Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis

Uwe Zdun
Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Austria
zdun@acm.org

Abstract

Software patterns provide reusable solutions to recurring design problems in a particular context. The software architect or designer must find the relevant patterns and pattern languages that need to be considered, and select the appropriate patterns, as well as the best order to apply them. If the patterns and pattern languages are written by multiple pattern authors, it might be necessary to identify interdependencies and overlaps between these patterns and pattern languages first. Out of the possible multitude of patterns and pattern combinations that might provide a solution to a particular design problem, one fitting solution must be selected. This can only be mastered with a sufficient expertise for both the relevant patterns and the domain in which they are applied. To remedy these issues we provide an approach to support the selection of patterns based on desired quality attributes and systematic design decisions based on patterns. We propose to formalize the pattern relationships in a pattern language grammar and to annotate the grammar with effects on quality goals. In a second step, complex design decisions are analyzed further using the design spaces covered by a set of related software patterns. This approach helps to systematically find and categorize the appropriate software patterns – possibly even from different sources. As a case study of our approach, we analyze patterns from a pattern language for distributed object middleware.

Keywords: Software patterns, pattern languages, pattern selection, design space analysis, software design

1 Introduction

Software patterns capture reusable design knowledge and expertise that provides proven solutions to recurring software design problems that arise in particular contexts and domains [45]. In recent years, patterns have become part of the mainstream of software development. Different types of software patterns have been published. The most popular software patterns are probably design patterns (see [21] or the proceedings of the PLoP conference series). Design patterns can be applied very broadly because they focus on everyday design problems. In addition to design patterns, the patterns community has created patterns for software architecture [12, 46, 49], analysis [20], and even non-IT topics such as organizational or pedagogical patterns. There are many other kinds of patterns, and some are specific for a particular domain. In this article, we focus on patterns for software architecture and software design.

Consider the relevant patterns in a design situation have been documented in different pattern texts and possibly by different pattern authors. In this and similar situations, systematically applying software patterns requires a certain amount of expertise from the software architect or designer, because she/he has to well understand how a pattern's solution fits into the overall architecture and how other patterns can be applied to resolve new open issues that might arise as a consequence of applying the pattern. To resolve these problems, today many pattern authors document patterns as part of larger pattern languages [5], containing rich pattern interdependencies and extensive examples or case studies (see for instance [46, 32, 49, 50, 53]). The purpose is to guide the architect or designer step by step to reach the domain-specific goal of the pattern language. Once an initial solution has been envisioned or designed using the pattern language, the patterns can be applied in an incremental refinement process that generates step by step a coherent "whole". Pattern languages are used for evolving whole architectures through this process of piecemeal growth (see [2]).

Patterns use a description format that supports the concept of piecemeal growth. In his original pattern concept, Christopher Alexander explains the role of patterns in a piecemeal growth process like this [5]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

A pattern following this concept must capture the possible variations of the same problem/solution pair. In other words, a pattern covers a whole problem space and an associated solution space, not only a single problem and solution[1]. This has severe consequences for the pattern description format: patterns are described with a consistent structure, but the pattern descriptions uses a fairly informal, narrative style. In addition, each pattern author uses a slightly different structure, according to the author's preferences and what fits best

[1] Below we use the term "design space" for this problem and solution space.

to the (technical) subject matter. Emphasis is on readability, understanding the language as a whole, and documenting all relevant variants and related patterns of a pattern in a domain.

In contrast to template-like descriptions of a solution to a problem, each pattern describes a great number of variants and possible relationships to other patterns, leading to a web of patterns that can be applied step by step. The individual paths through this web of patterns are called *pattern sequences* [3, 4]. Because this way a relevant domain can be captured quite completely and for each pattern all relevant links to other related patterns are documented, this description format enables piecemeal growth of a software architecture. This is a great strength of the pattern approach.

But at the same time, this strength is also a weakness because patterns require a certain expertise from the architects both for the relevant patterns and pattern languages and for the domain in which the patterns should be applied. Questions arise like: Which pattern should I choose first? Which variant of the pattern works best? Which pattern should be applied next? Usually the answers to these questions are not simple. In each particular design situation the architect must consider functional features as well as quality goals, such as performance, maintainability, reusability, etc. These quality goals are usually competing. That is, no optimal (i.e. maximal) solution can be found but only a compromise.

The main goal of this article is to supplement the pattern-based approach and the body of existing pattern literature[2] by providing an approach to better support the selection of patterns and systematic design decisions based on patterns. We propose a two step approach:

- Firstly, we formally document the grammar of a pattern language and annotate it with the effects on quality goals. From the pattern language grammar we can derive the pattern sequences of a pattern language. During the selection of sequence steps, we can use the quality goal annotations to get an approximate overview of the design decision's effects on quality goals.

- Secondly, we analyze in detail the more complex or difficult design decisions in the pattern language that can be found in the annotated pattern language grammar. This is done by documenting the design space [36, 35] for each of these design decisions. A design space explains the considerations and options of a particular design decision in depth.

The pattern language grammar and the design spaces have to be documented once for a pattern language (or a set of related patterns). The initial documentation of the pattern language grammar and design spaces can be provided by the pattern language author (as in the case study provided in this article), or it can be done during an architectural design process (as done in some of the projects discussed in Section 7). The

[2] Please note that many existing patterns in the literature are not documented as part of a pattern language, or miss important links to other, related patterns, for instance, because these other patterns were written later in time. We provide a practical approach which can be used to connect these patterns to existing pattern languages.

pattern language grammar and the design spaces can subsequently be applied to guide the pattern selection in many other concrete design situations. They are therefore also a means to document and reuse the design knowledge and design rationale from earlier design experiences based on a pattern language. Finally, as pointed out in Section 7, a pattern language grammar and design spaces documentation can also be used as a way to communicate design decisions to technical and non-technical stakeholders of a software system.

2 Patterns, pattern languages, and pattern selection

2.1 Definition

Patterns go back to the original pattern concept by the architect Christopher Alexander. In his book, *A Pattern Language* [5], he describes 253 patterns that guide the creation of space for people to live, including cities, houses, rooms, and so on. On the Hillside web site [27] a definition of software patterns is presented that is derived from the discussion in Alexander's book:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

A single pattern describes one solution to a particular, recurring problem. The full power of patterns is only achieved, however, if they are used in the context of each other to form a web of related patterns, more commonly known as a *pattern language* [14]. A single pattern can only solve a single problem. Patterns in a pattern language, in contrast, build on each other to generate a system. In the language context, the benefits of a set of related patterns is more than the sum of the benefits of each individual pattern in the set. Simply connecting the patterns is thus not enough. Patterns have to link to other patterns in the language that refine and complete them. Pattern languages have several further characteristics:

- A pattern language has a language-wide goal. The purpose of the language is to guide the user step by step to reach this goal. The patterns in a pattern language are not necessarily useful in isolation.
- A pattern language is generative in nature. Applying the pattern language generates a 'whole'. The whole is generated by applying the patterns in the pattern language one after another in an incremental process of refinement. The basic idea is that each refinement step makes the whole successively more coherent.
- To generate the whole, the pattern language has to be applied in a specific order. This order is defined by one or more sequences. Depending on the context in which the pattern language is applied, or which aspects of the whole are actually required, there can be several sequences through a pattern language.

- Because the patterns must be applied in a specific sequence, each pattern must define its place in the sequence. To achieve this, each pattern has a section called its context, which mentions earlier patterns that must be implemented before the current pattern can be implemented successfully. Each pattern can also feature a resulting context that describes how to continue. This contains references to the patterns that can be used to help in the implementation of the current pattern, or explains how to proceed in the incremental refinement process of the whole.

2.2 Pattern selection

The selection guidance offered by the different forms of pattern descriptions are highly valuable during an (iterative) design process. But for systematic design decisions based on both functional requirements and quality goals, a certain amount of expertise is required from the architect – both in the existing pattern literature and in the domain, where the patterns should be applied.

Well-elaborated pattern descriptions nevertheless contain all relevant information necessary for an informed design decision:

- *Functional requirements:* Each pattern describes a *solution* to a *problem* in a *context*. That is, the pattern describes which functional requirements it can potentially fulfill.

- *Quality goals:* Each pattern contains a description of the *forces* that govern the solution to the pattern's problem, as well as the positive and negative *consequences* of applying the pattern as a solution. These sections usually describe in detail the potential influences of the pattern on quality goals.

- *Pattern variants:* In the pattern's *solution* and in sections that describe the solution in detail, different variants of the pattern are described.

- *Related patterns:* Most pattern description formats contain an implicit or explicit *related pattern* section, pointing for instance to patterns that can be applied to new problems potentially arising as a consequence of applying the pattern.

As argued above, unfortunately, this information does not always lead us to systematic design decision based on patterns. Gamma, Helm, Johnson, and Vlissides (the "GoF") have already considered this as a problem for their catalog of design patterns: "with more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem" (p. 28, [21]). They then propose several different approaches to find a design pattern that is right for a particular design problem:

- Consider how design patterns solve design problems

- Scan pattern overviews[3]
- Study how patterns interrelate
- Study patterns of like purpose
- Examine a cause of redesign
- Consider what should be variable in your design

Pattern languages lay a stronger emphasis on pattern interrelations and morphological unfolding of a solution than the patterns in the GoF book by using the above mentioned process of piecemeal growth. For pattern selection, however, pattern languages also require the architect to study the pattern language in detail. There are a number of reasons why this might be a problem:

- *Selecting from many pattern sources:* There are usually much more patterns to be considered than just the 23 design patterns in the GoF book, simply because in many domains the pattern community has provided significantly more pattern material since. The above named selection approaches, however, require an architect to search through the patterns one by one, and remember the patterns' contents. If a significantly larger body of patterns needs to be understood, this is much work. This can only be mastered easily with sufficient expertise both in the domain and for the patterns. If the patterns and pattern languages are written by multiple pattern authors, it might be necessary to identify interdependencies and overlaps between these patterns and pattern languages first. Different pattern formats and writing styles must be aligned and mapped.

- *Reducing complexity of pattern selection:* Patterns usually need be understood in the context of other patterns, from the same pattern source and also from other pattern sources. The possible number of pattern combinations to be considered for a given design problem thus can be huge, but only a few of these possible pattern combinations work as a solution for solving the design problem. It is desirable to reduce the complexity of the design decision by eliminating all pattern combinations from consideration that do not work.

- *Selecting from domain-specific pattern sources:* The GoF patterns describe general solutions for software design. Today many architectural pattern languages are domain-specific and have rich interdependencies to other domain-specific pattern languages. Just consider the Remoting Patterns in [49]. In order to apply this pattern language, you usually have to consider patterns from other pattern languages,

[3] "Intent section" in the GoF book. Today many pattern authors use other overview-like sections, such as pattern thumbnails or problem/solution-pairs.

such as networked objects [46], resource management [32], general-purpose patterns [21, 12, 6], server-component patterns [50], language and component integration [55], aspect-oriented composition patterns [53, 51], and so forth. Thus the design decisions might become much more complex, due to these domain-specific dependencies.

- *Balancing quality goals of subsequent design decisions:* A good pattern language describes for each individual pattern and for each subsequent design step the consequences on quality goals. There is no overview of the consequences of a number of related design steps, however. The holistic consequences have to be deduced by stepwise examination of the patterns, one after another.

- *Coping with insufficient pattern material:* Not all domains are covered completely by patterns and not all patterns are well written. In such cases, a pattern selection approach should allow one to *identify gaps* in the patterns and in their relationships to other patterns.

3 Overview of the approach

Our concept is to create the pattern language grammar and the design spaces once for a pattern language. Then they can be used for multiple, systematic design decisions using the pattern language. Of course, using the pattern language grammar and the design spaces might create a feedback that improves them. Thus the pattern language grammar and the design spaces should not be seen as something static that cannot be changed later on[4].

Before we can apply our approach we need to identify the relevant quality goals of the patterns in a pattern language. Of course, the functional goals are important for a design decision. Functionality is the ability of the system to fulfill the task it was created for. A system that does not adequately fulfill its functional requirements is doomed to fail its user's expectations. The reasons for architectural defects and subsequent architectural reengineering, however, are often not concerned with the functionality of the system, but other quality goals, such as the system's performance, maintainability, portability, security, usability, scalability, time-to-market, or costs. In pattern descriptions we can find the quality goals and concerns addressed by the pattern in the *forces* and *consequences* sections.

The quality of an architecture (or a software system in general) can be seen as a collection of quality attributes [7]. Even though there are forces and consequences in an architectural pattern that do not directly relate to quality attributes, most often some quality attributes depict central aspects of an architectural pattern's design considerations. We use two systems of quality attributes, which complement each other: the

[4]Note that this view is consistent with the pattern community's view on patterns: they are also seen as living documents that change over time, for instance, because the technology for implementing the patterns changes and thus new pattern variants or dependencies to other patterns emerge.

system by Bass, Clement, and Kazman [7] and the ISO 9126 *International Standard for the Evaluation of Software* [30]. It is not particularly important for our approach which quality attributes are used to describe the quality goals; it is just important that they are well-defined in the context of a pattern language so that different readers do not understand different things for the same term. In other fields than software architecture, patterns have different kinds of quality goals as forces and consequences, other than architectural quality attributes. Our approach is nevertheless still applicable, provided that these quality goals are well-defined.

After identifying the quality goals of the patterns we document the grammar of the pattern language and annotate it with the effects on quality goals. The formalization allows us to derive the possible pattern combinations as pattern sequences. This step reduces the possible combinations of patterns dramatically, and thus gives a good overview of the possible pattern combinations in the pattern language. It also gives an overview of the effects on quality goals of the patterns and pattern combinations.

For the more simple design decisions in the pattern language, the pattern language grammar and the pattern sequences provide a good basis for a design decision. Some design decisions are more complex, however. For instance, in most cases where multiple variants and alternatives need to be considered, a more detailed analysis is needed. Therefore, as a next step, we identify the more complex design decisions, described by the pattern language grammar. For each of them we perform a question-option-criteria (QOC) design space analysis. The result is a detailed decision map for the design decision in question.

4 Formal documentation of the pattern language grammar

4.1 Pattern language grammars and pattern sequences

As Alexander points out [3, 4], pattern descriptions alone do not really allow a person to generate a good design, step by step, because they concentrate on the content of the patterns rather than laying the emphasis on morphological unfolding. The creative power lies in the sequences in which the patterns are applied. For a given task, the number of possible sequences is huge compared with the number of sequences which work.

The informal pattern overview diagrams (Figure 1 shows an example from [49]), used by many pattern authors, give a good overview of how the patterns relate, but they are not sufficient to derive the possible sequences in which the patterns can be selected during a design decision. In fact, we can understand the topology of the patterns in a pattern language as a partially ordered set (poset), and each sequence in the pattern language is a totally ordered set of patterns from the pattern language [43].

Note that there is more to a pattern language than its topology. A defining characteristic of any language is its grammar. It is from grammar that we can derive sequences [26].

In a pattern language, the language elements are individual patterns. To support a more systematic design decision, we transform the pattern overview diagrams and the information in the pattern texts into a formal

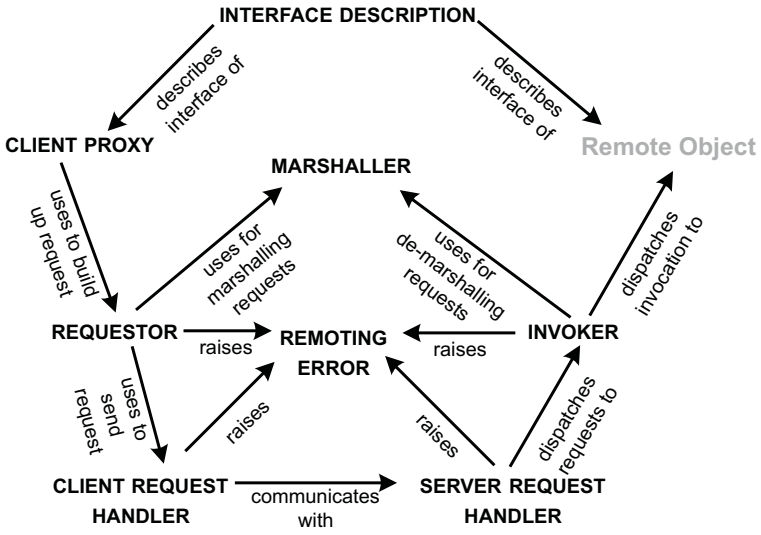

Figure 1. Example of an informal pattern diagram (from [49])

pattern language grammar. In order to document the quality goals of the design decisions, we annotate the grammar with expected effects on quality goals. To document a pattern language as a formal language, and apply our approach in concrete projects, it is important to understand the following two points:

- A pattern does not describe a structural element of the system to which it is applied, but it can be seen as an *event in a creative design process*. Each pattern describes a whole problem and solution space, not only a single solution, and thus selecting a pattern means to start a creative design process within this problem and solution space. The order of the language elements in the sequences thus describes the temporal order of a design decision. That is, the sequence $< A, B, C >$ means that pattern A is selected before pattern B, which itself is selected before pattern C.

  The order of pattern application is most often identical to the order of pattern selection. It is also possible, for instance, that some patterns that appear in the same pattern sequence are applied in parallel, or that a sequence needs to be applied as a whole. Consider, for instance, a sequence includes the patterns CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER (both from [49]). The fact that one of these two patterns was selected before the other pattern, which might be important for the documentation of the design rationale, does not imply that the two patterns must be implemented in that order. Usually the two patterns are implemented in parallel because implementations of these patterns usually share common components, and a client component is needed to test a respective server component and vice versa.

- A formal pattern language grammar does not necessarily describe all possible sequences of the patterns in a pattern language, but only a number of successful solution paths that have proven to work in the past. The sequences derived from a formal pattern language grammar need not be complete in the

sense that they cover all possible combinations that might work, nor do they need to work for all time. As mentioned above, a pattern language grammar should be seen as a living document that changes as new pattern combinations are mined, existing pattern combinations become obsolete, other patterns are added to the pattern language, technology changes, etc.

We propose a two step approach to come to a pattern language grammar. In the first step, we create a pattern language grammar overview diagram that uses only dependencies that can be mapped to formal language grammars. We annotate each dependency in this diagram with the expected effects on quality goals. In the second step, we also document the formal pattern language grammar by deriving it from the overview diagram (this can potentially be automated using a tool). Note that the pattern language grammar overview diagrams are not only useful as an intermediate step to create the pattern language grammar, but as they are more easy to read than the grammar itself, they are also useful for discussing design decisions, e.g. with the stakeholders of a software system. The formal grammar, in turn, is mainly useful for creating tools, say, by using a parser generator.

In this paper, we concentrate on examples showing and explaining the pattern language grammar overview diagrams because this representation is more useful for explanatory purposes than the formal grammars. We also present the details of the formal grammars to illustrate that the diagrams can easily be transformed into a more formal representation, suitable for providing tool support.

4.2 Mapping grammars to pattern relationships

In a grammar of a formal language, it is usually possible to provide production rules for required elements, optional elements, alternatives, and repetitions. We can map each of these elements to the possible relationships that patterns in a pattern language can have. We find the possible pattern relationships in the pattern overview diagrams and by scanning the pattern texts, as well as by looking at known uses or example implementations of the pattern language. In the pattern texts we search for *related patterns* and *pattern variants*.

The application of a pattern in a design context leads to a new context in which other patterns can be applied. A pattern can either require another pattern, or the use of the other pattern is optional. These two kinds of pattern relationships thus can be directly mapped to the required and optional elements of a formal language grammar. In the pattern language overview diagrams we mark optional pattern relationships as `[option]` and required pattern relationships as `[required]`.

The alternative element can be mapped to alternative variants (marked as `[variants]`) or alternative related patterns in a pattern language (marked as `[alternatives]`). `[variants]` and `[alternatives]` default to non-exclusive alternatives (‘or’) because only in seldom cases design decisions for patterns exclude each other. An example where exclusive alternatives are used is the pattern language *State Patterns* [17]. In

this pattern language, for instance, the absence of the pattern STATE MEMBER leads to the applicability of the pattern PURE STATE. Thus the design decision for one of the patterns means that the other pattern must not be selected. In such cases, we add `(xor)` in the notation as follows: `[variants (xor)]`, `[alternatives (xor)]`.

The repetition elements of the formal language grammar are needed to represent options and alternatives (more than one option or alternative may be chosen in one sequence). However, they are not directly mapped to pattern relationships. This is because in our formalization we interpret patterns as events in the pattern selection process. Consider the pattern sequence $< A, B, C >$ and the pattern sequence $< A, A, A, B, C >$. These two sequences are semantically equivalent regarding pattern application and pattern selection. They both say that the patterns are selected in the order: A, B, C. Please note that both sequences do not say how often one of the patterns is applied during the sequence. For instance, if the pattern INVOKER [49] is part of a sequence, this can mean that more than one INVOKER instance needs to be implemented, for instance one INVOKER per existing remote object type or backend type. In the respective sequence the INVOKER needs to appear only once. For this reason, we eliminate consecutive duplicates from a pattern sequence[5].

4.3 Pattern language grammar overview diagrams

As mentioned above, first we want to produce pattern language grammar overview diagrams. In these diagrams, boxes with the pattern names are connected with arrows that are labeled using the relationships introduced above. Sometimes participants, variants of patterns, or components of the domain are described, which are not patterns themselves, but have an important relationship to the patterns. In the diagrams these are used like patterns, but they have no box and are rendered in gray.

Each pattern and pattern variant in these diagrams represents one design solution. Thus the diagram already provides an overview of the topology of the pattern language. But to understand the possible design steps in a pattern language, we must additionally consider the forces of the patterns. The forces are a list of factors that influence the pattern's solution. In the software architecture field, forces are often quality attributes of the solution. We thus propose to add these quality attributes (and other quality goals) to the pattern relationships in the diagrams using a simple notation: the quality goal is followed by an approximated score.

- `++`: a very positive influence on the quality goal can be expected
- `+`: a positive influence on the quality goal can be expected
- `o`: no influence on the quality goal can be expected

[5] Please note that we discussed this mapping issue only for explanatory purposes: if a tool is used to derive the formal grammar from the pattern language grammar overview diagrams, of course, only repetitions which make sense as patterns selections are derived.

- `-`: a negative influence on the quality goal can be expected
- `--`: a very negative influence on the quality goal can be expected

If a quality goal is not mentioned, the meaning is identical to the "`o`-symbol": no influence on the quality goal can be expected. In addition to this simple score, scores can be combined using the slash-symbol ("`/`"). For instance, "`maintainability o/--`" means either no or a very negative influence on maintainability can be expected.

We also allow for comments in diagrams, describing aspects that are not yet covered by the visible diagram. These are placed in curly brackets ("{...}").

Figure 2 shows an example of a pattern language grammar overview diagram.

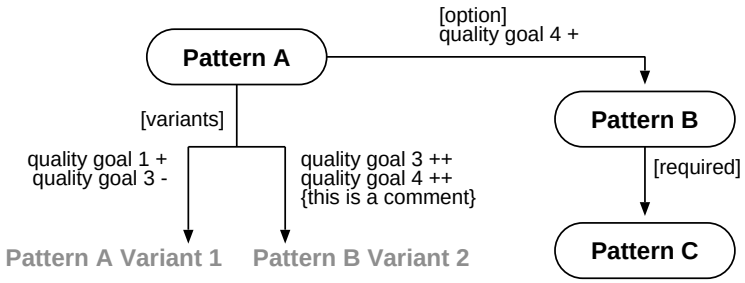

Figure 2. Example of a pattern sequence diagram

4.4 Deriving a formal pattern language grammar

In a second step, we transform the pattern language grammar overview diagrams into a formal grammar for the pattern language. A grammar G of a language L is defined as the tuple: $G = (V, T, P, S)$, where:

- V is a set of syntactic variables (non-terminals).
- T is a set of terminal symbols, which represent a sub-set of the alphabet A of the language L. The terminal symbols in a pattern language grammar are patterns (and pattern variants).
- P is a set of production rules $X \longrightarrow Y$, meaning that X can be transformed into Y. Both X and Y can contain syntactic variables and terminal symbols. The production rules are applied until all syntactic variables are transformed into terminal symbols. The result of this transformation is a pattern sequence that conforms to the grammar of the pattern language.
- S is a start symbol, $S \in V$, which starts the application of the production rules.

The language L is the set of all words w that can be derived from $G = (V, T, P, S)$. In a pattern language the words w are the sequences that can be derived from the pattern language.

$$L(G) = \left\{ w \ in \ T^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} w \right\}$$

As an example, let us derive the grammar *G_EXAMPLE = (V_EXAMPLE, T_EXAMPLE, P_EXAMPLE, S_EXAMPLE)* for the example diagram in Figure 2:

V_EXAMPLE = { *patternA_selection, patternA, patternAs, patternA_options* }
T_EXAMPLE = { *“”*, *“*PATTERN A*”*, *“*PATTERN A*: Pattern A Variant 1”*,
*“*PATTERN A*: Pattern A Variant 2”*, *“*PATTERN B*”*, *“*PATTERN C*”* }
P_EXAMPLE = { *patternA_selection* ⟶ *patternAs patternA_options*,
patternA ⟶ *“*PATTERN A*”*,
patternA ⟶ *“*PATTERN A*: Pattern A Variant 1”*,
patternA ⟶ *“*PATTERN A*: Pattern A Variant 2”*,
patternAs ⟶ *patternA*,
patternAs ⟶ *patternA patternAs*,
patternA_options ⟶ *“”*,
patternA_options ⟶ *“*PATTERN B*” “*PATTERN C*”* }
S_EXAMPLE = { *patternA_selection* }

4.5 Deriving sequences from the pattern language grammar

Using the grammar we can derive sequences that conform to the pattern language. Please note that deriving sequences from pattern language grammars should not be done manually. Instead we propose to generate a language parser from the pattern language overview diagram using a parser generator. A tool can then use the generated parser to validate that the sentences (i.e., pattern sequences) conform to the language. For explanatory purposes, we nevertheless will derive two example sequences from the example grammar above. For instance, we can derive a sequence that only selects “PATTERN A” using the following production rule transformations:

1. *patternA_selection*
2. *patternAs patternA_options*
3. *patternA “”*
4. *“*PATTERN A*” “”*

—

Resulting sequence = <“PATTERN A”>

Please note that in the resulting sequence we have eliminated the empty element. Another example is to select the option and both of the two variants of “PATTERN A”:

1. *patternA_selection*
2. *patternAs patternA_options*

3. *patternA* *patternAs* “PATTERN B” “PATTERN C”

4. “PATTERN A*: Pattern A Variant 1”* *patternA* “PATTERN B” “PATTERN C”

5. “PATTERN A*: Pattern A Variant 1”* “PATTERN A*: Pattern A Variant 2”* “PATTERN B” “PATTERN C”

—

Resulting sequence = <“PATTERN A*: Pattern A Variant 1”*, “PATTERN A*: Pattern A Variant 2”*, “PATTERN B”, ‘PATTERN C”>

Often we are faced with multiple pattern languages that have links among each other. That is, a pattern in a pattern language is also part of another pattern language or referenced as a related pattern. In such cases the pattern appears in two pattern language grammars as a terminal symbol. Thus it can be used to compose sequences from the two pattern languages.

For instance, consider the sequence $S1$ = <A, B, C> is derived from pattern language $L1$ and the sequence $S2$ = <X, Y, A, Z> is derived from pattern language $L2$. In the two sequences the same pattern A is selected. Thus A can be used as a link from $L1$ to $L2$, and vice versa. For instance, after selecting $S2$, we can continue with a design decision for $S1$. When we apply the patterns for $S1$, we (most likely) do not need to apply A anymore, because it was already applied during the application of $S2$.

5 Patterns-based design space analysis

The pattern language grammar is a formalism that helps us to systematically understand the topology and the language element relationships in the pattern language. It is important, however, to be aware of the pattern language grammar’s scope of application. Of course, there is more to a pattern language than grammar [26]. For a pattern language, especially the question of why we choose a particular sequence among the possible alternatives is important.

The annotated pattern language grammar gives an overview of the pattern language topology and the effects on quality goals. In some cases, this is enough to come to an informed design decision. A simple example is a `[required]`-relationship: if pattern A requires pattern B, the design decision is already clear. Where a more detailed design decision is needed, the subsequently introduced QOC design space analysis should be applied. This is especially needed for pattern variants and alternatives because here usually many factors need to be considered at once.

The goal of the QOC design space analysis approach is to qualitatively analyze and decompose software patterns to create an organized “space of design patterns”. Let us first give a brief motivation for design spaces in general. Then we explain the details of the approach.

5.1 Design spaces

Design space analysis is a common technique applied to design problems in human-computer-interaction (HCI) and other design disciplines [36, 35]. The *design space* is an explicit representation of alternative design options and reasons for choosing among those options. In contrast to the traditional conception of design, which assumes that the output of the design is only a specification or artifact, the output of the design is conceived as a design space. The design space preserves the thinking and reasoning that went into its creation, the design rationale.

The main purposes of documenting the design rationale in this way are (see [16]):

- to serve as a means for communication among design team members
- to transfer design knowledge and expertise between projects with similar rationales
- to encourage the explicit and systematic consideration of alternatives

Note that these goals are pretty similar to those of software patterns. It is thus desirable to combine the two approaches to their mutual advantage; this is the goal of the approach proposed in this article.

Design space analysis is one approach proposed for capturing the design rationale. Lee and Lai categorize it as a structure-oriented approach [34], because the approach is concerned with the structure of the space of all design alternatives, which may be constructed considering the design process after the event. Other approaches for documenting the design rationale are using a process-oriented design rationale or a psychological design rationale. The structure-oriented approach, however, is more suitable for the goals pursued in this article because we want to analyze the design decisions in a pattern language, which is also a primarily structural description of a design process: it captures the recurring temporal structure of pattern sequences [43].

There is one important difference in our use of design spaces to its traditional perception in HCI. The traditional view in HCI applies the design space to a concrete design, whereas our approach is to build a more abstract design space around the patterns, which are themselves abstractions used to describe numerous concrete designs. The reason for this is the changing nature of software: design rationale capture requires a lot of effort and the created models are hard to maintain. With every software change, the design rationale documentation must be updated, which can only seldom be done under the daily pressure in practice. Thus, with every evolution step, the documented design rationale gets more and more out-of-sync with the system that is documented. To avoid this problem, we concentrate on patterns as elements of the design space because they are more stable in time than the elements of a single software design.

5.2 QOC design space analysis

MacLean, Young, Bellotti and Moran [36] propose a semi-formal notation, called QOC, for *questions*, *options*, and *criteria*, to represent the design space around an artifact being produced:

- the *questions* highlight the key issues to be considered in a design situation
- the *options* are the possible answers to the questions
- the *criteria* are the reasons that argue for or against the possible options of a question

QOC-analysis is used for traditional design space creation, mainly in the HCI field. For the systematic derivation of a detailed design decision, we also use a questions, options, and criteria notation to visualize alternatives for design decisions and related design considerations. In addition to the original approach in HCI, we need to document the mapping of the questions and options to pattern alternatives and relationships. The criteria need to be mapped to the functional solutions in the patterns, as well as the quality attributes that can be found in the consequences and forces of the patterns. In the remainder of this section, we explain the necessary mapping processes and the visual notations used.

The goal of a QOC-analysis is to provide a detailed decision map for one design decision. As depicted in Figure 3 the first step of a QOC-analysis is to identify those patterns that are relevant for a design decision. The previously performed formalization of the pattern language grammar delivers those patterns: especially the alternatives and variants are interesting to be explored further, and the patterns related to these design decisions, because they likely lead to complex design questions. We explain the process of mapping the relationship type “alternatives” first.

For each of the sets of alternatives identified, we must create a design space question (Q). The result are a number of top-level questions. The options belonging to questions are the solutions that might solve the design problem raised by the question. That is, the respective patterns, pattern variants, and other alternative solutions are used as options (O) for the question. As shown in the Figure 4 questions are visualized in a box marked with “Q” and are connected to their options with an arrow to an option box, marked with “O”.

The next activity is to identify all criteria (C) for the options. As explained before, the criteria are (mainly) derived from the forces and consequences documented in the pattern descriptions. Usually alternative patterns have some overlapping forces and consequences, and some forces directly lead to consequences. Thus we catalog the forces and consequences of all options belonging to one question in one list, and eliminate all duplicates in this list.

In the next activity we need to map the options to criteria. In the design space diagram we draw a solid line from the option to the criterion, if and only if the option fulfills the criterion. That is, we can draw such a line, if the pattern has a positive influence on its force, or the criterion is a positive consequence of the

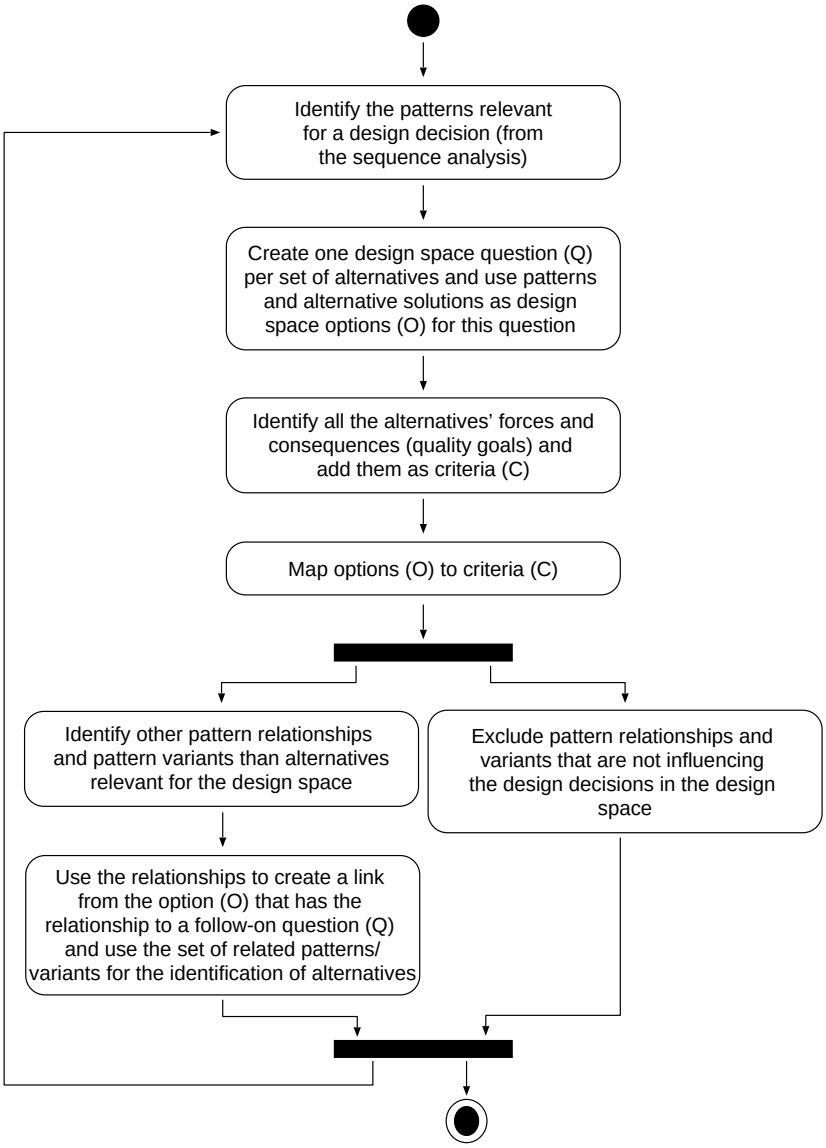

Figure 3. Activities for mapping patterns to a design space

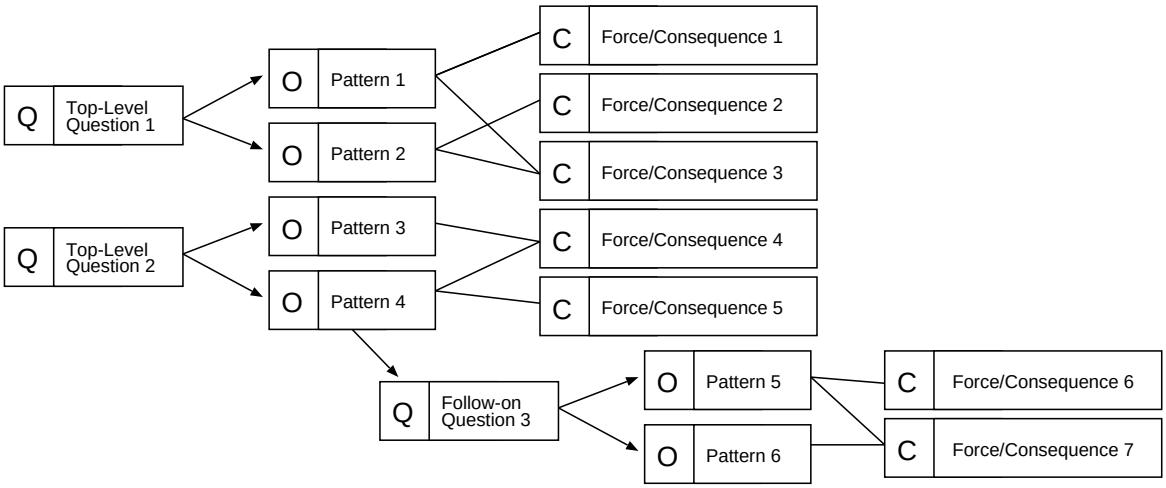

Figure 4. Template for design space visualization

Top-Level Question 1		
	Pattern 1	Pattern 2
Force/Consequence 1	X	
Force/Consequence 2		X
Force/Consequence 3	X	X

Table 1. Mapping table template for top-level question 1

pattern. Note that it might be necessary to map one force or consequence to a number of criteria so that an unambiguous mapping of options to criteria is always possible.

For convenience we also derive a mapping table for a question in a design space containing the same information, if the mappings between options and criteria are hard to read. Table 1 shows such as mapping table as a template for "Top-Level Question 1" from Figure 4.

Patterns have relationships to other patterns and sometimes these patterns need to be considered for a design decision.

For each pattern that is used as an option in the QOC design space, it should be considered whether its related patterns need to be considered for the QOC design space. This should be assessed by answering the following question: Does one of the options (i.e. one of the patterns in which the pattern relationship is mentioned), when the option is chosen, lead to a new, follow-on design question? If this is the case, we should draw an arrow between the option and the follow-on design question, as depicted in Figure 4 for "pattern 4" and "follow-on question 3". Next, we need to go through the whole process again for this new question (i.e. identify patterns as options and map criteria to these options).

All pattern relationships that are not directly related to the design space should be excluded from consideration in the design space. Thus, the resulting design space is a complete, hierarchical decomposition of one design problem using patterns and their variants as options (solutions). Other top-level questions or design spaces can be used to analyze other design problems covered by the pattern language.

When there are no more pattern relationships left to be considered, the QOC-analysis is finished. Note that there are no feedback loops depicted in Figure 3 for the sake of simplicity. Of course, feedback loops need to be considered between all activities. If, for instance, the identification of a pattern relationship reveals another alternative in the core patterns, this alternative should be added to the core patterns and the whole process should be applied (again) considering this new alternative.

So far we have discussed how to handle design decisions for mutual alternatives because the QOC-analysis approach can basically only distinguish between alternative options (O) for a question (Q). Thus, the approach directly covers alternatives in the pattern language grammar (i.e. `[variants]`-relationships and `[alternatives]`-relationships in the pattern language grammar overview diagrams). But how should `[option]`-relationships and `[required]`-relationships be handled?

`[required]`-relationships are very simple: if a pattern implies the use of another pattern, there is no extra design decision necessary. Thus we do not need to perform a QOC-analysis of `[required]`-relationships.

`[option]`-relationships can be mapped to options (O) in the QOC-analysis. But how to derive the question (Q) then? We can simply use the opposite of the option, and ask as a question (Q): "Should the option (O) be taken?"

Note that a QOC-analysis should only be performed if the information in the annotated pattern language grammar is not enough to make an informed design decision. It does not make much sense to add a QOC-analysis for very simple `[option]`-relationships or its alternatives. Especially if more simple decisions are part of a larger design decision, it makes sense to perform a QOC-analysis for that design decision. For instance, some alternatives might have an `[option]`-relationship as a related pattern, and the option chosen here influences the decision for one of the alternatives. In such complex cases, where the annotated language grammar does not reveal the complete interdependencies, a QOC-analysis can provide a much more detailed view on the design decision, because it captures both structural and semantic information.

6 Case study: Remoting patterns

In this section, we describe how we derived a design space for the remoting patterns, described in [49], following the approach explained above. The remoting patterns describe the inner workings of distributed object middleware systems (such as CORBA, Java RMI, .NET Remoting, or Web Services), and how to use them for creating distributed systems. The remoting pattern language consists of 31 patterns, and has numerous links into other pattern languages and pattern systems. A full design space analysis for the whole pattern language would be much too long for this article. Hence we only present a design space analysis for a part of the remoting pattern language: the asynchronous invocation patterns.

6.1 Overview of the asynchronous invocation patterns

In contrast to synchronous or blocking invocations, asynchronous invocations allow a client to resume its work while a remote invocation is running. The most common variants of client-side asynchrony are described by the asynchronous invocation patterns. Developers can use four asynchronous invocation patterns, rather than ordinary synchronous invocations, if invocation asynchrony is required:

- The FIRE AND FORGET pattern describes best effort delivery semantics for asynchronous operations but does not convey results or acknowledgments.
- The SYNC WITH SERVER pattern describes invocation semantics for sending an acknowledgment back to the client once the operation arrives on the server side, but the pattern does not convey results.

- The POLL OBJECT pattern describes invocation semantics that allow clients to poll (query) for the results of asynchronous invocations, for instance, in certain intervals.
- The RESULT CALLBACK pattern also describes invocation semantics that allow the client to receive results; in contrast to POLL OBJECT, however, it actively notifies the requesting client of asynchronously arriving results rather than waiting for the client to poll for them.

All four patterns can be used when synchronous invocations are insufficient because an invocation should not block. Developers can use FIRE AND FORGET or SYNC WITH SERVER when no result should be sent back to the client. POLL OBJECT is appropriate when a result is required and the client has a sequential programming model, whereas RESULT CALLBACKS require a client with an event-based programming model.

6.2 Annotated pattern language grammar

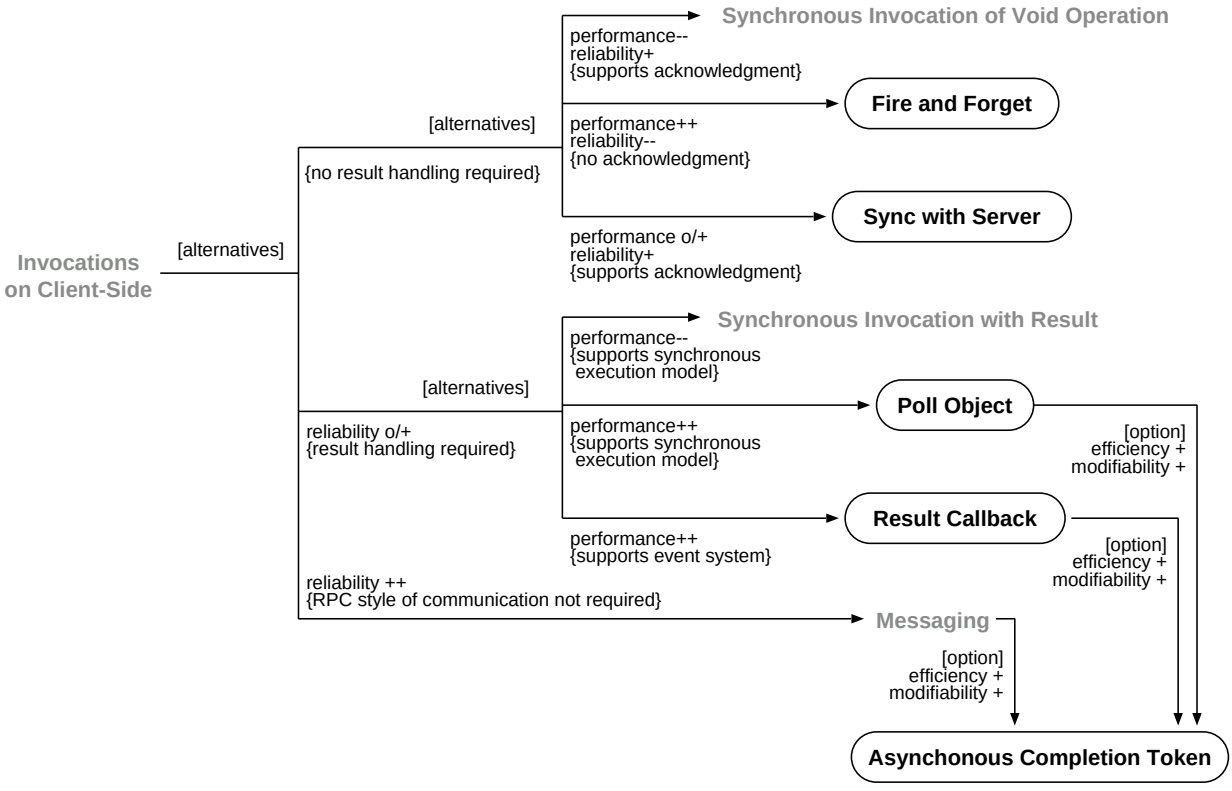

Figure 5. Annotated pattern language grammar overview for asynchronous invocations

Obviously all four asynchronous invocation patterns are mutually exclusive. Figure 5 shows an overview of the pattern language grammar. Deciding for one of the patterns as a basis for a client communication model might have a significant impact on the architecture of the client. For instance, RESULT CALLBACK requires an event-based programming model, whereas POLL OBJECT works in a client with a sequential control flow

as well. Changing a complex client from a sequential control flow logic to an event-based programming model is a complex task. Thus a systematic upfront decision for an invocation architecture is necessary.

Ordinary synchronous invocations are an alternative to all client asynchrony patterns. When examining the pattern descriptions a sixth alternative becomes obvious: a messaging infrastructure, for instance following the messaging patterns explained in [29], can be used as well. In messaging systems, also called message-oriented middleware (MOM), messages are not passed from client to server application directly, but through intermediate message queues that store and forward the messages. Messaging is inherently asynchronous. Many messaging protocols provide a reliable means of message delivery, such as an acknowledgment scheme, and tolerance of temporal failures among peers. Consequently messaging systems can provide a high level of reliability to their users.

These six alternatives are shown in Figure 5 with their respective quality goals. For the three asynchronous variants that provide a result (POLL OBJECT, RESULT CALLBACK, and messaging), we need a way (i.e. an `[option]` in Figure 5) to align the result to the invocation: when invocations are performed asynchronously, it is possible that one client sends multiple invocations after another, and results for these invocations arrive in a different order than the invocations. Thus the order or OBJECT ID is not sufficient to align invocation and result. This is solved by the ASYNCHRONOUS COMPLETION TOKEN pattern [46] (the respective messaging pattern is called CORRELATION IDENTIFIER [29]), which contains information that identifies the individual invocation and is sent along with each asynchronous invocation of a client.

The asynchronous invocation patterns have relationships to some of the basic invocation patterns from the remoting pattern language; in particular (see also Figure 6):

- The pattern REQUESTOR is used on client-side to construct and forward invocations. A REQUESTOR is inherently required for implementing the asynchronous invocation patterns.
- A CLIENT PROXY is a placeholder for the remote object in the client process. By presenting clients an interface that is the same as the remote object's, the proxy lets the client interact with the remote object as if it were a local object. Internally, the client proxy transforms the invocations it receives into REQUESTOR invocations, which the REQUESTOR then constructs and forwards to the target remote object. The CLIENT PROXY is optional for the asynchronous invocation patterns: it has a slight performance impact, but lets the asynchronous invocations be handled by generated code, which might be beneficial for maintainability and understandability.

6.3 Formal pattern language grammar

The formal pattern language grammar *G_ASYNC = (V_ASYNC, T_ASYNC, P_ASYNC, S_ASYNC)* for the asynchronous invocation patterns can be derived from the diagrams in Section 6.2 as follows:

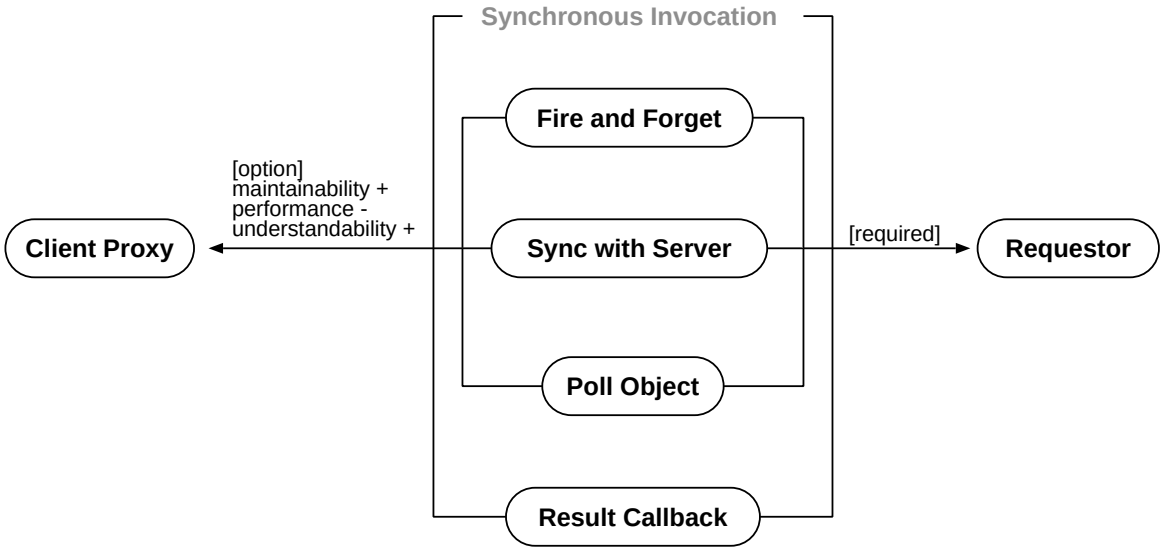

Figure 6. Asynchronous invocations and basic invocation patterns

V_ASYNC = { *invocation_alternative, invocation_alternatives, result_alternative, result_alternatives, no_result_alternative, no_result_alternatives, poll_object_selection, result_callback_selection, messaging_selection, fire_and_forget_selection, sync_with_server_selection, sync_void_selection, sync_result_selection, cp_invocation_option, act_options* }

T_ASYNC = { *""*, *"Synchronous Invocation of Void Operation"*, "FIRE AND FORGET", "SYNC WITH SERVER", *"Synchronous Invocation with Result"*, "POLL OBJECT", "RESULT CALLBACK", *"Messaging"*, "REQUESTOR", "CLIENT PROXY" }

P_ASYNC = { *invocation_alternative* ⟶ *no_result_alternatives*,
invocation_alternative ⟶ *result_alternatives*,
invocation_alternative ⟶ *messaging_selection*,
invocation_alternatives ⟶ *invocation_alternative*,
invocation_alternatives ⟶ *invocation_alternative invocation_alternatives*,

no_result_alternative ⟶ *sync_void_selection*,
no_result_alternative ⟶ *fire_and_forget_selection*,
no_result_alternative ⟶ *sync_with_server_selection*,
no_result_alternatives ⟶ *no_result_alternative*,
no_result_alternatives ⟶ *no_result_alternative no_result_alternatives*,

result_alternative ⟶ *sync_result_selection*,
result_alternative ⟶ *poll_object_selection*,
result_alternative ⟶ *result_callback_selection*,
result_alternatives ⟶ *result_alternative*,
result_alternatives ⟶ *result_alternative result_alternatives*,

poll_object_selection ⟶ "POLL OBJECT" "REQUESTOR" *cp_invocation_option act_option,*
result_callback_selection ⟶ "RESULT CALLBACK" "REQUESTOR" *cp_invocation_option act_option,*
messaging_selection ⟶ *"Messaging",*
messaging_selection ⟶ *"Messaging" act_option,*
fire_and_forget_selection ⟶ "FIRE AND FORGET" "REQUESTOR" *cp_invocation_option,*
sync_with_server_selection ⟶ "SYNC WITH SERVER" "REQUESTOR" *cp_invocation_option,*
sync_void_selection ⟶ *"Synchronous Invocation of Void Operation"* "REQUESTOR" *cp_invocation_option,*
sync_result_selection ⟶ *"Synchronous Invocation with Result"* "REQUESTOR" *cp_invocation_option,*

cp_invocation_option ⟶ *"",*
cp_invocation_option ⟶ "CLIENT PROXY",

act_option ⟶ *"",*
act_option ⟶ "ASYNCHONOUS COMPLETION TOKEN" }

S_ASYNC = { *invocation_alternatives* }

6.4 Core QOC design space for distributed client-side invocations

The core QOC design space for distributed client-side invocations is shown in Figure 7 and Table 2. The core question for deciding for a design of a client-side invocation is: "Which invocation solution should be chosen on client side?" We can see from the grammar overview diagram that there are six alternatives to be considered: the four patterns for asynchronous invocations, the synchronous invocation alternative (we use this alternative for synchronous void invocations and synchronous invocations with a result), and the messaging patterns. If we analyze the four pattern descriptions and the two other alternatives to be considered, we can identify a system of nine forces and consequences. These are shown as criteria on the right hand side of Figure 7.

Let us explain these nine criteria in more detail:

- *Invocation must not block:* A central requirement that differentiates any asynchronous from any synchronous solution is that in asynchronous solutions the client can directly resume with its work, without having to wait for the reply of the server, whereas the synchronous invocation variants blocks.

- *Result required:* For non-`void` remote operations it is necessary to receive a result. Two of the asynchronous invocations patterns, FIRE AND FORGET and SYNC WITH SERVER, are not able to send back a result to the client, all other solutions are able to send back results.

- *Acknowledgment required:* Sometimes clients need to know that an invocation has reached its target. FIRE AND FORGET does not even send back any acknowledgment of the receipt of the message, all

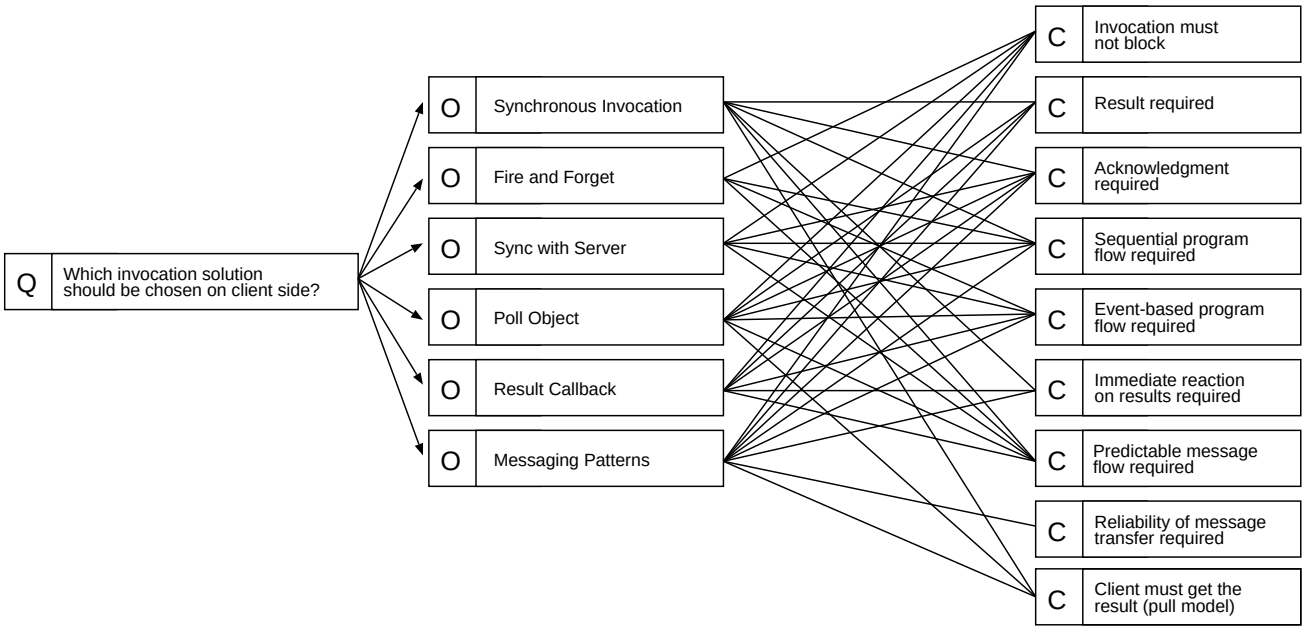

Figure 7. Design space for the asynchronous invocation patterns: core design decision

other solutions are able to send back acknowledgments. Note that a reply (e.g. a result message) is an implicit acknowledgment.

- *Sequential program flow required:* Sometimes clients need to be written with a sequential program flow logic. Then using purely event-based models, such as RESULT CALLBACKS, is not advisable.

- *Event-based program flow required:* If clients are written using an event-based programming style, sequential models, such as synchronous invocations and some POLL OBJECT variants, might be problematic.

- *Immediate reaction on results required:* Some results must be processed as soon as they are available. Synchronous invocations and RESULT CALLBACKS actively inform the client about the result, and are thus advisable in such situations. In contrast, POLL OBJECTS require the client to get the result, possibly later in time. Messaging solutions usually can be configured to support immediate reactions.

- *Predictable message flow required:* Under high volume conditions, messaging might incur problems such as a bursty and unpredictable message flow. Messages can be produced far faster than they can be consumed, causing congestion. Messaging solutions introduce some latency due to the processing of the message queues. Such issues require the messages to be throttled by flow control. This might be undesirable for application areas that require a predictable message flow.

- *Reliability required:* If reliability of message transfer beyond simple acknowledgments is required,

Which invocation solution should be chosen on client side?						
	Synchronous invocation	FIRE AND FORGET	SYNC WITH SERVER	POLL OBJECT	RESULT CALLBACK	Messaging patterns
Invocation must not block		X	X	X	X	X
Result required	X			X	X	X
Acknowledgment required	X		X	X	X	X
Sequential program flow required	X	X	X	X		X
Event-based program flow required		X	X	X	X	X
Immediate reaction on results required	X				X	X
Predictable message flow required	X	X	X	X	X	
Reliability required						X
Client must get the result (pull model)	X			X		X

Table 2. Mapping table for core options and criteria

messaging protocols provide an out-of-the-box solution, whereas all other solutions require hand-crafting of the reliability functions.

- *Client must get the result (pull model):* In many cases, it is not important, whether the client is informed about the result (push model) or is itself responsible for getting the result (pull model). Sometimes, however, clients need to get the result (e.g. at a specific time).

6.5 Adding related patterns and pattern variants to the QOC design space

After we derived the core QOC design space for distributed client-side invocations in the previous section, let us now refine the QOC design space by considering related patterns.

As we want to create a design space that covers the design decision for client-side asynchrony, we can abstract from all pattern relationships that are not directly related to this design issue. For instance, all invocation asynchrony patterns have a dependency to the CLIENT PROXY pattern, but whether a CLIENT PROXY is used or whether the client invokes the invocation asynchrony pattern implementation directly (using

the REQUESTOR pattern), plays no role in the selection of an asynchrony pattern. Thus we abstract from the relation to these two patterns, CLIENT PROXY and REQUESTOR, in the QOC design space.

However, other questions – those that arise from the core design decision – are of central importance to be considered in order to come to a sound design. These follow-on decisions are depicted as extensions to the core design space in Figure 8. In particular there are the following important design decisions:

- If a result is sent back as a reply to an asynchronous invocation, it is – in contrast to synchronous invocations – possible that the client waits for more than one result. Thus we must identify the client-side object to which the result should be dispatched. An efficient and flexible solution for this problem is the ASYNCHRONOUS COMPLETION TOKEN (ACT) [46] pattern, which is an option in the pattern language grammar overview diagram. Alternatively, if the communication protocol allows for identifying the receiver based on the connection, we can instantiate one receiver object (POLL OBJECT or RESULT CALLBACK) per asynchronous invocation. This incurs some memory overhead. Finally, if the client is also a server, we can make the receiver on client side a remote object and send the result as a new invocation.

- SYNC WITH SERVER requires an acknowledgment sent from server to client. Basically, we explained all options for sending an asynchronous result back before – with their respective consequences. In addition we can use a synchronous invocation to send the acknowledgment. That means, the client sends the invocation to the server and blocks. The server sends the acknowledgment immediately and then processes the messages later asynchronously, but sends no result for the message back.

- Somehow we need to realize the asynchrony of an asynchronous invocation on client side; that is, we need to map the synchronous client program to asynchronous invocations and asynchronous replies. If the operating system and the middleware environment support asynchronous I/O, we can use this facility, which has little overhead. If only synchronous communication is possible, we can spawn a thread (or process) for each invocation. This incurs a thread (process) creation overhead and scalability might be influenced negatively, when a large number of invocations is running in parallel and thus many threads (processes) need to be managed. The pattern POOLING [32] can be used to limit the number of threads and minimize the thread creation overhead.

- If we decide for a messaging solution, we need to decide for a pattern for asynchronous result handling. Here, one of the two patterns POLL OBJECT or RESULT CALLBACK can be used. (Note that the respective messaging patterns are called POLLING CONSUMER and EVENT-DRIVEN CONSUMER [29]). Alternatively, we can block on the message queue. This solution has the same consequences as using a synchronous invocation. The criteria for the three options are not depicted in Figure 8 because we have shown them already in Figure 7.

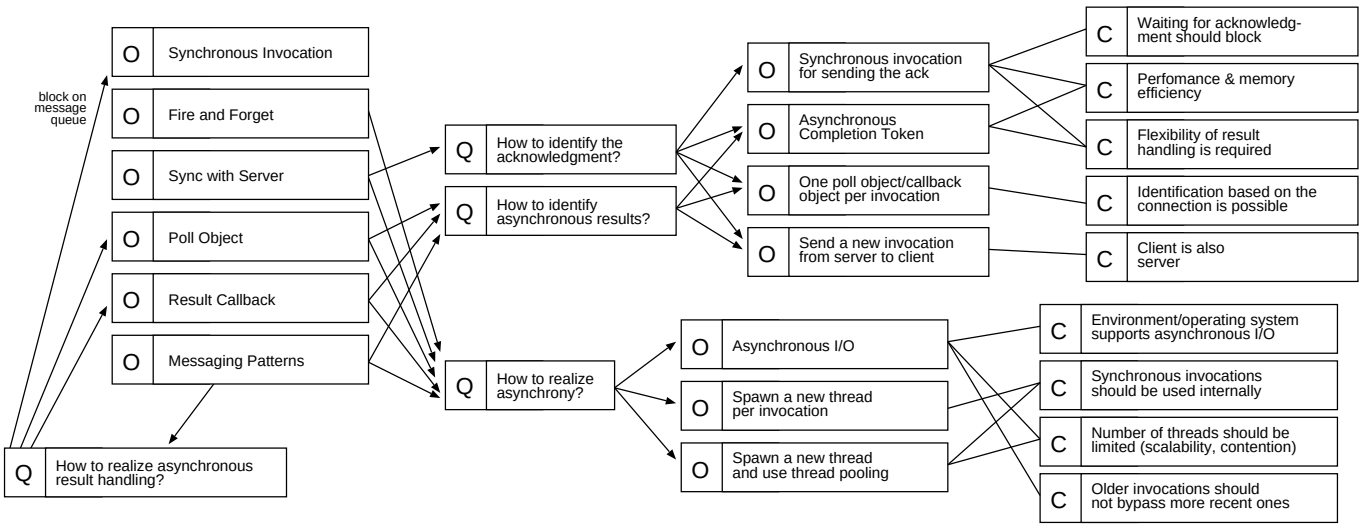

Figure 8. Design space for the asynchronous invocation patterns: links to pattern variants and other patterns

7 Validation and experiences in using the approach

To validate our approach, we have used and refined the approach presented in this article excessively in projects and case studies over the last couple of years. We summarize these practical experiences in this section. Please refer to the referenced papers for a detailed discussion of the projects and case studies.

In some early projects (both industrial and research systems) we used the approach intuitively for developing an understanding of the relevant patterns and domains, and to communicate the results of a pattern-based design to project members. The experiments in these projects helped us to refine the approach. Those experiences also revealed that it is useful to integrate the coarse-grained, structural pattern language grammar approach with the more fine-grained QOC-based design space analysis, because the QOC-based design space analysis reveals semantics of a design decision that cannot be seen in the pattern language grammar.

In addition, the combination of the two approaches is also useful for communication of design decisions: for communicating design selections and design rationale to stakeholders not involved in the design decision – especially non-technical stakeholders like managers, customers, or end-users – it is useful to present the more holistic overviews of the pattern language grammar overview diagrams, whereas for technical stakeholders the much more detailed QOC-analysis is useful for leveraging a technical discussion.

7.1 Industrial Case Study 1: Reengineering a document archive system

In a reengineering project for a large-scale, industrial document archive system (see [24]) we were confronted with a monolithic architecture to be migrated to a “modern” component-oriented design. We applied a pattern-based approach, following the patterns for flexible, component-oriented architectures described in

[55] and related patterns [21, 12, 46].

We informally documented the relationships among these patterns using applicable pattern sequences. In addition to early versions of our pattern language grammar overview diagrams, we used tables similar to the ones derived by our QOC-approach for systematic design decisions. These illustrations and tables were also used in technical discussions with the technical staff of the company (who had not read the pattern texts in detail), and for company-internal presentations of the results. Our initial re-design recommendations were based on a thorough code review, which was enough for getting a first impression of the system, but not for discussing design decisions and presenting the results of these discussions.

In this project, the pattern language grammar overviews, design space illustrations, and tables were used for enriching the technical design decisions with the domain knowledge of the stakeholders in the company. Also the approach was used to present the technical knowledge in the patterns to the companies' staff. In other words, we used our approach to bridge the gap between our technical expertise for the pattern domain and the stakeholders' domain knowledge about document archive systems. Please note that it was not an option in this project to a priori require the company staff to read through the complete pattern material; the visual pattern representations of our approach were accepted, though, and have leveraged an interest in the patterns.

The case study has shown for one non-trivial case that it is possible to use the approach to support an iterative design process. It has also shown that the approach can be used to communicate complex pattern material and design rationale both to technical and non-technical project members.

7.2 Industrial Case Study 2: Reengineering a document archive system

In another industry project we were involved in the design for a product line for multimedia home platform (MHP) [19] terminals – i.e. digital television receivers like settop-boxes (see also [22, 23]). The difficulty in this project was the design of flexible runtime variation points in the constrained environment of the settop-box and the television broadcast. To design the product line architecture's variation points, we used the patterns from [55] and [53, 51], as well as related patterns from [21, 12].

In this project, the main use of our approach was to balance the quality goals of the individual patterns – we had a working solution on a personal computer in mind but had to adapt it to the particularities of the platform. We thus used an approach where we used a pattern language grammar analysis that was annotated with the impacts on quality attributes, as described in Section 4. This was very useful for combining multiple patterns from different sources domain-specifically. The resulting view on the pattern languages and respective case studies are described in [22].

The case study has shown for one non-trivial case that it is possible to use the approach to organize pattern material and reduce the complexity of the design decisions.

7.3 Research Prototypes as Case Studies

The industrial cases explained before have been performed in the context of existing systems or platforms and in projects with clear design goals. In addition to those larger industrial case studies we used the approach proposed in this article in a number of research prototypes to design architectures with patterns from scratch. This work was done to validate that the approach is also useful for organizing pattern material in a less clear and more innovative design situation. That is, we used the approach to apply a systematic kind of pattern-based prototyping.

The goal was always to realize a specific functionality with specific quality attributes by experimenting with the patterns and applying them incrementally (and removing not-working solutions again). Using an unsystematic approach, such a proceeding can fail dramatically, because the wrong patterns are used for a task, leading to an incremental downgrade of the architecture instead of a refinement, or very long development times.

Instead our goal was to come more quickly to a working, innovative architecture by reusing the design knowledge in the patterns. We applied this approach in the following research projects:

- We used our pattern language grammars and design spaces for patterns from [55] and [53, 51], as well as related patterns [21, 12], in the implementation of the programming language Frag and its SPLIT OBJECT framework [57, 56, 59].

- We used the pattern language grammar and design spaces for the remoting patterns presented before in two projects:

  - SAIWS is an asynchronous invocation framework for Web services (see [54, 61, 60]).

  - Leela (see [58, 52]) is a Web services framework that provides a loosely coupled infrastructure for remote object federations.

In all those projects we developed an innovative research idea with specific goals in mind first and then used our approach to come to an implementation that fulfills these goals but does not re-invent the wheel. Because the initial ideas were quite “open”, it was very helpful that our approach provides a way to reduce the number of options in each design step, without removing viable options from the overall space of possible solutions.

The project experiences show at least that innovative designs following the approach are possible. The design and development times fell below the initial expectations, which is also encouraging.

7.4 Research Project: Technology comparison of AOP frameworks

In [53] we illustrate how the pattern language from [51] is applied in popular aspect languages and aspect composition frameworks using more simple pattern overview diagrams [3, 4] and feature-oriented domain analysis (FODA) [31]. This work can be seen as a direct predecessor of the work presented in this article. The option types in the FODA diagrams can roughly be mapped to the elements of our pattern language grammar overview diagrams.

The goal in this project was a bit different to the problem of pattern selection though: we used the pattern language overview diagrams for a comparison of different aspect composition frameworks. That is, the goal of this project was a technology comparison.

Even though this is not the main goal of our approach, we still consider this project an important experience, as it shows that the approach can be used for other tasks than pattern selection: in this project it has helped us to perform a technology comparison and to bring the results across to a broader research audience.

7.5 Validation of support for understanding by non-experts

The approach presented in this article is a supplemental technique to the pattern languages concept. A main goal is to support situations in which the pattern language itself is highly complex, the pattern material is not organized well, or patterns from multiple sources must be organized. We claim that our approach works better than simply scanning the pattern texts.

The case studies and projects explained before show that the approach can be applied successfully, but they do not validate that the approach helps non-experts in the pattern domain to understand the patterns better than they would without the approach. This can hardly be validated in a typical industrial or research project, because here it is usually required that experts for the patterns are in the project.

We thus validated this claim in a student group of five students[6]. The task of the group was to conduct a study in the area of security patterns. We have chosen this domain because at the time of this study, the field of security patterns was not completely covered by patterns, the existing patterns were incomplete and of varying quality, there was no comprehensive pattern language for security patterns, and many links between the patterns were missing. In other words, the ideal situation that good and coherent pattern descriptions are existing, was not given in this area.

All five students have visited a patterns introduction course (mainly covering the GoF patterns [21]) and a one semester (2 hours/week) course about network security, before they joined the group. The project lasted for one semester.

[6]This study was conducted by a seminar group at the Vienna University of Economics in the winter semester 2003/2004. The participants of that group, supervised by Mark Strembeck and Uwe Zdun, were: Roland Berger, Jürgen Hasiner, Mareike Kukacka, Wilhelm Nowak, and Leila Saleh.

We instructed the students to first identify the existing patterns in the security domain, summarize them, and explain the relationships to other patterns. The students studied a body of literature containing 49 patterns. We analyzed the resulting document produced by the students after half the semester: more than 70% of the pattern summaries were clearly wrong. Especially there were many mistakes in the quality goals/forces of the patterns. Less than 50% of the pattern relationships mentioned in the pattern texts were correctly identified. The students have identified only 2 pattern relationships that were not documented in the patterns, and none of them was correct. The students have not identified any of the major gaps in the security pattern landscape.

Next, we introduced the students to the approach described in this article: they first identified possible sequences and gaps – in a graphical diagram similar to the pattern language grammar overview diagrams used in this article. Next, they mapped the patterns to quality goals/forces described in the pattern texts using tables following those used for the results of our QOC-analysis. The result of this process was a pretty comprehensive overview of the existing pattern material in the security domain. We also analyzed the final resulting document, delivered at the end of the semester: less than 10% of the pattern summaries had mistakes, especially the interpretation of quality goals/forces has significantly improved. Almost all documented relationships of the patterns were identified by the students, and the students correctly identified 8 pattern relationships that were not documented in the pattern texts. Also, the students were able to organize the security pattern landscape in topic clusters. A subsequent comparison to the security literature has revealed gaps in the security pattern landscape.

This experiment shows that the approach described in this article can significantly improve a non-expert's understanding of incomplete pattern material from multiple sources. In our interviews with the students, they confirmed that the more structured and systematic approach helped them to improve the results and raised their confidence in their results.

8 Related work

8.1 Selecting patterns

To describe how to select and apply software patterns is one of the main goals of our approach. We already discussed the related work on pattern languages, which is probably the 'most practical' other approach addressing this goal (see for instance the pattern languages in [46, 32, 49, 50, 53] and Alexander's work [2, 5, 3, 4]). Throughout this article we discussed open issues when selecting patterns using existing software pattern languages, and explained that our approach supplements the pattern language concept.

A number of other supplements to pattern languages have been proposed, most of them focusing on enriching the pattern language with examples, for instance, to show how they can be applied in a particular domain. In [46] general patterns of networked and concurrent objects are supplemented with a more domain-specific

pattern language on middleware design. In [32] a number of design case studies are given for the pattern language presented, and in [49, 50] technology projections (illustrating the use of the pattern language in a popular technology) are provided. Note that these approaches are – in contrast to our approach – parts of the pattern language descriptions and do not represent software engineering approaches for aiding the application of patterns. We also provided a number of case studies for general purpose patterns from [21, 12, 46] to illustrate how they can be applied together in a particular domain: Web servers [41] and XML/RDF parsing and interpretation [42].

Ziegler [62] proposes a design space-based approach for mapping the dimensions "navigation structure" and "content structure" onto each other. The goal is to identify navigational HCI patterns using a systematic approach. We did not follow this "dimensional" approach to design spaces, but did instead follow the QOC approach [36], because an n-dimensional matrix of design dimensions limits the solution space to these dimensions. When the goal is pattern selection in a pattern language this approach is dangerous: we might not be able to use important sequence steps that lie outside of the design space dimensions. Ziegler's approach is useful for mining undocumented software patterns though, because the dimensional approach clearly reveals gaps in the pattern landscape. For all specific combinations of the options in the two dimensions that are not covered by patterns yet, you can seek for this combination in existing systems, and if found multiple times, it is a candidate for a pattern.

Braga and Masiero [10] propose a systematic approach to identify hot spots [44] in a pattern language. They use different types of hot spots that can be identified from the information present in the elements of each pattern of the pattern language. The approach thus uses a similar approach to our approach for understanding a pattern language: elements are identified from the pattern texts and mapped to a systematic framework of hot spot types. The goal of the approach is different though: it aims at reducing the complexity of object-oriented framework development, and thus introduces hot spot types that can be found in object-oriented frameworks rather than general pattern language relationships (as used in our approach).

8.2 Formalization approaches and language support for patterns and styles

Porter, Coplien, and Winn [43] explain the application of Alexander's concept for composing pattern languages using pattern sequences (see [3, 4]) to the software domain using two example pattern languages from the software domain. A formalism for representing a sequence as a single, totally ordered set of patterns is proposed. A pattern language is consequently seen as a partially order set (poset).

Henney extends the approach of Porter, Coplien, and Winn using a grammar-oriented approach for deriving sequences [26]. Our approach extends Henney's approach with graphical pattern language grammar overviews which can be used to provide tool support for grammar generation, annotations for quality goals, and a subsequent QOC design space analysis.

Coplien and Zhao [15] propose a formalism for software patterns using group theory and symmetry concepts. They argue that many programming languages provide constructs to support symmetry, i.e. invariant changes, and that software patterns describe situations of symmetry breaking, i.e. situations in which the design of the programming language is not adequate for a design problem and thus a pattern must be applied. The proposed pattern formalism is useful to explain software patterns as a design discipline and to relate them to Alexander's work. It does not specifically aim at pattern selection or the design process with patterns, though, which is the aim of our approach.

A number of approaches have been proposed for the formal specification of software patterns (see for instance [18, 39, 48, 37]). These approaches, however, have not gained much momentum in recent years mainly because of their complexity and the resulting limitations regarding their practical use. In contrast to the formalisms described before, these approaches are not explicitly derived from Alexander's work on pattern languages, and they have not been used for architectural patterns or whole pattern languages, but just for some isolated patterns from the GoF book [21]. Similar issues arise for approaches proposed for language support for design patterns [40, 9] or implementations of patterns as aspects [25, 28]. These approaches make patterns first-class entities of the language or aspect framework, which means that the formal dependencies can be used as a guide for selecting a pattern-based solution. This works only at the level of singular patterns and their participants. The approaches provide only little guidance in how to design complex software architectures.

Architectural styles consist of system composition patterns and constraints on architectural elements, which are targeted at families of systems with shared characteristics [1, 38]. Some studies of styles focus on the component connectors as the key elements of architectural styles (see e.g. [47]). Thus styles provide some dedicated guidance on how to assemble an architecture from building blocks. Some approaches exist that further help in the composition of architectural styles. For instance, the Alfa framework [38] allows for systematically composing architectural styles from a number of architectural primitives.

Similar to pattern formalization attempts, there have been formalization attempts for architectural styles (see e.g. [1]). These have similar limitations as the approaches proposed for patterns: they do not consider the composition of larger architectures from primitive building blocks [38].

8.3 Other design space approaches in software engineering

A number of approaches utilize the design space approach in software engineering. Let us briefly explain how these approaches relate to our approach.

De Bruin, van Vliet, and Baida [11] propose an approach for mapping the required features of the system, modeled in a so-called feature space, to the elements of the so-called solution space that have an effect on the realization of the respective features. The approach also provides a kind of design space, represented as a

feature-solution graph. The approach's main objective is to reuse and select solution elements by documenting feasible architectures. The approach also serves as a basis for tracing design decisions. In contrast to our approach, this approach does not explicitly explain how to map the solution space's elements to larger software architectures and the relationships among architectural elements. This important aspect is an inherent part of the pattern descriptions used in our approach.

Baum, Becker, Geyer, and Molter [8] use a design space-based approach to map requirements to components. The approach aims at the reuse of components in similar application scenarios. However, no "foundation" for finding the relevant requirements (like the patterns in our approach or the feature space in [11]) is used, and no iterative analysis process is proposed e.g. to identify conflicts. Instead requirements are captured using the properties of existing components. Thus only component selection is supported, but no further architectural decision or documentation.

There are a number of requirements engineering approaches that loosely resemble the design space decomposition method, used as part of our approach. These approaches organize the functional and non-functional requirements of a system. For instance, in goal-oriented requirements engineering [33], goals capture the objectives a system should achieve. Goals can be related and refined in goal graphs. Similarly, soft goal contribution graphs [13] decompose soft goals (e.g. quality attributes) of a system. These approaches, however, work at a different level of abstraction than our approach: their aim is to organize concrete system requirements systematically, whereas our approach organizes patterns (i.e. recurring solutions to recurring problems) and pattern languages. The (soft) goals used for decomposition of the requirements loosely resemble the forces of the patterns, which our approach uses as criteria, but again concrete goals of concrete systems are working at a different abstraction level than the forces of patterns (which can be used over and over again). A requirements/goals decomposition, however, can be quite useful in combination with our approach: the decomposed requirements/goals of a concrete system can be matched to the decision guides provided by the pattern language grammars and design spaces in order to select the patterns as solutions supporting specific goals.

9 Conclusion

We presented an approach to systematize the selection process of software patterns in pattern languages. The approach is feasible because the documentation of the pattern language grammar and the design spaces has to take place only once (for instance performed by the pattern language author or during the first design with a pattern language), and then the pattern language grammar and the design spaces can be applied for multiple systems (of course they can be refined if information is missing). In contrast to former approaches (see the discussion in Section 8), our approach is capable to use patterns from many pattern sources, deal with architectural patterns (not only GoF design patterns), and provide domain-specific design decisions based on

quality goals and forces. The options in the selection process are first reduced to pattern sequences using a formal pattern language grammar, and then to a detailed view on a single design decision in a QOC design space. Thus, we significantly reduce the complexity of the pattern selection process – from the selection among the whole patterns and potential relationships in a number of pattern catalogs or pattern languages to an ordered, systematic sequence of documented design decisions. Throughout the decision process, our approach explicitly considers the quality goals (forces) of the patterns and their variants.

It is important to note that our approach is focused on the abstraction level of patterns and pattern languages (as in the examples in this article). Thus it should not be directly applied to concrete system designs without using the pattern language abstraction. That is, it should not be used for organizing concrete system requirements, because this would entail the danger that other possible design solutions than those documented as patterns are not considered, or that patterns as solutions are applied "blindly" leading to pattern overkill and abuse. It would be interesting to investigate in how far our approach can be extended with requirements engineering approaches, such as goal-oriented requirements engineering [33], to systematically map concrete requirements to pattern forces.

As discussed in Section 7, our approach can also be used for other goals than only pattern selection. We mentioned projects where our approach helped in the communication with (non-technical) stakeholder, for understanding pattern material, and for technology comparison/evaluation. We envision further application areas for the approach: for instance, the pattern language grammars and design spaces can potentially be used as an input for model-driven tools. As a future work, we want to examine these and other further application areas for the approach in more depth. We also plan to investigate the use of the approach for pattern language composition.

Acknowledgments

I wish to thank James Coplien for providing detailed comments on an earlier draft of this manuscript. Thanks to Kevlin Henney for pointing out that the state patterns are a good example for the XOR-alternatives. Many thanks to all project members in the projects described in Section 7.

References

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.

[2] C. Alexander. *The Timeless Way of Building*. Oxford Univ. Press, 1979.

[3] C. Alexander. *The nature of order*. Center for Environmental Structure, 2004.

[4] C. Alexander et al. Patternlanguage.com. http://www.patternlanguage.com, 2001.

[5] C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language – Towns, Buildings, Construction*. Oxford Univ. Press, 1977.

[6] P. Avgeriou and U. Zdun. Architectural patterns revisited – A pattern language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPloP 2005)*, Irsee, Germany, July 2005.

[7] L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice (2nd edition)*. Addison-Wesley, Reading, USA, 2003.

[8] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In *Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, pages 159–167, Schaumburg, Illinois, 2000. IEEE Computer Society.

[9] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[10] R. Braga and P. Masiero. Finding frameworks hot spots in pattern languages. *Journal of Object Technology*, 3(1):123–142, 2004.

[11] H. Bruin, J. Vliet, and Z. Baida. Documenting and analyzing a context-sensitive design space. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 127–141. Kluwer, B.V., 2002.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[13] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, Boston, 1999.

[14] J. Coplien. *Software patterns, SIGS Management Briefings*. SIGS Books & Multimedia, 1996.

[15] J. Coplien and L. Zhao. Symmetry breaking in software patterns. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.

[16] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction (Second Edition)*. Prentice Hall, 1998.

[17] P. Dyson and B. Anderson. State patterns. In *Proceedings of First European Conference on Pattern Languages of Programs (EuroPlop 1996)*, Irsee, Germany, July 1996.

[18] A. H. Eden and Y. Hirshfeld. LePUS – symbolic logic modeling of object oriented architectures: A case study. In *Second Nordic Workshop on Software Architecture - NOSA'99*, Ronneby, Sweden, April 1999.

[19] ETSI. MHP specification 1.0.1. ETSI standard TS101-812, October 2001.

[20] M. Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] M. Goedicke, C. Koellmann, and U. Zdun. Designing runtime variation points in product line architectures: Three cases. *Science of Computer Programming*, 53(3):353–380, 2004.

[23] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architectures. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS 2002)*, pages 384–396, Montpellier, France, June 2002. LNCS 2425, Springer Verlag.

[24] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1):1–30, 2002.

[25] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In C. Norris and J. Fenwick, editors, *Proceedings of the 17th ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, Nov. 2002. ACM Press.

[26] K. Henney. Context encapsulation – Three stories, a language, and some sequences. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPloP 2005)*, Irsee, Germany, July 2005.

[27] Hillside Group. A pattern definition. http://hillside.net/patterns/definition.html, 2004.

[28] R. Hirschfeld, R. Lämmel, and M. Wagner. Design patterns and aspects – modular designs with seamless runtime integration. In *Proc. of the 3rd German GI Workshop on Aspect-Oriented Software Development, Technical Report, University of Essen*, Mar. 2003.

[29] G. Hohpe and B. Woolf. *Enterprise Integration Patterns.* Addison-Wesley, 2003.

[30] ISO. ISO/IEC 9126: Quality characteristics and guidelines for their use, 1991.

[31] K. Kang, S. Sholom, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute (SEI), 1990.

[32] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. J. Wiley and Sons Ltd., 2004.

[33] A. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–261, Toronto, Canada, August 2001.

[34] J. Lee and K. Lai. What's in design rationale? *Human-Computer Interaction*, 6(3–4):251–280, 1991.

[35] A. MacLean and D. McKerlie. Design space analysis and use representations. In *Scenario-based design: Envisioning work and technology in system development*, pages 183–207. John Wiley & Sons, 1995.

[36] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options, and criteria: Elements of design space analysis. *Human-Computer Interaction*, 6(3–4):201–250, 1991.

[37] J. Mak, C. Choy, and D. Lun. Precise modeling of design patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 252–261, Edinburgh, Scotland, United Kingdom, 2004. IEEE Computer Society.

[38] N. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, Helsinki, Finland, 2003. ACM Press.

[39] T. Mikkonen. Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, Kyoto, Japan, 1998. IEEE Computer Society.

[40] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, pages 1–14, San Diego, California, USA, May 1999.

[41] G. Neumann and U. Zdun. High-level design and architecture of an HTTP-Based infrastructure for web applications. *World Wide Web Journal*, 3(1):13–26, 2000.

[42] G. Neumann and U. Zdun. Pattern-based design and implementation of an XML and RDF parser and interpreter: A case study. In *Proceedings of the 16th European Conference on Object-Oriented Programming, LNCS 2374, Springer Verlag*, pages 392–414, Malaga, Spain, June 2002.

[43] R. Porter, J. Coplien, and T. Winn. Sequences as a basis for pattern language composition. *Science of Computer Programming*, 56(1–2), 2005.

[44] W. Pree. *Design Patterns for Object-Oriented Software Development.* ACM Press Books. Addison-Wesley, 1995.

[45] D. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *25th International Conference on Software Engineering*, pages 694–704, May 2003.

[46] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[47] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.

[48] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Proceedings of the 26th International Conference on Software Engineering*, pages 666–675. IEEE Computer Society, 2004.

[49] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns.* Pattern Series. John Wiley and Sons, 2004.

[50] M. Völter, A. Schmid, and E. Wolff. *Server Component Patterns – Component Infrastructures illustrated with EJB.* J. Wiley and Sons Ltd., 2002.

[51] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003)*, pages 581–616, Irsee, Germany, June 2003.

[52] U. Zdun. Loosely coupled web services in remote object federations. In *Proceedings of the Fourth International Conference on Web Engineering (ICWE'04)*, volume 3140 of Lecture Notes in Computer Science, pages 118–131, Munich, Germany, July 2004. Springer-Verlag, Berlin.

[53] U. Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings Software*, 151(2):67–83, April 2004.

[54] U. Zdun. Simple asynchronous invocation framework for Web services (SAIWS). http://saiws.sourceforge.net, 2004.

[55] U. Zdun. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004)*, Irsee, Germany, July 2004.

[56] U. Zdun. Using split objects for maintenance and reengineering tasks. In *8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 105–114, Tampere, Finland, Mar 2004.

[57] U. Zdun. Frag. http://frag.sourceforge.net/, 2005.

[58] U. Zdun. Leela. http://leela.sourceforge.net/, 2005.

[59] U. Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems and Structures: An International Journal, Elsevier (accepted for publication)*, 2005.

[60] U. Zdun, M. Voelter, and M. Kircher. Design and implementation of an asynchronous invocation framework for web services. In M. Jeckle and L. Zhang, editors, *Web Services – ICWS-Europe 2003, International Conference ICWS-Europe 2003, Erfurt, Germany, September 23-24, 2003, Proceedings*, volume 2853 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2003.

[61] U. Zdun, M. Voelter, and M. Kircher. Pattern-based design of an asynchronous invocation framework for web services. *International Journal of Web Services Research*, 1(3):42–62, July–Sept 2004.

[62] J. Ziegler. Structure mapping patterns – a family of patterns for user and group navigation. In *Proceedings of the CHI 2004 Workshop on Human-Computer-Human Interaction Patterns*, Vienna, Austria, April 2004.