *material* artefacts, i.e. hardware such as buildings and circuits. Their structures can be easily made explicit as geometric models or schematic diagrams. The mode of designing the product is the mode of documenting it. On the other hand, software engineers have to struggle with *immaterial* constructs, i.e. data structures and algorithms. The *essence* of software complexity, as Frederick Brooks has coined it [6], lies in the mixture of data structures, algorithms, and function calls.

Although today's software engineers have far more expressive constructs at their disposal, such as inheritance and polymorphism, they still lack architectural constructs with clear semantics. *Categories* [5], *Subjects* [11], or *Clusters* [20] aim at grouping semantically linked classes. However, these terms are used notationally. They are of little value as architectural vehicles for they function only as *ad hoc* containers: What is put into them is at the will of the designer. It is this lack of architectural constructs that makes class libraries, and especially frameworks, so hard to design and difficult to comprehend and maintain (see Fig. 1).

Obviously, it was the deficit of conceptual structures beyond the boundary of individual classes that has initiated the search for an "Architecture Handbook" of software engineering; first articulated at OOPSLA '91. And it was the framework context, where design patterns were first identified for documentation purposes (see Erich Gamma's thesis [14]). The GoF ("Gang of Four") authors of the best-selling standard pattern text [15] have gained their expertise in the development and documentation of well-known frameworks (ET++, UniDraw, and InterViews). Design patterns are both a means and technique to design and document *micro-architectures* that can be easily identified and reused. By establishing a further level of abstraction, patterns can reduce the *accidental* complexity (again, in Brooks' terms) of system description. This has been regarded as a further indirection, and therefore as a disadvantage. With documentation, however, the advantages overwhelm. Documents are living products that should be allowed to evolve *together* with the iterative and incremental design cycle. Using patterns in both ways – generative and descriptive – promotes this principle of documenting by designing.

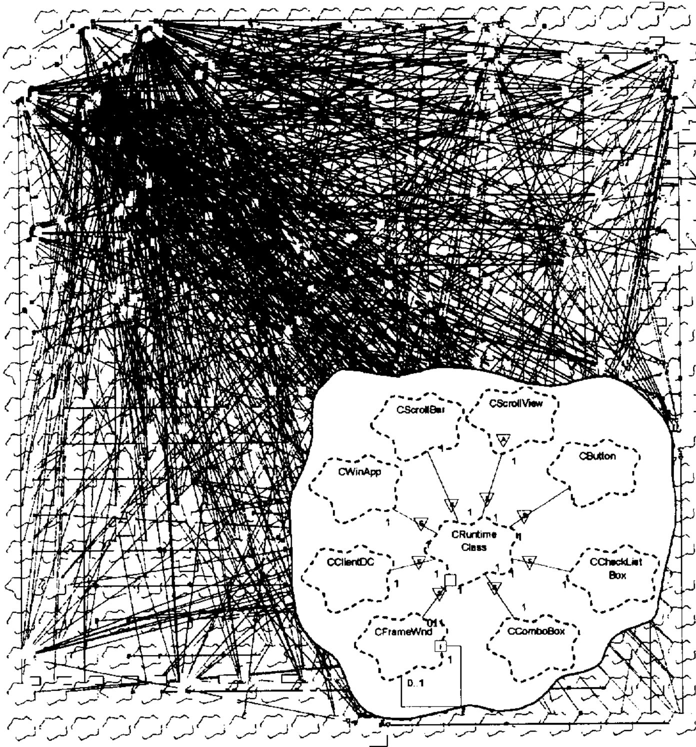

Fig. 1. In need of architectural information and guidance[2]

1.2 The Interplay between Form and Content

If we relate the pattern form to software documentation, we should be ready to give some pragmatic answers to "What constitutes *good* software documentation?" To sum up the literature [5, 20] and one's experiences [23], characteristic of good software documentation is a sound *mix* of formal and informal means of description: graphic versus textual notations; natural versus formal languages. As an empirical fact, most designers are reluctant to rigid formalisms of description. On the other hand, however, documenting a design in natural language has psychological constraints of its own – the "white paper" barrier, for example. Additionally, many designers associate bureaucratic activities with documenting. Moreover, for documentation in a natural language there is only little systematic CASE support; mostly restricted to fragmented and distributed annotations of individual design components. In contrast to traditional means of documentation (use cases, CRC cards, class diagrams), the pattern *form* as a natural prose style is systematic, disciplined, cohesive, and more comprehensible. Re-

[2] What is shown here resulted from reverse engineering the MFC framework into a Booch diagram. Zooming into this mass of classes does not reveal any architectural structure that could be easily recognized, or directly used for documentation.

garding prose style, Alexander's and the Portland pattern forms are narrative, while the GoF form is more structured. The latter seems to better reflect the engineer's writing mentality. Besides, documents structured in the GoF form can be directly supported by hypertext techniques and retrieval systems. In contrast to paper documents, a fine-grained and consistent information system is feasible.

Form and content of a pattern stimulate design and documentation *in concert*. Its content motivates documenting by its very nature: it helps the designer to reflect on his decision. Either by way of confirmation or by contrast, he will document his current design. Let us elaborate on this point: The GoF pattern form comprises more than a dozen sections such as Applicability, Consequences, and Implementation. In the process of instantiating a pattern of this form in your design, you will refer to the original pattern description. Documenting the resulting class structure, you will, *again*, refer to the pattern description. In doing so, you self-critically validate, justify, or dismiss your design decision. Hence, documenting your design *rationale* in the pattern form will make you reconsider its validity. Take, for example, the Applicability section: Do you recognize the situations described there? Does your design fit to the context? Or take the Consequences section: What are the trade-offs and results of instantiating the pattern in your design context? What aspects of your system structure does the pattern instance let you vary independently? Or the Implementation section: What pitfalls, hints, or language-specific issues should a maintenance programmer reading your document be aware of? Thus, besides the pattern form, the pattern's content, too, will help you produce the design documents. Finally, your growing experience in using a certain pattern will feed back to the original pattern itself (we will discuss this in Section 3.1 on the role of hypertext). It is the *dual* nature of a pattern – generative and descriptive – that lets form follow content, and vice versa. Documenting with the pattern form helps justify one's design rationale to oneself and to others, and might reveal design alternatives otherwise not taken into account.

1.3 Pattern Activities

How are design patterns related to the software development process? According to our approach of documenting by designing, we can differentiate and localize the following main activities of applying design patterns to the development cycle (Fig. 2):

- **Pattern Instantiation**. This is *standard* pattern practice: choosing a design pattern to *generate* parts of a design (a process driven by analysis efforts).
- **Pattern Candidate Identification**. This is *non-standard* pattern practice: locating a given class structure to insert a design pattern instance for *flexibility* reasons (a process initiated by design walkthroughs, code inspections, or changed requirements preparing the ground for a framework development).

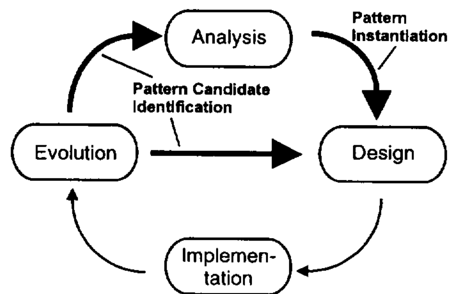

Fig. 2. Pattern-related activities

A third pattern activity, not argued in this report, is loosely related to design but closely related to documentation: Mature designs can be made more reusable and self-explaining by documenting the patterns used. The popular InterViews framework [19] for example, would be a worthwhile candidate for this activity of *reverse* documenting. It was designed with design patterns in mind[3], their use and the resulting pattern instances, however, have never been annotated in the documents (as far as we know). In the following section, we will discuss the first two activities in the context of non-trivial design examples.

2 Pattern-Oriented Design

In standard pattern texts, the authors unanimously point out that patterns alone do not constitute a design *method.* Patterns are (mental) *building blocks* that support the designer in certain phases of the software development cycle. They can, however, put some of the decision processes in a concrete form, which would otherwise remain vague and without guidance. In the following, we will demonstrate this guidance by examples. We emphasize the less typical pattern practice, i.e. pattern candidate identification. Additionally, we argue that reducing a pattern's original flexibility can be opportune in a particular design context.

2.1 The Evaluation Project

The aim of the project was to develop an object-oriented interface to interoperate between the SAP-R/3 Business Object Repository (BOR) and the Open Scripting Architecture (OSA) from Apple/IBM. OSA is comparable to Microsoft OLE Automation. The Business Object Repository is the managing unit for Business Objects, which are mainly used by the SAP Business Workflow. Business Objects allow an object-oriented access to and modification of R/3 data. With the Business Object Repository, a client can make up object types from data fields and ABAP functions (corresponding to the attributes and operations of a class). It was agreed from the beginning of the project to apply patterns of the GoF form for both design and documentation. Fig. 3 illustrates the components designed in the project. Two of them, i.e. "Storing Business Object Types" and "Process Control", will be taken as examples in Sections 2.3 and 2.4.

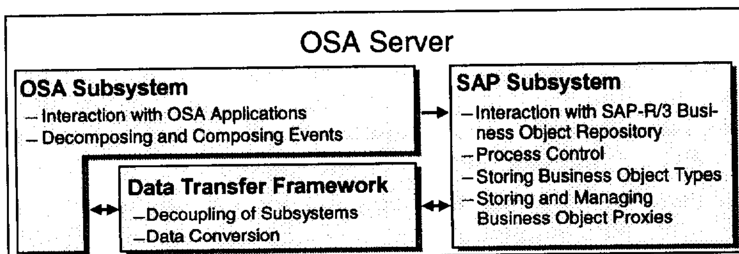

Fig. 3. Components of the OSA server

[3] This framework is often mentioned in the Known Uses section of the GoF form [15].

2.2 Steps for Instantiating and Identifying Patterns

In the previous section, we have differentiated between two main pattern activities: (1) designing with patterns and (2) making a design more flexible through patterns. The first one is standard practice, generally called "instantiating a pattern". The second one is *non*-standard practice that we call "identifying a pattern candidate" in a given class structure. Both activities can be divided into four major steps. While the first two steps refer to *decision-making problems*, the latter two refer to *structural changes.* Nota bene: The steps are not meant to represent a design method – our intent is to make the cognitive and technical processes of using patterns explicit.

Step 1: Searching and Choosing. The designer looks for a suitable design pattern in a pattern catalogue, or tries to identify a candidate for a pattern in a given design. Instantiating a pattern takes place in the initial phase of a design, while identifying a candidate follows first experiences with a prototype or working system. For example, some features of a design component have proved to be insufficient and shall be made more flexible by incorporating a pattern into the class structure. Both activities presuppose a thorough knowledge of patterns. The more patterns a designer knows, the more he will cover his design by instances of patterns; or in the case of a given design, the more likely he will find a pattern that matches a problem identified in the class structure. Generally, there will be several alternatives, so that the process of choosing or identifying implies a decision-making problem. The *Forces* section of a design pattern may function here as a first guidance.

Step 2: Planning and Allocating. The process of instantiating a pattern poses the following questions: What are the pattern's classes called in the problem domain? What additional responsibilities must be assigned to the pattern's classes? Generally, the assignment will be *complete* since there is no fixed design to be taken into account. In the case of identifying a pattern candidate, the questions are: Which roles do the given classes, operations, and attributes play in the pattern? Do they all play a role? In this case, the assignment will often be *incomplete* since some classes, operations, or attributes do not fit. This may lead to the question: Does the pattern really match at all? The problems encountered should not be solved in this step, just properly documented. The documentation of this planning stage will guide the next steps of structural changes.

Step 3: Fitting. Instantiating a design pattern might involve changes to its original structure. Sometimes, it can be appropriate to reduce the original flexibility of a pattern by changing or dismissing classes (we give an example below). With the help of a proper documentation of this pattern instance, the original flexibility can be restored, should requirements change. In the case of inserting a pattern in a given design, it will often be necessary to change a class' interface or to add new classes. These are likely the pattern's abstract classes that carry the desired flexibility.

Step 4: Elaborating. Finally, technical classes complete the design. They won't add any further semantics but separate the concerns of the problem domain from technical issues and coding directives modelled so far in domain classes. Examples are container classes like lists or sets from a class library. Additionally, an extension to a class' interface might be necessary, e.g. to add dynamic type checking. It is up to the designer to state the completion criteria.

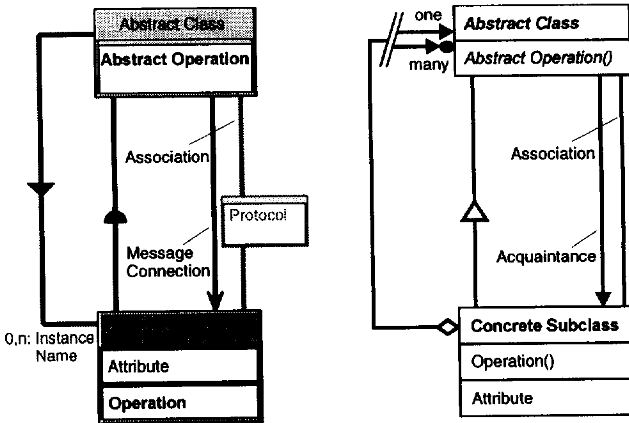

Fig. 4. Coad's notation used in *objectiF*® vs. OMT used in the GoF book

Some remarks on the graphic notations: We have used a mix of Coad's notation of class diagrams [11] and the OMT-based notation as used in the GoF book [15]. Handling different notations has two reasons:[4] First, the OMT notation of steps 1 and 2 is used to keep the original association of the GoF pattern alive. This facilitates communication among designers: both pattern and field experts are able to judge the design effort. Our extended OMT notation of step 2 also reflects the *interfacing* quality of the diagram. The concepts of both domains are annotated to the classes, separated by a colon (*pattern* concept : *field* concept). If an assignment is unclear or impossible, a question mark is annotated. Elements not used at all are crossed out. The diagrams of these steps can be freehand or drawn with the help of some semantic graphic editor. The second reason for using different notations is to make the transition from planning (steps 1 and 2) to developing (steps 3 and 4) explicitly clear. In the latter steps, a development tool with a notation of its own is involved (in our examples, this is Coad's notation as used in *objectiF*® V 1.1 from microTOOL, Berlin, see Fig. 4).

Annotating a pattern's instance is still an issue of debate. We have chosen the "Pattern:Role" labelling of Gamma assigning the pattern's name and its role names to the corresponding classes. A Venn diagram, however, is more suitable when the interplay between several instances of different patterns is to be illustrated (for a first impression, see Fig. 11). We argue against the current UML proposal (V.0.9) to annotate a design pattern as a Jacobson's Use Case, as this would lead to overloading both the term and the graphic design representation.

2.3 Example of Instantiating a Pattern: "Storing Business Object Types"

Context: The object type (class) of a SAP Business Object is defined and maintained in the Business Object Repository. The interface of such a class comprises a list of attributes and operations. An operation, on the other hand, comprises a list of parame-

---

[4] Admittedly, the pragmatic reason is sort of *willy-nilly*: At the time of the project, there were three different notations, i.e. of Booch, Coad, and Rumbaugh, spread over the CASE tools we had at our disposal.

ters. The *problem* is to develop a unit maintaining and storing the classes' interfaces. *Solution*: Regarding the hierarchical structure of the data, the *Composite* GoF pattern is a likely choice. An intuitive instantiation of the pattern is shown in Fig. 5; for the original GoF structure see Fig. 6, step 1.

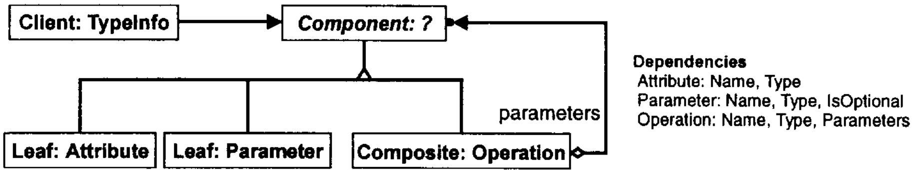

Fig. 5. A first try with the *Composite* pattern

Forces: Considering the dependencies of the context, we modified our first try (see step 2 of Fig. 6): a separation between attributes and parameters is not necessary. Being convinced that the *Composite* pattern will work, we transferred it into the design (step 3). Further adapting the structure of the pattern, we realized that we could do without a *Leaf* class at all. Hence, we changed the *Component* class from abstract to concrete. By that, we somehow reduced the original flexibility of the *Composite* pattern for reasons of simplicity. However, if requirements will change and we wish to restore the pattern to its full potential, we can simply achieve this by inserting *Leaf*

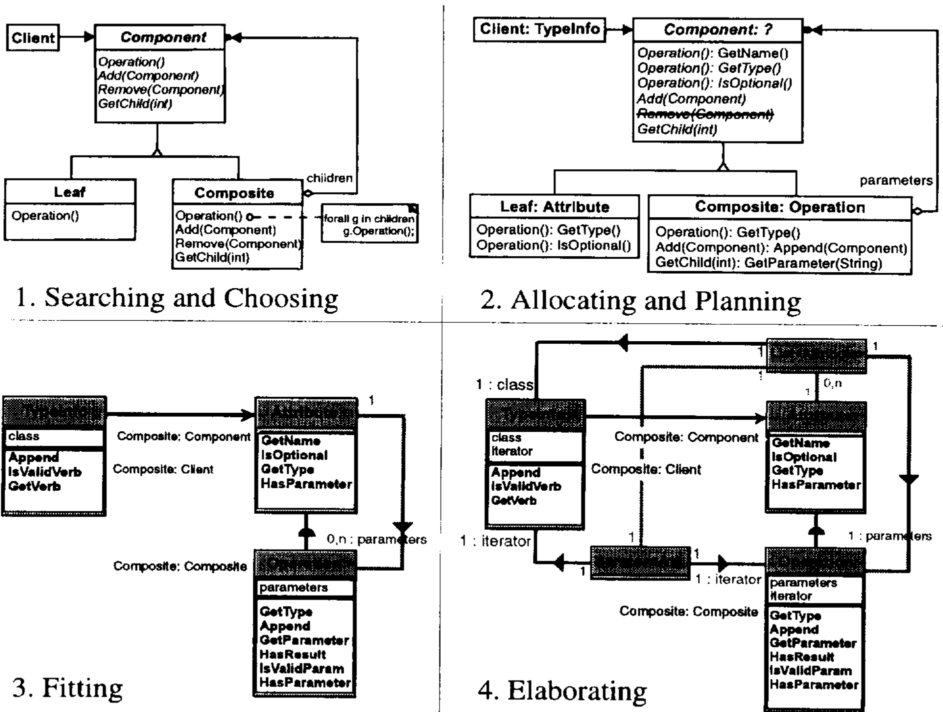

Fig. 6. Steps of instantiating the *Composite* pattern

classes. This presupposes, of course, that the pattern instance is clearly annotated and documented. Finally, the resulting design has to be elaborated upon to make it work. Thus, the technical task of storing parameters is delegated to class templates, i.e. *List* and *Iterator*, taken from a standard library (step 4).

2.4 Example of Identifying a Pattern Candidate: "Process Control"

Decoupling was an important goal in the development of the OSA server. The subsystems to be designed for interacting with SAP R/3 and OSA were to be strictly separated from one another. The aim was to guarantee easy substitution of the interoperability interface. Decoupling was to be achieved for the flow of both data and control. In this example, we will concentrate on the control flow, i.e. the reaction to an OSA event. Fig. 7 (upper section) depicts the design fraction representing the general problem: How is the control flow maintained between the receiver of an event (OSA-Dispatch) and the class representing the BOR component of the SAP system? For an orientation, we bring the motives behind the requirements to the fore:

- Early feedback from rapid prototyping: decoupling of subsystems; process control between OSADispatch and BOR.
- A postponed requirement from analysis: The SAP subsystem represented by the BOR class is considered not to be changed. OSADispatch should be easily substitutable by another interoperability interface, e.g. OLE.
- An extension to prior requirements from analysis: one-to-many relationship between OSADispatch and BOR so that several SAP systems can be addressed simultaneously by a single OSA event.

In short, these requirements aim at improving the quality of the design by inserting additional *flexibility*. With this aim in mind, there are initially several design patterns at our disposal:

- *Facade* encapsulates a subsystem and defines a generalized interface to make the subsystem easier to handle.
- *Adapter* converts the interface of a class into something a client expects.
- *Chain of Responsibility* chains several receiving objects to one sending object. A request is passed along the chain of receivers until an object handles it.
- *Observer* defines a one-to-many dependency between a "Subject" and one or more "Observers" ensuring that all Observers are notified when the Subject changes state.

Choosing a pattern is the most important step as (a) all subsequent changes to the given design and (b) the quality of result will depend on it. To make a choice, the designer has to rely on his knowledge of design patterns and his understanding of the requirements of the problem. A deep understanding of a pattern's potential, its *essence*, can only be gained through practising the pattern several times. The lower section of Fig. 7 represents the essence of the *Observer* pattern:

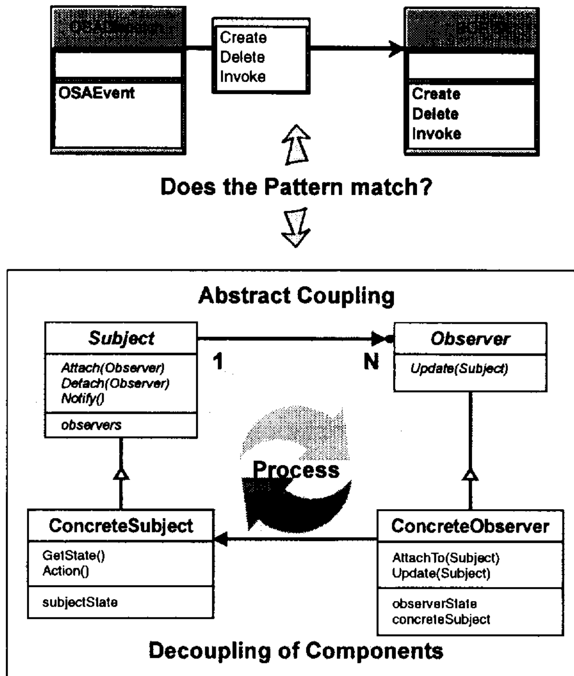

Fig. 7. Step 1: In search of the right match

- Dividing the components' interactions into two parts: an unchangeable abstract coupling (upper part) and the specific request from a ConcreteObserver to a ConcreteSubject (lower part).
- One-to-many relationship between a subject and its observers
- Intuitively, the proceeding is clockwise:
  AttachTo() → Action() → Notify() → Update() → GetState().

Some remarks on our decision-making: *Facade* and *Adapter* are not suitable in our context as they are primarily structural patterns. The behavioural aspect, i.e. the control flow between OSADispatch and BOR, is paramount. *Chain of Responsibility* does not fit either: The essence of this pattern lies in the ability of objects to pass a request along their class hierarchy. Incorporating a subsystem's classes into an inheritance relationship that is semantically not justifiable – just to insert a pattern in the design – makes little sense. Finally, the *Observer* pattern seems most suitable. Its transfer into the design will be discussed below.

We first examine the existing design (upper section of Fig. 7): After receiving an event, the OSADispatch class passes the request to BOR's interface. Thus, BOR plays the role of a server, while OSADispatch is the client. This approach turns out to be disadvantageous as the responsibility lies with OSADispatch concerning control of communication and the selection among several SAP systems. Additionally, a client component is the easier to replace the leaner it is. By changing the roles of client and

server, the design becomes more flexible and is closer to the framework idea: "don't call us, we'll call you". Thus, the flow of communication within the OSA-Server will be reversed (see Fig. 10). OSADispatch now functions as a server. Having received an event, it only forwards a message of notification (Notify()) saying that something has changed. All SAP subsystems attached to Subject get an update message and can individually decide which one is meant. Afterwards, additional information will be requested from OSADispatch (GetCommand()).

In step 2 (see Fig. 8), it is obvious that the whole interface of the BOR class, i.e. Create(), Delete(), and Invoke(), could not be allocated to the pattern's functionality. As a consequence, we changed BOR's interface as shown in Fig. 9. Now, BOR's previous functions cannot be called any longer by a client. They have become protected member functions and are called via Update(). Thus, being notified, BOR itself will take control of this function. Finally, in step 4 we have inserted technical classes, such as List<Observer> and Iterator<Observer>.

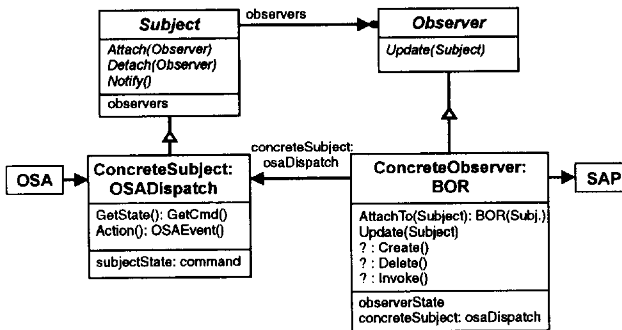

Fig. 8. Step 2: Allocating and Planning

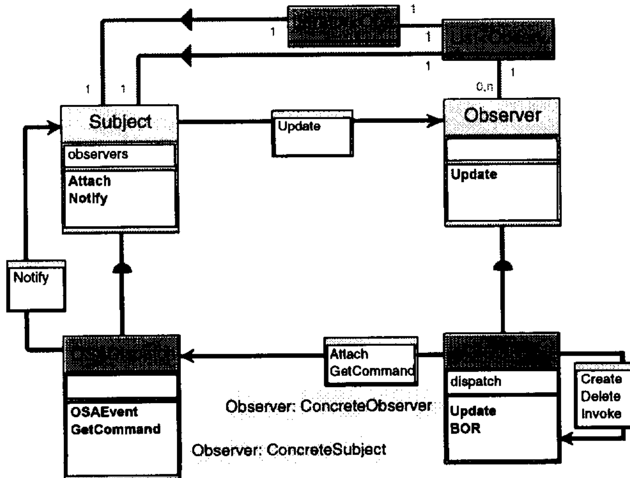

Fig. 9. Steps 3 and 4: Fitting and Elaborating

Taking this approach, the responsibility of process control is with the SAP subsystem that has been considered fixed. Hence, as a by-product, the one-to-many relationship has come along by inserting the *Observer* pattern.

To fulfil the requirement of making OSADispatch easily replaceable, the instantiation of the *Observer* pattern would have to be realized as a framework. For this, an abstract message dispatcher (AbstractDispatch) would be inserted to define a protocol of communication between BOR and AbstractDispatch. Before that, communication was handled by the operation GetCommand(). With communication getting more complex, GetCommand() would gain a symbolic meaning. A concrete dispatcher like OSADispatch would be derived from Subject and AbstractDispatch. For a translation of the communication protocol, the *Adapter* pattern could be applied. The steps described here clearly indicate that the original approach has been made more flexible and that a shift of responsibility has taken place: a smooth transition to framework development is mapped out (Fig. 10).

Fig. 10. Change of roles: Seamless transition to framework design

3 Pattern-Oriented Documentation

With software becoming a capital stock for many companies, two problems aggravate:

- How can a company's design knowledge and experience be preserved in the face of fluctuation (braindrain)?
- And how can we effectively integrate a newcomer into a design team, i.e. how can the learning process be shortened?

It is well known that these problems are closely related to widespread negligence of proper software documentation. In the following, we outline our approach to using design patterns for documentation. The aim is to document designs for better understanding and for identifying, evolving, and applying *reusable* components. From a designer's point of view, what is most needed for documenting a complex object-oriented design is an abstract layer right above the class level (see Figures 11 and 12). The components at this *Pattern level* meet a major requirement of design reuse: they are large enough to make reuse economic, and small enough to stay in the realm of a designer's concerns. In addition, with a slight modification of the templates introduced in Sections 3.2 and 3.3, *domain components* can be documented as well. In this case, the annotation of patterns and domain components often will overlap.

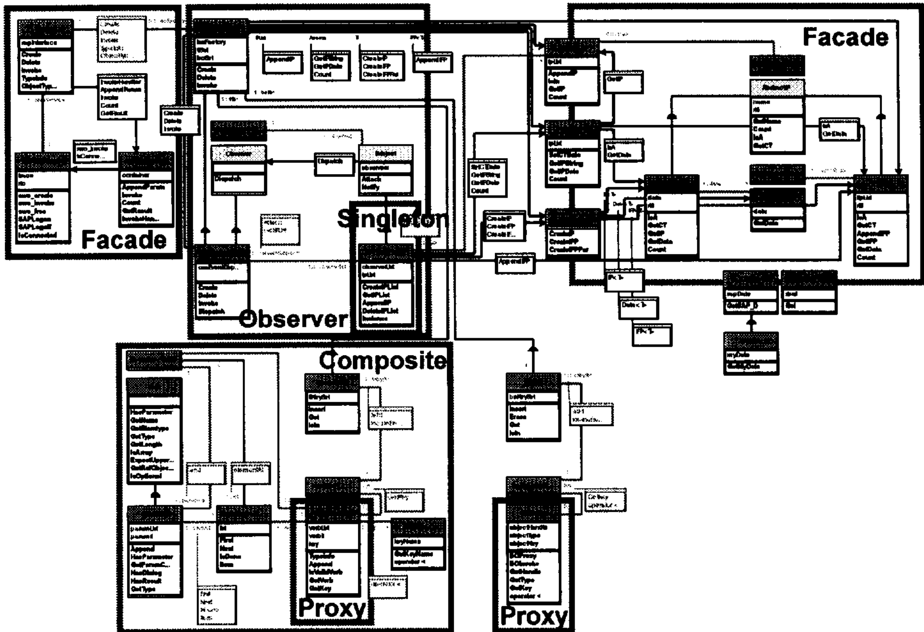

Fig. 11. Reducing descriptive complexity by covering the design with pattern instances[5]

Our pattern-oriented approach to documentation concentrates on the *reflective* use of (on-line) pattern texts, be they proprietary or standard. It does not aim at finding and describing new design patterns. Pattern-oriented documenting logically continues pattern-oriented designing: Covering a design partly by pattern instances (see Fig. 11, which shows the project's overall class structure), these instances *simultaneously* lay the foundation for documenting the design. They structure the system under development at a higher level of abstraction. Hence, the design is structured independently of the problem domain, establishing a *meta*-level documentation: For those designers who are not familiar with the problem domain, but familiar with design patterns, there is a *neutral* access to understanding the system. Pattern-oriented documentation supplies a *link* between the general description of design patterns and their instances in the problem domain: This link documents why, in which context, and how a design pattern has been instantiated – that is the *rationale* of a design decision. The most suitable technique for a pattern-oriented approach of documenting is *hypertext*. We will discuss the tool aspect below.

[5] Compare with Fig. 1: Imposing a Pattern level there could break the complexity barrier of understanding. Unfortunately, as far as we know, patterns were not used for the design of the MFC framework.

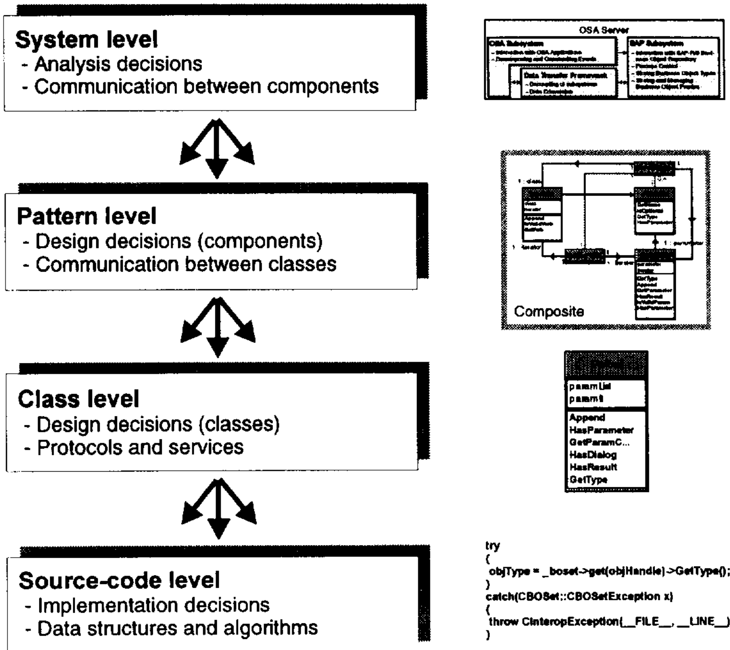

Fig. 12. Levels of system description: A hierarchical hypertext

3.1 The Role of Hypertext

Recapitulating the interplay of a pattern's form and content, what is needed for an efficient and comfortable medium to quickly switch from one aspect to another is hypertext.[6] Fig. 12 illustrates the description levels we differentiated in our project. Both aspects of system description, design and documentation, are accessible by hypertext techniques. With the help of hypertext, the items of interest can be made *sensitive* for design navigation, searching, filtering, and modification.

Consider this: The description of standard design patterns is available as hypertext. You are in the process of instantiating a design pattern. The hypertext documentation system will support you directly: It will generate a documentation frame to function as a *cardinal point* for the emerging texts, graphics, even source code. While you are instantiating design patterns or identifying pattern candidates – being involved in the steps of searching and choosing, allocating and planning, fitting, and elaborating (see Section 2.2) – the documentation *evolves*. Moreover, the content of a design pattern may evolve, too. Take, for example, structural changes that you have found appropriate and worthwhile in your current project. You probably will document these changes in the Consequences, Applicability, or Implementation sections of the original pat-

[6] Authors of standard pattern texts have emphasized this point before [9, 15, 22], but mainly with regard to an on-line version of a pattern catalogue.

tern's description. In fact, a *new* pattern might evolve. A pattern-driven design will eventually lead the designer to *unknown* patterns of his problem domain, particularly when his repertoire of patterns proves to be insufficient. If the items of design and documentation are captured as a hypertext, some kind of *yo-yo* access is supported: You can dive into the highest system description, inspect some component's pattern description by penetrating its class structure as deeply as to code fragments, and come up to the system level again taking a changed perspective of the design under development, and so forth.

For lack of space and because a linearized hypertext does not disclose its potential, we would like to direct the reader's interest to the on-line *WinHelp* version of the project's hypertext documentation.[7]

3.2 Template for Documenting Pattern Instances

In the following, we concentrate on the document's structure at the Pattern level (see Fig. 12). For our project, we found the following hypertext template appropriate for documenting pattern instances. What is most important, is the *consequent* use of the same template at a certain level of description. This will enhance the reader's familiarity with the documentation as a whole.

Overview. Give a reference to a class diagram or produce some other kind of visualization for a first orientation in the design context.

Intent. State the reason why you have instantiated just this design pattern.

Motivation. Describe the design context in more detail. Give a survey on the design component. It is very useful to illustrate the design actions that have led to the actual instantiation of the pattern (as shown in the examples of Sections 2.3 and 2.4). If possible, state the references to documents from the analysis phase, e.g. the relationship to analysis patterns of the problem domain [13].

Roles. Label the classes of the pattern instance with "Pattern:Role". This will quickly inform the reader on the role-specific assignments. Briefly outline each role and how it contributes to the pattern's essence.

Collaborations. Describe the interaction between the client and the pattern instance.

Consequences. Argue the pros and cons, e.g. design issues like extendibility, contrasting them with other design alternatives.

Implementation. Point out special features of your implementation and make references to the corresponding sections of the source code.

3.3 Extended Template for Documenting Frameworks

The second main pattern activity that we exemplified in Section 2.4 supplies a further argument for a pattern-oriented approach to documentation: The need to document those parts of a design that have been made *more flexible* by inserting a pattern. This

[7] http://www.ti.et-inf.uni-siegen.de/Entwurfsmuster/

is especially the case when flexibility is intended for *framework* development. We took this argument into account by the following extension to the previous template:

Hot Spots vs. Frozen Spots. For the documentation of a framework, a detailed description of the parts not to be changed and the parts to be extended by the user is essential [22, 24]. State clearly what degree of flexibility is offered and what the conceptual constraints are.

Recipe. Sketch the use and adaptation of the framework in a cookbook style with an example [16, 18]. If possible, give a ready-to-use example to test the framework by running it. To enhance one's practical understanding, a pre-configured debugger session could help. If your framework is designed for a certain run-time environment (e.g. an OLE component), consider an interactive on-line support (assistant or wizard) for adaptation and use of your framework. This would be the optimum help for the user.

Integration. State briefly (e.g. by making references to the corresponding locations in design and implementation) which assumptions the would-be environment of the framework has to fulfil.

Known Uses. List all successful uses (and typical misuses, too!), so that a prospect can quickly assess the potential of the framework for his requirements.

Structural Extensions. Mention the aspects that you consider limiting the framework's current design and implementation. If you have any solutions or hints for follow-up extensions, write them down. Localizing all relevant hot spots of a framework requires several design cycles in (ideally) slightly different environments. This section can document the history of these efforts.

Finally, some remarks on omissions: We have omitted here those links of the hypertext template that refer to the library management of pattern texts. For lack of space, we also left out the links to the information by whom and when the design pattern was instantiated in the process model (aspects of version control and process management). Similar templates for structuring documents can also be used at the *Class level* (see Fig. 12) to document attributes and operations. We did this, too, in our project [21].

4 Experiences Gained

Comparing our experiences with those from literature [4, 7, 8, 25, 26, 27], we can name the following two categories: Category A for experiences gained in applying design patterns *generatively* (where ours are similar to those from literature) and category B for experiences gained in applying a pattern's *form and content* for "documenting by designing". Let us contrast A with B: Identifying a candidate class structure to be merged with a design pattern for some flexibility reasons is harder than "just" instantiating a design pattern. As Booch observed: "Identifying involves both discovery and invention" [5]. In the case of "pattern instantiation", there is no fixed design context the designer has to take into account. Especially, when it comes to allocating class

roles, the case of "candidate identification" leaves a couple of design decisions open to the designer. For example: Which of the classes' roles of the given design can be matched by the abstract design pattern? Which new roles are there to be taken over from the pattern? Finally, there are also *synergetic* experiences gained from both categories, A *and* B, confirming the principle of documenting by designing.

Category A: Using Patterns for Design

- Cohesive and comprehensive documentation of *design decisions* (trade-offs, forces, context, etc.).
- Higher degree of reusability (making explicit the flexibility locations in a design (*hot spots* according to Pree [22]), facilitating maintenance).
- Potential pitfalls: patterns must have been *internalized* before they can be applied effectively. A basic knowledge of design patterns is sufficient to understand existing designs that refer to these patterns. However, when software is to be designed in the pattern fashion, design patterns must have been utterly understood and practised several times before.

Category B: Using Patterns for Documentation

- A design covered with pattern instances is accessible for understanding from both sides: (a) from an in-depth knowledge of the problem domain and (b) simply with design patterns in mind. The pattern documentation functions as the "missing link" between the meta-level of pattern annotations and the problem domain.
- Systematic support of natural language documentation enhances the stimulus for documenting in general.
- Alleviation of descriptive complexity: documenting a complex design with patterns draws one's attention to the *architectural* structures of interest and, hence, facilitates one's understanding of the design.
- The design *rationale* (i.e. compensation of forces and fitting to the problem's context) is made explicit.
- The guiding interplay of a pattern's form and content animates documentation and reduces the time spent for it.
- Hypertext is the ideal medium to take full advantage of the *dual* nature of design patterns: it facilitates navigation through design documents and is a guiding instrument during the design process.

Categories A and B: Using Patterns for Design & Documentation

- Synergetic effect: Designers documenting their designs with the pattern form will be more inclined to apply patterns, and vice versa.
- Smooth transition to framework development: in contrast to prior experiences from literature, this is a transition from a *given* design to a framework; other reports stated a transition from scratch [3], or documented the framework's rationale with patterns afterwards [16].

- Improvement of software documentation: Instead of *reverse* documenting a design, a design is documented while it evolves, that is, when design knowledge is at its highest.
- Potential pitfall: if a design pattern has been changed *structurally* to make it fit to a particular design context, all changes should be properly documented. Otherwise, the pattern instance will be difficult to recognize and its potential flexibility may not be seen any longer.

5 Concluding Remarks

Componentware, especially frameworks, will have a tremendous impact on software engineering in the years to come. On the one hand, the development of complex applications will be accelerated by visually configuring standard components. On the other hand, however, documenting the demanding design of such components will become a crucial factor to realize their *ilities*: applicability, reusability, extendibility, maintainability. We have tried, in this report, to show the correlation between design and documentation. The pattern-related design activities sketched in Section 2.2 also produce parts of the design documentation. Such a documentation will show the path that was followed from the initial design problem to its final solution: the design traps and pitfalls, the trade-offs, and the final rationale of a design are documented *in passing*, and, hence, can be easily traced back by another designer.

The spreading knowledge and acceptance of patterns will change our current *culture* of designing and documenting: There is a need to explicitly document the designer's patterns of thinking and doing. Eventually, this need will establish a new level of abstraction, what we call the "Pattern level" (see Fig. 12). It helps to put design reuse on a firm footing and will function as a *meta model* for documentation systems to come.

The project's results sketched in this experience report have encouraged us to keep on using patterns for design and documentation. Currently, we are *reverse* documenting a mature manufacturing framework [24] in the hypertext fashion of Sections 3.2 and 3.3. Besides, we consequently follow our principle of "documenting by designing" in a framework project for scheduling at universities [2].

Acknowledgements: We would like to thank the anonymous reviewers for their helpful suggestions and comments.

References

[1] Alexander, C., et al. *A Pattern Language: Towns, Buildings, and Construction.* Oxford University Press, New York, 1977.

[2] Baumgart, M., Kunz, H.P., Meyer, S., and Quibeldey-Cirkel, K. Priority-driven Constraints Used for Scheduling at Universities. In *Proc. of the 3rd Int. Conf. on the Practical Application of Constraint Technology*, London, UK, 1997 (accepted for publication).

[3] Beck, K., and Johnson, R.E. Patterns Generate Architectures. In *Proc. of ECOOP '94*, Bologna, Italy, 1994, 139-149.

[4] Beck, K., Coplien, J.O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., and Vlissides, J. Industrial Experience with Design Patterns. In *Proc. of 18th Int. Conf. on Software Engineering (ICSE '18)*, Berlin, Germany, 1996, 103-114.

[5] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. 2nd Edition.

[6] Brooks, F.P. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer 20*, 4 (1987), 10-19.

[7] Brown, K. Using Patterns in Order Management Systems: A Design Patterns Experience Report. *Object Magazine*, Jan. 1996.

[8] Budinsky, F.J., Finnie, M.A., Vlissides, J.M., and Yu, P.S. Automatic Code Generation from Design Patterns. *IBM Systems Journal 35*, 2, 1996.

[9] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.

[10] Coad, P. Object-Oriented Patterns. *CACM 35*, 9 (Sep. 1992), 152-159.

[11] Coad, P., and Yourdon, E. *Object-Oriented Analysis*. Englewood Cliffs, Yourdon Press, Prentice Hall, 1991.

[12] Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[13] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA, 1996.

[14] Gamma, E. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (in German). Ph.D. thesis, University of Zurich, 1991.

[15] Gamma, E., Helm, R., Johnson, R. E. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[16] Johnson, R.E. Documenting Frameworks Using Patterns. In *Proc. of OOPSLA '92*, Vancouver, BC, Canada, 1992, 63-76.

[17] Keller, R.K., and Lajoie, R. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In *Proc. of 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences*, Montreal, QC, Canada, 1994.

[18] Krasner, G.E., and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP 1*, 3 (Aug./Sep. 1988), 26-49.

[19] Linton, M.A., and Calder, P.R. The Design and Implementation of InterViews. In *Proc. and Additional Papers, C++ Workshop*, Santa Fe, NM, 1987. USENIX Association, El Cerrito, CA, 1987, 256-268.

[20] Meyer, B. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[21] Odenthal, G. *Design and Implementation of an Interface between the SAP-R/3 Business Object Repository and the Open Scripting Architecture (OSA)* (in German). University of Siegen, Master thesis, 1996.

[22] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, MA, 1994.

[23] Quibeldey-Cirkel, K. *The Object Paradigm in Computer Science* (in German). Teubner, Stuttgart, Germany, 1994.

[24] Schmid, H.A. Creating the Architecture of a Manufacturing Framework by Design Patterns. In *Proc. of OOPSLA '95*, Austin, USA, 1995.

[25] Schmidt, D.C. Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *CACM 38*, 10 (Oct. 1995), 65-74.

[26] Schmidt, D.C., and Stephenson, P. Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms. In *Proc. of ECOOP '95*, Aarhus, Denmark, 1995.

[27] Special issue on Software Patterns. *CACM 39*, 10 (Oct. 1996), 36-82.