SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1121

The use of pattern participants relationships for integrating patterns: a controlled experiment

Ahmad Waqas Kamal [1,*,†], Paris Avgeriou [1] and Uwe Zdun [2]

[1] *Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands*
[2] *Faculty of Computer Science, Software Architecture, University of Vienna, Vienna, Austria*

SUMMARY

Architectural patterns are often applied in combination with related patterns within software architectures. The relationships among architectural patterns must be considered when applying a combination of patterns into a system; for example the way the Model-View-Controller uses the Observer pattern to implement the change propagation mechanism needs to be carefully designed. However, effective integration of architectural patterns within software architectures remains a challenging task. This is because the integration of any two architectural patterns can take several forms. Furthermore, existing pattern languages define generic and abstract relationships between architectural patterns without going into detail about associations among the participants of architectural patterns. In this paper, we propose to address the pattern integration issue by discovering and defining a set of pattern participants relationships that serve the purpose of effectively integrating architectural patterns. Our findings are validated through a controlled experiment, which provides significant evidence that the proposed relationships support inexperienced designers in integrating patterns.

Received 30 January 2011; Revised 4 July 2011; Accepted 14 August 2011

KEY WORDS: architectural patterns; pattern languages; pattern relationships; modeling

1. INTRODUCTION

Over the past decade, architectural patterns have increasingly become an integral part of software architecture design practices [4]. Architectural patterns often specify solutions to recurring design problems by describing essential components, their responsibilities and relationships [3]. In practice, architectural patterns are seldom applied in isolation to a software architecture, as a single pattern may not suffice to fully resolve a design problem at hand. For instance, the Client–Server and Broker patterns are often used in combination to design distributed systems [4]. It is commonly accepted that patterns are somehow connected to each other giving them the potential to solve larger design problems [36] [3].

There have been several approaches in the patterns community that aggregate a number of patterns that define, to some extent, relations between those patterns. We characterize these approaches into four major categories:

- *Architectural pattern languages* define a network of patterns, connected through specific relationships [2], forming a graph of nodes where each node represents a pattern [3]. Pattern languages are the most common and well-known form, used by the software patterns community for defining relationships among architectural patterns. Several pattern languages have been documented in the literature, for example, pattern languages for distributed computing

*Correspondence to: Ahmad Waqas Kamal, Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands.
[†]E-mail: a.w.kamal@rug.nl

[3], domain specific pattern languages [4], architectural view-specific pattern languages [2], and so on.

- *Pattern catalogs*, which may be organized according to different categories like patterns for object-oriented frameworks [8], patterns for enterprise computing [6], patterns for security [34], and patterns for user interface design [5]. Pattern catalogs often list patterns alphabetically or according to some categorization, but they do not always describe the relationships between patterns. Patterns in a pattern catalog do not form a pattern language because their contexts do not weave them together [3].
- *Pattern compounds* capture recurring use of a set of patterns that are often used as a single decision to solve a recurring design problem [3]. For instance, the Batch Method [3] and Iterator [3] are often used together leading to the commonly used term BatchIterator.
- *Pattern sequences* document the possible successive application of patterns for designing software architectures [7]. For instance, first the Iterator pattern can be applied to provide the notion of a traversal position, then, the Batch Method is applied to define the style of access on a component [36].

However, these approaches do not support the effective integration of architectural patterns within software architecture, for two main reasons:

- Existing pattern languages, pattern compounds and pattern sequences document associations between patterns at a generic level but do not go into details concerning the relationships between the pattern participants[‡]. For instance, a pattern language may suggest that the communication between Client and Server [4] can be mediated through a Broker [4] and hidden by a Proxy [3]. But it does not elaborate on how the participants of these three patterns will collaborate in order to achieve the envisioned goal. The relationships among architectural pattern participants are important to effectively address the extended set of requirements that mandate the combination of two or more patterns. The details of such relationships between participants concern, for example, how participants of related patterns overlap, interact, or override other participants in the resulting software architecture.
- The integration of two selected architectural patterns does not always result in one particular solution but leads to several possible design solutions depending on the system context at hand. In other words, pattern-to-pattern relationships are not always fixed but may entail a great deal of variability. For instance, to model interactive applications, the Model-View-Controller (MVC) [4] and Layers [4] patterns can be combined in several different forms. In the two-tier layered variant, the presentation layer consists of the View and Controller participants whereas the application logic layer owns the Model participant. However, in the three-tier application architecture, the View may correspond to user interface layer, the Controller corresponds to business layer, and the Model corresponds to data logic layer. This variability in pattern combinations is currently not explicitly addressed by existing pattern languages.

In this paper, we aim at supporting architects and designers in integrating patterns by using relationships at the level of pattern participants. The relationships were discovered after reviewing architectural patterns modeled in several industrial software architectures and pattern integration examples documented in the literature. The notion of pattern participants relationships was first proposed by us in [23]. Our current work provides detailed documentation of discovered relationships and is supported by evidence in a controlled experiment. The documentation of pattern relationships at the pattern participants level contains several possible associations between architectural patterns, which correspond to alternative design solutions. Furthermore, we have validated our approach through a controlled experiment where we investigated the effectiveness of using pattern participants relationships in integrating architectural patterns. We advocate that the use of pattern participants relationships: (i) leads to appropriate integration of architectural patterns; (ii) improves design comprehensibility; (iii) helps architects to better document design decisions; and (iv) assists in decomposing software architectures.

[‡]By the term pattern participants, we refer to the architectural elements within the solution of architectural patterns, for example, pipes and filters are pattern participants of the solution specified by Pipes and Filters pattern.

DOI: 10.1002/spe

The remainder of this paper is structured as follows: In Section 2, we describe related work in the field of architectural patterns relationships. Section 3 describes our effort for identifying relationships among architectural patterns participants and lists a set of pattern participants relationships. Section 4 provides the description of the controlled experiment that was conducted to test the effectiveness of using pattern participants relationships for integrating architectural patterns, and Section 5 documents the execution of the experiment. Section 6 presents statistical results from the controlled experiment. Section 7 interprets qualitative data gathered after the experiment and discusses the possible threats to the validity of the results. The study is concluded in Section 8.

2. RELATED WORK

In this section, we discuss some of the work done by other researchers in the area of relating architectural patterns.

Zimmer [39] classifies the relationships between several design patterns. He categorizes pattern relationships into three categories, namely *uses* where a pattern A must use a pattern B, *similar* where a pattern A is similar to pattern B, and *combined* where two patterns can be applied as one design solution to a design problem. However, the classification of relationships addresses only design patterns where each pattern is represented as a single unit or object. His work addresses the abstract relationships between patterns and the pattern links in the context of a pattern language. Our work is aimed at documenting relationships between the participants of architectural patterns and the way patterns are combined in real software architectures. Thus, we provide a more fine-grained approach to associate patterns using pattern participants relationships for effectively combining architectural patterns.

Fayad *et al*. [20] have proposed the concept of *stable software patterns*. The process of developing stable software patterns involve four main steps: *developing stable patterns*, *documenting stable patterns*, t*esting–validating stable patterns*, and *applying stable patterns*. For each of the four steps,

Table I. Overview and comparison of related work.

Approach	Granularity	Application	Scope	Contribution
Relationships categorization [39]	Classes, objects	Design patterns	Object-oriented system design	Abstract relationships between design patterns
Software stability concerns [20]	Design activity nodes	Processes	Software stability concepts	Stable software development
Pattern relationship types [11]	Architectural elements	Architectural and design patterns	General	Generic pattern language
Architectural concerns [5] [34]	Components and connectors	Architectural patterns	Quality-attributes driven design	Quality-attributes specific pattern languages
Grouping patterns based on problem-domain [3]	Components and connectors	Architectural patterns	Distributed systems architecture design	Domain-specific pattern language
Formal approaches to modeling patterns [13] [32]	Classes, nodes, objects, function calls	Design patterns	General	Ontology-based pattern modeling
Empirical research [15] [19]		Architectural patterns, design patterns	General	Statistical results for applying patterns
Pattern participants relationships	Components as participants of architectural patterns	Architectural patterns	General	Relationships between pattern participants and statistical results for combining patterns

DOI: 10.1002/spe

there are different sets of patterns that interact together to accomplish the goal of the step. However, their work is more focused on software stability concepts [27]. Our contribution differs from this work as our focus lies in discovering relationships between patterns at a rather detailed level of abstraction, that is, between the participants of patterns in real software architectures, which are not addressed before. In relation to their work, if more relationships are identified, our work can be used in the *applying stable patterns* step of their approach.

Buschmann *et al*. [11] document three kinds of pattern relationships: pattern complements, where one pattern competes with another by providing an alternative solution to a specific problem; pattern compounds that relate two or more patterns for their use as a single decision to solve a design problem; and pattern sequences that describe the progression of patterns by having predecessor patterns forming part of the context of successive patterns. However, these types of pattern relations are defined at an abstract level and do not provide concrete relationships between the participants of related patterns. Our work aims to fill this void by addressing pattern relationships at a detailed level of granularity, that is, at the level of pattern participants.

Some work has been done on proposing pattern languages that address specific architectural concerns such as pattern languages for usability [5], pattern languages for concurrency [34], pattern languages for performance-critical systems [3], and so on. However, these languages provide relationships that address specific architectural concerns they relate to and do not address the relationships among participants of related architectural patterns. For instance, the 'event handling' relationship [3] between the Reactor and Leader–Follower patterns does not specify the participants of the two patterns that need to be combined for designing an event handling solution.

Buschmann *et al*. [3] present a pattern language for distributed computing that includes 114 patterns grouped into 13 problem areas. The problem areas address technical topics related to building distributed applications, for example, Event Demultiplexing, Concurrency, Synchronization, and so on. This pattern language serves as an overview of the selection and use of related architectural patterns to solve design problems in specific problem areas. However, the language in itself presents architectural patterns as components, objects, and entities linked through generic textual expressions. For instance the MVC has a "request handling" relationship with the Command, Command Processor, Application Controller, and Chain of Responsibility patterns. Similar to the previous cases, relationships between participants are not defined.

In our previous work [2], we have documented relationships among architectural patterns in different architectural views that show specific aspects of systems like data flow view, interaction decoupling view, and so on. We had focused on providing rich pattern-to-pattern relationships (e.g. communication between Layers may use Pipes and Filters) and not on relationships among participants of architectural patterns.

There have been several attempts at introducing formal representations in the design patterns area such as pattern representation supported with ontologies [13] or formally specifying design patterns solution (see for instance [28]). The ontologies concept is used primarily to describe the structure of source code, which is done according to a particular design pattern. The work in the field of ontology-based pattern modeling mostly covers [32]: (i) creation of standardized templates for the description of ontology-based design patterns; (ii) creation of functional and generalized methods for users with different level of expertise in pattern reuse; and (iii) creation of techniques and tools for supporting a semi-automatic or automatic pattern selection. However, most of the ontology-based pattern modeling approaches work at detail-level design issues such as functional calls, parameter passing, and so on. Similarly, the approaches to formally specify design patterns have not gained much momentum in recent years mainly because of their complexity and their resulting limitations regarding their practical use. Moreover, these approaches have not been used for architectural patterns or whole pattern languages, like our relationships, but just for some isolated patterns. Our work focuses on relationships between the participants of several architectural patterns at the architecture design level offering different possibilities to combine architectural patterns, which is not yet fully addressed by existing ontology-based pattern modeling approaches and formal pattern specification approaches.

There have also been several attempts for specifying existing Architecture Description Languages (ADLs) [29] or proposing new ADLs such as ADLs proposed as extensions of UML [26], usually

DOI: 10.1002/spe

in the form of profiles. Most of these ADLs treat architectural patterns as first-class entities and provide tool support for modeling patterns. For instance, the ACME [29] provides built-in templates that can be used to model patterns. However, extensive design effort is required to merge, remove, and override the participants of related patterns for integration. Our approach aims at more flexibility by providing a wider range of lower-level relationships that once supported by an existing ADL, can be used for effectively combining architectural patterns.

Some researchers have performed empirical studies for the use of architectural patterns in designing software architecture [15] as a mechanism to capture design decisions [19] and as solutions to satisfy specific quality attributes [18]. However, to the best of our knowledge, no work has been done so far to evaluate the effectiveness of using pattern languages for combining architectural patterns within software architectures. Table I gives an overview of the related work, and how it compares to the approach presented in this paper.

3. MINING PATTERN PARTICIPANTS RELATIONSHIPS FOR MODELING PATTERNS

The relationships presented in this section are based on the study of software architectures from 32 industrial software systems [14], pattern integration examples documented in the literature [4][3][36], and patterns presented in workshops and conferences [21]. The patterns integrated within real software architectures are analyzed to discover the relationships between the participants of related architectural patterns. In the following sub-sections, we describe the approach for mining pattern participants relationships, a template to document the discovered relationships, and finally present all relationships discovered during this study.

3.1. The mining process

The underlying idea behind our approach is that various architectural patterns can be effectively integrated using a set of "pattern participants relationships". The relationships serve as a basis for identifying the participants of architectural patterns that share, overlap, or override other participants of related architectural patterns. Specifically, we followed three steps to mine pattern participants relationships:

- We started by identifying architectural patterns from architecture design diagrams by following the pattern mining process defined in [18]. Subsequently, we identified the participants of the discovered patterns that associate with participants of other patterns within the software architectures. We documented such associations.
- We used the software architecture design documents of several industrial software systems to read the description of architectural patterns integrated for designing such systems. The relationships between the participants of such patterns were identified and documented.
- We studied the pattern integration examples documented in the literature to look for pattern participants relationships. We came across several pattern integration examples in [4], [3], [36], and so on.

3.2. Template for pattern participants relationships documentation

Each pattern participant relationship discovered during this study is documented according to the following template:

- *Name*: Short intent of the relationship and its name.
- *Issue*: Brief description of a design issue for integrating patterns.
- *Definition*: Description of the pattern participant relationship in the context of a combination of architectural patterns.
- *Known uses in patterns integration*: Three known uses of the relationship in combinations of architectural patterns.
- *Example*: A pattern integration example to describe the discovered relationship.

DOI: 10.1002/spe

3.3. Pattern Participants Relationships

The solution specified by architectural patterns is comprised of components and connectors called pattern participants [4]. However, certain patterns can be integrated in a single component or connector of another pattern and hence can be considered as participants of such patterns as documented in [18]. For instance, the Asynchronous [3] pattern can be combined with the Pipe participant of the Pipes and Filters pattern for defining asynchronous connections between adjacent Filters. To avoid complexity, we call such participants as patterns in the examples documented in this section. Following, we use the mining process described earlier to document relationships between participants of different architectural patterns.

Redundant pattern participants: absorbParticipant

Definition: An absorbParticipant relationship defines how the participants of different patterns performing similar responsibilities are integrated in a single element. In an absorbParticipant relationship, certain participants of a pattern are absorbed by the participants of another pattern to avoid redundancy.

Issue: Considers the case where two patterns consist of two or more functionally equivalent, yet independent pattern participants. One or more such participants in a pattern, however, may become redundant and cannot be included in the software architecture as they are. The problem, now, is the way in which the pattern integration process should deal with these redundant participants in an effective way.

Known uses in pattern integration:

- The event-handling solution is present in both the Proactor [3] and Leader–Follower [3] patterns.
- Both the Reactor and Proactor [4] patterns introduce their own handles for demultiplexing and dispatching events to corresponding event handlers.
- The Dispatcher participant is present in both the Acceptor–Connector [36] and Interceptor [36] patterns.

Example: Both the Reactor and Acceptor–Connector patterns introduce their own event-handling participants for using different services [36]. The separate event-handling solutions in both patterns carry redundancy in applying these patterns in combination to a software architecture, for example, the handler participant is present in both the Reactor and Acceptor–Connector patterns. Figure1 shows the absorbParticipant relationship between the Reactor and Acceptor–Connector patterns.

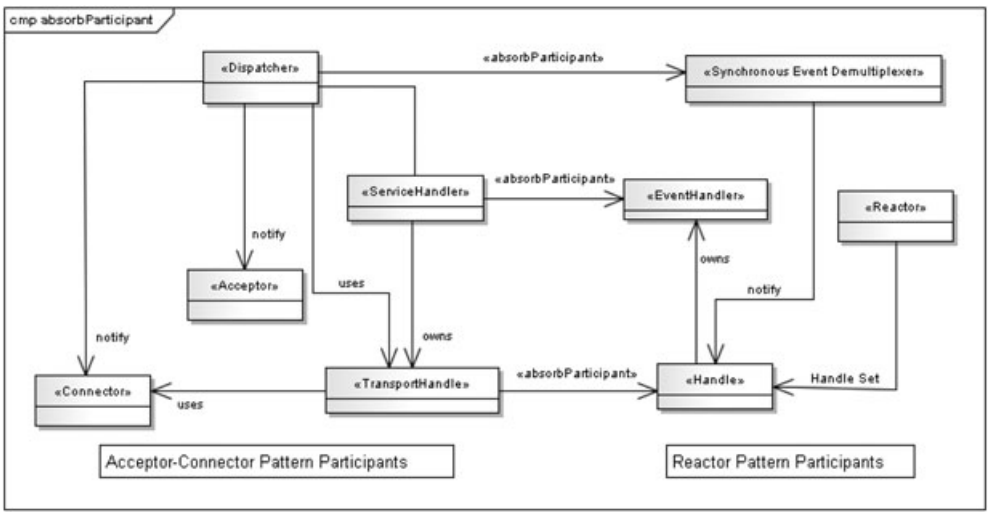

Figure 1. The absorbParticipant relationship between the Reactor and Acceptor–Connector patterns.

DOI: 10.1002/spe

In the Reactor's architectural structure, for each service an application offers, a separate event handler is introduced that processes different types of events from certain event sources. However, the Acceptor–Connector pattern can be considered as an option to implement the Reactor's event handlers. This ensures that the Reactor pattern specifies "right" types of event handlers associated with the Acceptor–Connector pattern. In order to integrate both patterns, the overlapping pattern participants either need to be merged or participants of one pattern be replaced by the other. However, removing a specific participant within a pattern may impact the solution specified by that pattern and may require new associations between the participants of both patterns, which is not a trivial task and requires extensive design effort. Figure 2 shows the resulting architecture after combining the Reactor and Acceptor–Connector patterns using the absorbParticipant relationship.

Overlapping pattern participants: mergeParticipant

Definition: The mergeParticipant relationship is used to combine one or more semantically different pattern participants into a single participant within the target pattern. Such an integration retains the structural and semantic properties of individual participants into the target element. The mergeParticipant relationship is different from the absorbParticipant relationship where participants performing similar responsibilities are absorbed (i.e. redundant participants are virtually removed in the resulting software architecture).

Issue: While integrating architectural patterns, the overlapping pattern participants problem occurs when participants present in different patterns are intended to represent the same concept and hence need to be merged in a single component. Nevertheless, the resulting component is deemed to represent the result of the merge, in the same way that functions of both participants are present in the resulting participant and not merely the increment added by a participant.

Known uses in pattern integration:

- For logic-intensive interactive applications, the Model participant of the MVC [4] pattern can merge the responsibilities of the Strategy [3] pattern.
- In the Document-View pattern variant[4], the View participant combines the responsibilities of both the View and Controller from MVC using the mergeParticipant relationship while the Document participant corresponds to the Model in MVC.
- The Master participant within the Master–Slave [3] pattern can merge the Strategy [3] pattern for configuring the varying strategies without affecting the slaves.

Example: In a distributed data processing arrangement, pipes are realized as a form of messaging infrastructure to pass data streams between remote filters. Such a design supports flexible redeployment of filters in a distributed pipes and filters architecture. In such a structure, the message pattern can be merged with the Pipe participant to setup messaging pipes between filters. Figure 3 shows the mergeParticipant relationship between the Pipe participant and the Message pattern whereas

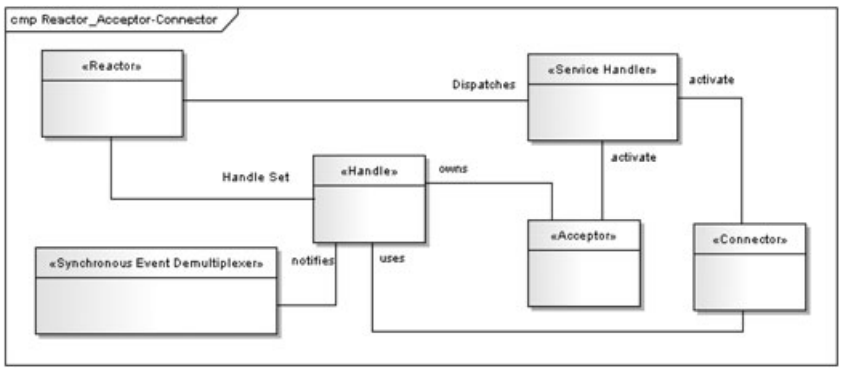

Figure 2. Integrating the Reactor and Acceptor–Connector patterns.

DOI: 10.1002/spe

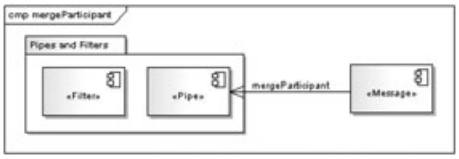

Figure 3. The mergeParticipant relationship.

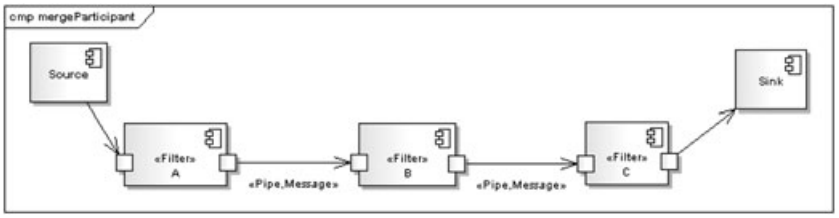

Figure 4. The mergeParticipant relationship example.

Figure 4 shows an example architecture after combining the Pipes and Filters pattern with the Message pattern.

Modeling patterns within the participant of a target pattern: importPattern

Definition: importPattern is a relationship where the participant(s) of a target pattern import all participants from a source pattern. This means that all participants of a pattern are modeled within the participant of another pattern. The importPattern relationship is similar to Package import in UML [38], Family import in ACME [16], and so on.

Issue: In certain cases of pattern integration, it is possible that one pattern acts as a solution participant of another pattern to solve a design problem at hand. However, one challenge to model such a solution is that the imported pattern must not overwrite the target pattern as both work as complementary solutions to a design problem and not as alternatives.

Known uses in pattern integration:

- The Dispatcher participant within the Client–Server [4] pattern imports the Activator pattern to activate–deactivate services running on different servers.
- In a distributed software architecture design, the Broker [4] hides and mediates all communication between the objects or components of a system. A Broker can import the Requester pattern for its internal implementation in order to forward requests from a client to associated components.
- The Server participant within the Client–Server pattern can itself be internally partitioned into several layers by importing the Layers pattern.

Example: Individual layers in the Layers pattern can import other patterns for their complete implementation. For instance, a single layer may be implemented as a data processing layer using the Pipes and Filters pattern as shown in Figure 5 while an example of the importPattern relationship is shown in Figure 6.

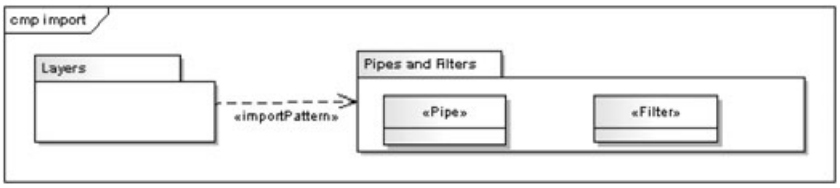

Figure 5. The importPattern relationship.

DOI: 10.1002/spe

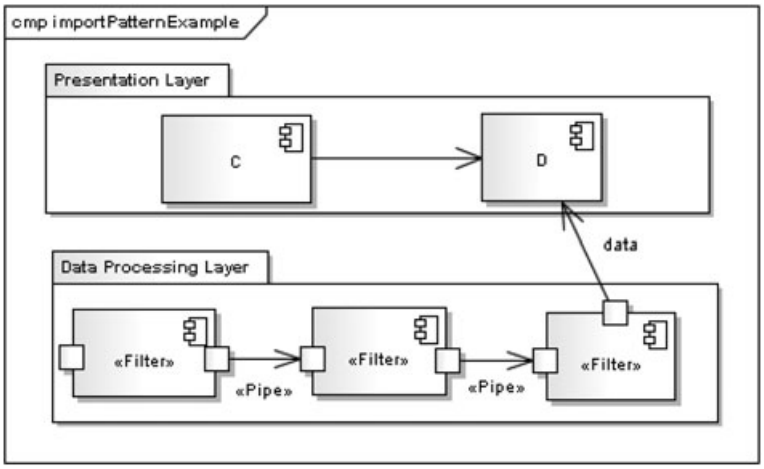

Figure 6. The importPattern relationship example.

Modeling participants within another pattern participant: importParticipant

Definition: An importParticipant is a relationship where participants of the target pattern import *specific* participants from the source pattern.

Issue: Similar to the problem addressed in the importPattern relationship, the import of specific participants into target pattern must not replace the target pattern's participants. For instance, integrating two or more objects or classes, defined as pattern participants, must not override each other in the resulting architecture.

Known uses in pattern integration:

- A specific Layer may import the Model participant of the MVC [4] pattern.
- The Broker [4] and Reactor [36] patterns can be integrated to design event-driven software architecture. In this particular example, the Request Handler participant of the Broker pattern imports the EventHandler participant of the Reactor pattern to handle multiple event sources simultaneously.
- When modeling the Half-Sync–Half-ASync [3] and Reactor [36] patterns in combination, the Querying Layer participant of the Half-Sync–Half-ASync pattern imports the EventHandler participant of the Reactor pattern to synchronize the invocation of services.

An example to describe the issue: Individual layers in the Layers pattern can import other patterns's participants. The Presentation layer can import only the View and Controller participants of the MVC pattern while the Model participant of the MVC pattern resides in the data logic layer as shown in Figures 7 and 8.

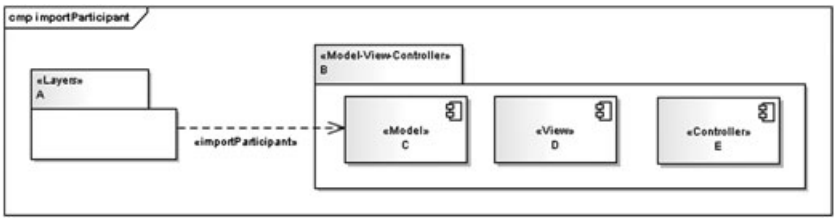

Figure 7. The importParticipant relationship.

DOI: 10.1002/spe

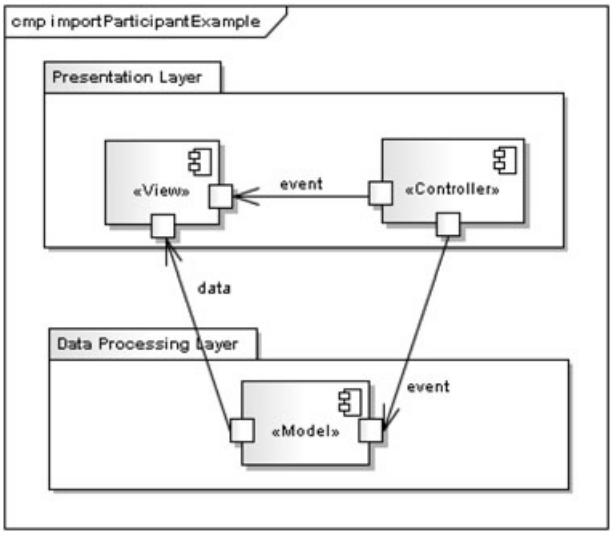

Figure 8. The importParticipant relationship example.

Participants make use of related pattern participants: employ

Definition: Employ is a relationship where participants of a pattern generally make use of another pattern for their complete implementation. Patterns using the "employ" relationship are often applied together within software architectures where one pattern "makes use of" another pattern to fulfill specific design needs. Frequent use of these patterns together helps associate the related participants of such patterns.

Issue: The loose dependency relationship described earlier is not explicit in current pattern relationship approaches making it difficult for software architects to combine related architectural patterns.

Known uses in pattern integration:

- The MVC [4] pattern employs the Observer [4] pattern to implement the change propagation mechanism.
- The Command [3] pattern implementation using the Composite [3] pattern is so common that it is often considered as single design solution to which the name Composite-Command is also used.
- The Broker [4] pattern often employs the Receiver and Invoker patterns so that clients can receive data and invoke services effectively.

Example: The Iterator pattern often employs the Batch Method pattern that supports access to aggregate elements without causing performance penalties and unnecessary network loads when the Iterator is remote to the aggregate [3]. The combined use of both patterns often leads to the term "Batch-Iterator" pattern in the literature [3]. However, both patterns can be applied independently to a software architecture to solve specific design problems. Figure 9 shows the employ relationship among the participants of the Iterator and Batch Method patterns. The resulting architecture after combining the Iterator and Batch Method patterns is shown in Figure 10.

Strong coupling between pattern participants: depends

Definition: The depends relationship shows the need of pattern participant(s) to use another pattern for their complete implementation. In contrast to the employ relationship, the depends relationship is a strong dependency of a pattern's participants on another pattern where, in practice, the participants of the source pattern always use the participants of the target pattern.

Issue: The strong coupling between the participants of relating patterns require that the participants from the source and target patterns must be present in the resulting software architecture. The

DOI: 10.1002/spe

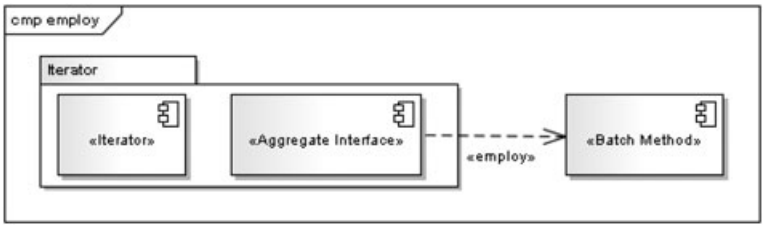

Figure 9. The employ relationship.

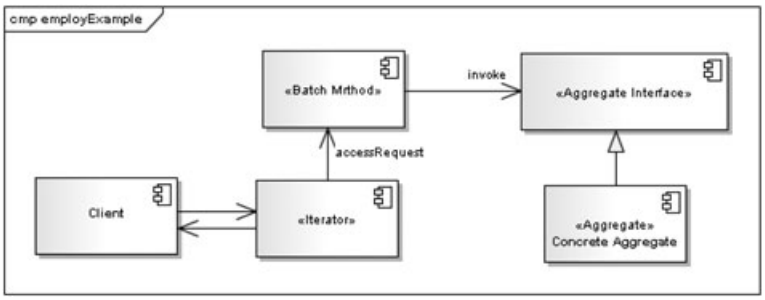

Figure 10. Integrating the Iterator and Batch Method patterns.

strong dependency relationship between the participants of related patterns is not explicit in current pattern languages.

Known uses in pattern integration:

- The Control participant within the Presentation-Abstraction-Control (PAC) [4] pattern depends on the use of the Mediator [3] pattern for coordinating with other PAC agents.
- The Microkernel [4] pattern is often modeled as three-layered architecture, that is, the Microkernel depends on the Layers pattern.
- The Reflection [4] pattern is modeled as a two-layered architecture using the Layers [4] pattern: a meta level contains the metaobjects, a base level the application logic.

Example: The Broker pattern separates and encapsulates the details of communication infrastructure in distributed systems. In such a structure, clients invoke remote services using the Broker as if they were local and, in return, receive response from servers that offer these services. In such a system context, the Broker pattern is always modeled in combination with the Client–Server pattern, as analyzed in this study, which is documented using the depends relationship between the Broker and Client–Server patterns. The depends relationship is shown in a particular example of Client–Server and Broker patterns integration in Figures 11 and 12.

Mediator pattern participants: interact

Definition: An interact is a relationship where certain participants of the source pattern interact with the participants of the target pattern to solve a design problem. In an interact relationship, the target pattern often acts as a mediator/redirector by mediating the requests between the source pattern and surrounding architectural elements.

Issue: The mediation/redirection role of the participant of a pattern to integrate related patterns requires that the communication to target components must pass through only the mediating participant. Such a relationship is challenging to identify, at pattern participants level, using the current pattern relationships approaches.

Known uses in pattern integration:

- The Client–Server [4] pattern interacts with the Proxy [4] pattern for sending/receiving messages. The proxy acts as a communication redirector between Clients and Server.

DOI: 10.1002/spe

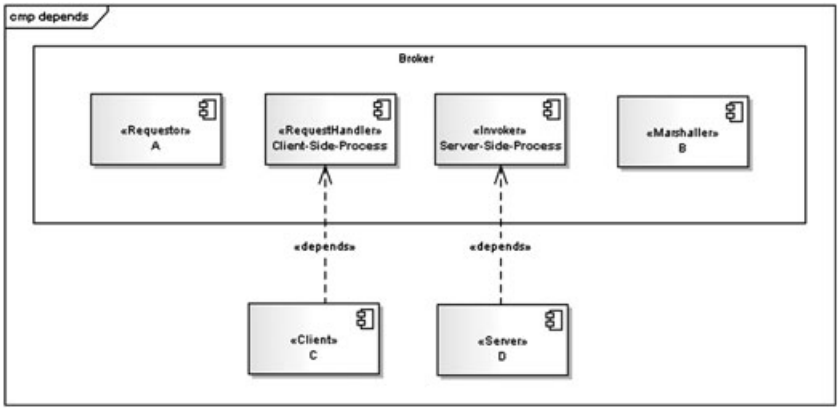

Figure 11. The depends relationship.

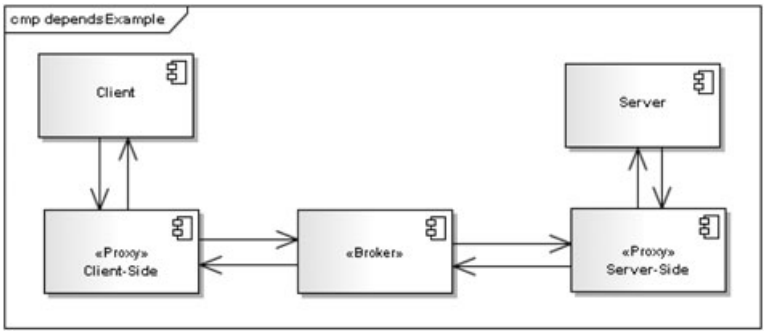

Figure 12. Combining the Broker and Proxy patterns.

- The Layers [4] pattern can use the Adapter [3] pattern that connects provided interface of the components in one layer into the interface that the clients expect in another layer, and vice versa.
- A Virtual Machine [2] interacts with the Layers [4] pattern by redirecting invocations from a bytecode layer into an implementation layer for the commands of the bytecode.

Example: The Broker [3] pattern interacts with the Proxy [3] pattern to send/receive information to/from Client and Servers. The Proxy pattern acts as a service redirection to forward requests to appropriate client–server. Figure 13 shows the interact relationship between the Proxy and Broker participants and Figure 14 shows the resulting architecture.

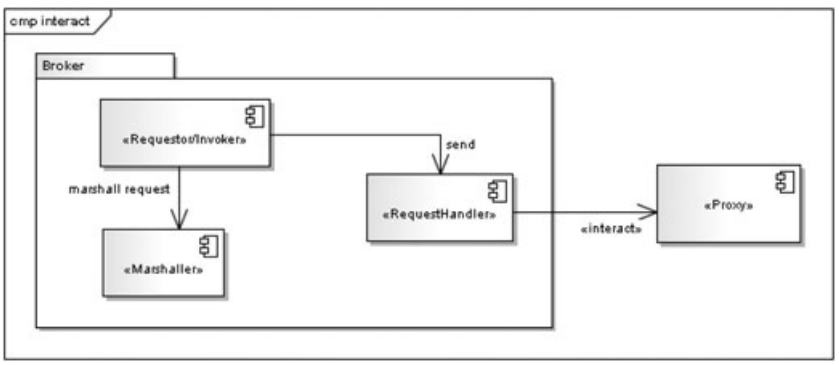

Figure 13. The interact relationship.

DOI: 10.1002/spe

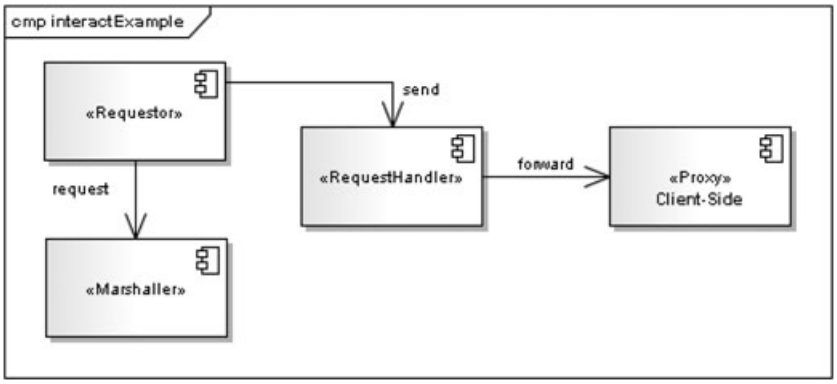

Figure 14. The use of interact relationship for combining patterns.

The intention is to use the pool of all available pattern participants relationships to integrate several architectural patterns. However, the relationships between patterns, as listed in Table II, are not fixed: rather the solutions entailed by two selected patterns can be combined in infinite different ways and so is the selection of relationships for integrating patterns. Thus, the decision to apply a specific relationship for integrating patterns lies with the architect who picks relationships that best meet the design needs at hand, that is, the *absorbParticipant* relationship is used only if it is required to avoid redundant participants in the resulting architecture. Using our relationships allows an architect to integrate several architectural patterns with certain level of design support w.r.t design requirements at hand.

Table II lists the mined relationships in a tabular form. We note that the set of relationships were elicited from real architectures, so they are actually practiced by software architects. However, they are not explicitly documented and architects cannot reuse them but need to discover them on a case-by-case basis. By documenting them, we strive for reusability of the relationships, especially among inexperienced architects and designers. Moreover, we have previously proposed an approach, called Pattern-Driven Architectural Partitioning (PDAP)[18] that documents how a pattern may influence the use of other patterns; a pattern may specialize the use of another, or how two patterns may be alternatives, and so on. An architecture design process, such as PDAP, can be successfully followed alongside the use of pattern participants relationships as the proposed relationships are aimed at complementing the existing software architecture design methods without affecting the core design activities of these methods, for example, analysis, design, evaluation, and so on. For instance, in the PDAP method, the step to partition the system by applying a combination of the candidate patterns can benefit by the use of pattern participants relationships. In the following sections, we present results from a controlled experiment where subjects had the freedom to follow any architecture design method for modeling patterns. In addition, a group of participants were provided the list of relationships to test the effectiveness of using the relationships for integrating architectural patterns.

4. EXPERIMENT DESIGN

We claim that the pattern participants relationships, discussed in the previous section, help architects to effectively integrate architectural patterns within software architectures. To test this claim, we performed a controlled experiment. In this experiment, two groups of graduate students were provided with the same information for designing software architectures based on a set of requirements. In addition, one group was provided with information about pattern participants relationships. The experiment was designed according to the guidelines for conducting empirical research in software engineering [31], and the results were documented according to the reporting guidelines for controlled experiments in software engineering [30]. Moreover, the suggestions from the working group for conducting controlled experiments [9] were taken into consideration while designing the experiment.

DOI: 10.1002/spe

In the following sub-sections, the design of the experiment, our hypotheses, involved subjects, variables, and the analysis of the data collected from the experiment are presented. With the final data gathered from the outcome of the experiment, we evaluate how the use of pattern participants relationships can help architects to effectively integrate architectural patterns within software architectures.

4.1. Research question and hypotheses

To analyze the use of pattern participants relationships for integrating architectural patterns, we present the research question and construct null hypotheses (H0i) and alternate hypotheses (H1i) as explained next.

Research Question: *Does the use of pattern participants relationships help to effectively integrate architectural patterns within software architecture?*

The effective integration of architectural patterns covers several aspects of software architecture design [10]. We have selected four such aspects that are evaluated according to the following hypotheses:

4.1.1. Null hypotheses.

1. *H00:* The use of pattern participants relationships does not help software architects to more accurately[§] combine architectural patterns within software architectures as compared with integrating architectural patterns without using such relationships.
2. *H01:* The integration of architectural patterns using pattern participants relationships does not result in a more comprehensible software architecture as compared with integrating patterns without the use of such relationships.
3. *H02:* The use of pattern participants relationships does not help software architects to better document architectural design decisions as compared with documenting design decisions without using such relationships.
4. *H03:* The use of pattern participants relationships does not help software architects to more effectively partition a software architecture into components and sub-components, and assign responsibilities as compared with partitioning a software architecture without the use of such relationships.

4.1.2. Alternate hypotheses.

1. *H10:* The use of pattern participants relationships helps software architects to more accurately combine architectural patterns within software architecture as compared with integrating architectural patterns using such relationships.
2. *H11:* The integration of architectural patterns using pattern participants relationships results in a more comprehensible software architecture as compared with integrating patterns without the use of such relationships.
3. *H12:* The use of pattern participants relationships helps software architects to better document architectural design decisions as compared with documenting design decisions without using such relationships.
4. *H13:* The use of pattern participants relationships helps software architects to, more effectively, partition software architecture into components and sub-components, and assign responsibilities as compared with partitioning software architecture without the use of such relationships.

4.2. Variables

We used independent variables as presumed "causes" and dependent variables as presumed "effects" [25]. Dependent variables are not manipulated in this study and are presented as is. We use independent variables to measure their influence on the final results. For instance, a participant having

[§]The term accurately refers to identifying the presence of redundant pattern participants and inappropriate linking of pattern participants within a software architecture.

DOI: 10.1002/spe

Table II. Pattern participants relationships discovered in software architectures.

Patterns	Acceptor-Connector	Activator	Adapter	Blackboard	Broker	Client-Server	Composite	Event-Handler	HalfSync/AHalfSync	Invoker	Mediator	Layers	Microkernel	MVC	Observer	Pipes and Filters	Plug-in	Proxy	Publish-Subscribe	Receiver	Requester	RPC	Scheduler	Shared Repository	Strategy	Leader-Follower	Mediator	Proactor	Virtual Machine
Active Repository [3]															importPattern														
Broker [3]						depends				employ								interact		employ	importPattern								
Client-Server [4]	absorbParticipant	importPattern		mergeParticipant	interact				absorbParticipant						importPattern			interact		importPattern									
Command [3]							employ																						
Layers [4]			interact		importPattern,interact	importParticipant	importParticipant					importParticipant	importParticipant	importParticipant	importPattern	importPattern	importPattern	importParticipant											interact
Master-Slave [3]																								mergeParticipant					
Microkernel [3]												depends																	
MVC [3]															employ	interact							interact						
PAC [3]												mergeParticipant				interact							interact	importPattern	mergeParticipant		depends		
Publish-Subscribe [3]					interact											importPattern													
Reactor [3]	absorbParticipant				importParticipant			importParticipant	importParticipant																			absorbParticipant	
Reflection [3]												depends																	
Proactor [3]																										absorbParticipant			
Interceptor [3]	absorbParticipant																												

a very good understanding of architectural patterns may significantly influence the results gathered from a group.

Independent variables: We considered four independent variables prior to conducting the experiment as listed in Table III. All four independent variables, namely architecture design experience,

DOI: 10.1002/spe

Table III. List of independent variables.

Name of the variable	Class/entity	Type of attribute (internal/external)	Scale	Measurement unit	Range	Counting rule
Architecture design	Software architecture	external	ordinal	year	Above 5, 3 to 5, 1 to 3, no experience	Pre-experiment feedback
Experience of integrating patterns	Architectural patterns	external	ordinal	expertise	Very good, good, average, little or no experience	Pre-experiment feedback
Belief in using patterns	Architectural patterns	external	ordinal	agreement	Very helpful, helpful, just OK, not very helpful	Post-experiment Post-experiment feedback from
Understanding of candidate patterns	Architectural patterns	external	ordinal	patterns count	More than 10, 8 to 10, 4 to 7, 0 to 3	Post-experiment feedback

Table IV. List of dependent variables.

Name of the variable	Class/entity	Type of attribute (internal/ external)	Scale	Measurement unit	Range	Counting rule
Pattern integration	Architectural patterns	internal	interval	numeric	1 to 10	score
Design comprehensibility	Software architecture	internal	interval	numeric	1 to 10	score
Design decisions documentation	Architectural patterns	internal	interval	numeric	1 to 10	score
Architecture decomposition	Software architecture	internal	interval	numeric	1 to 10	score

pattern modeling experience, belief in using patterns, and understanding of candidate architectural patterns are measured according to ordinal scale as per the measurement scales documented in [24].

Dependent variables: The experiment used four dependent variables, as shown in Table IV, for analyzing the integration of architectural patterns within software architectures. The four dependent variables correspond to the four hypothesis. We evaluate the consequences of combining architectural patterns, once with the use of pattern participants relationships, and once without, respectively, for the treatment and the control group. Each of the four dependent variables is measured according to an interval scale with values ranging from 1 to 10 (1, poorest; 10, good). The selection of the interval measurement scale was based on the measurement scales documented in [24].

4.3. Experiment design

Each of the subjects participating in the experiment was specifically asked to integrate architectural patterns for designing software architecture. Two balanced groups with comparable skill levels were formed to serve the purpose. The assessment of subject skills was performed by providing a pre-experiment questionnaire to students where they were asked to provide information about their architecture design experience, and educational background as further discussed in Section 4.4. The outcome from this experiment was analyzed using statistical methods as discussed in Section 5.

4.4. Subjects

The subjects in the experiment were 36 graduate students from the Computer Science department of University of Groningen, Netherlands. All subjects were enrolled in a software patterns course, and they had previously passed a Software Architecture course. This software architecture course also included designing a non-trivial system. In the software patterns course, students were instructed about the use of patterns as solutions to design problems, consideration of alternate patterns, integrating patterns, and patterns influence on quality attributes. Before assigning subjects to the control

DOI: 10.1002/spe

and treatment groups, we determined the background knowledge and software architecture design experience of the subjects through a pre-experiment questionnaire.

We believe that software architecture design requires a certain level of expertise. For instance, subjects must have some knowledge about architectural views (e.g. structural view [38], behavioral view [38]), architectural concerns (e.g. [35]), architectural elements (e.g. components, connectors ports [26]), and so on. The skill level of subjects was assessed based on the subjects architecture design experience and educational background. This was achieved by seeking pre-experiment feedback from subjects. Among the 36 subjects participating in the experiment, there were two PhD students and 34 master students.

4.5. Objects

The subjects were provided with a Software Requirement Specification of an industrial warehouse management system, a list of candidate architectural patterns (such as Client–Server [4], Broker [3], Layers [4], MVC [4]), a template-to-document design decisions, and a number of quality requirements. Additionally, handouts containing the description of several architectural patterns for designing distributed systems were provided to the subjects. All architectural patterns were alphabetically indexed on a separate sheet with page numbers for easy referencing for the subjects to search and read the description of patterns.

4.6. Instrumentation

Before the start of the experiment, both groups were given sufficient time to read the Software Requirement Specification document and ask questions to ensure their understanding of the requirements. Additionally, the treatment group was provided the pattern participants relationships document. To guide subjects for documenting the design decisions, a simple template was provided where the subjects were asked to document the decision number, a short description of the design decision, and the rationale for supporting the design decision.

4.7. Data collection procedures

The experiment was performed in two sessions held at the same day. After an introduction of the system and a question/answer session, the subjects had 2 h and 15 min to design the architecture. Furthermore, the subjects were requested to fill out a post-questionnaire to document their feedback about the experiment, knowledge of the listed candidate architectural patterns, and issues in identifying participants of related patterns. Additionally, the treatment group was asked to document how helpful they found pattern participants relationships for integrating patterns. The architecture design document, design decisions document, and post-questionnaires were collected from the subjects after the experiment.

4.8. Analysis procedure

To analyze the data, we requested three expert reviewers to give their judgement upon the selected aspects of the software architectures. The external reviewers had several years of architecture design experience. One of them is an industrial practitioner for the past several years, whereas the other two have substantial industrial and academic experience. The information about the subjects allocation to the treatment and control groups, and treatment information was not revealed to the external reviewers. The expert reviewers were asked to provide both the grades and comments for each architectural design. For this purpose, a set of architecture evaluation criteria was provided to the reviewers to document their feedback. However, reviewers' identity and final results were not revealed to each other until the final data were collected.

To make analysis efficient, we considered it highly important that the reviewers reached consensus in their understanding of the selected architectural aspects that we used in the evaluation criteria such as pattern integration, design comprehensibility, design decisions, and decomposition. This was achieved by providing a brief description of each architectural aspect used in the evaluation criteria and asking reviewers to send us their feedback in case they disagree with the documented

DOI: 10.1002/spe

description of architectural aspect. This procedure was performed several days prior to conducting the experiment. Only minor modifications to the architectural aspects descriptions were suggested by reviewers, which were revised accordingly.

To perform the statistical analysis of the data gathered from the expert reviewers feedback, we performed Levene's test [17] and *t*-test [12] to determine whether the differences in mean values calculated between the groups are significant. The Levene's test is used to check if the two groups have equal variances for the selected dependent variable. The *t*-test is used to measure whether the found differences are statistically significant [12]. The *t*-test calculates the chance that similar results will be produced when the experiment is repeated (i.e., the chance that mean values differ for control and treatment groups).

4.9. Validity evaluation

We improved the reliability and validity of the experiment and data collection in two ways. First, by performing a pilot run of the experiment with one subject a few days prior to conducting the experiment and taking feedback from the subject about any issues in understanding and executing the plan. This subject did not participate in the real experiment execution, and he had no contact with any of the subjects participating in the experiment. Secondly, we ensured that one of the authors was available to the participants during the entire experiment, in case they faced any issues like understanding the design decisions template, availability of paper sheets, and so on. Furthermore, the design of the experiment was revised several times by sharing the study design with researchers having good know-how of empirical research, and changes were made where necessary.

5. EXECUTION OF THE EXPERIMENT

This section discusses the instantiation of samples, randomization, instrumentation, execution of the experiment, data collection, and validation of results.

5.1. Sample

- *Blind experiment:* The subjects in the experiment were not told about the hypothesis, that is, we performed a blind experiment.
- *Blind Task:* To ensure that all participants had the same knowledge of the system to be designed, the system description for which the architecture has to be designed and information about the use of pattern participants relationships, and so on, were kept secret until the day of exercise.
- *Technology restriction:* To make sure that the use of technology did not influence the results, students were not allowed to use software architecture design tools or refer to the internet.

5.2. Preparation and data collection

The preparation and data collection went smoothly according to the experiment design described in Sections 4 and 4.7.

5.3. Validity procedure

No major problems were encountered during the execution of the experiment. One participant was concerned with the extent to which he should document the design decisions. The subject was briefly consulted and guided appropriately.

5.4. Statistical analysis of the data

With the availability of a limited number of subjects for software architecture design experiments, we believe it is important to obtain maximal information from the data gathered to draw any conclusion. The *t*-test and Levene's test are used to analyze the numerical data, and subjective analysis is performed to interpret the results.

DOI: 10.1002/spe

The *t*-test aims at hypothesis testing to answer questions about the mean of the data collected from two random samples of independent observations. The Levene's test is performed for equality of variances among the control and treatment groups. For the Levene's test, if the significance value is less than or equal to 0.05, then equal variances is not assumed, or else, the variance for both groups is considered to be equal. Separate graphs are used to present data generated from the resulting software architectures. The statistics (using demographics and tables) show the difference in the results between the control and treatment groups.

6. RESULTS OF THE EXPERIMENT

In this section, for each dependent variable, we document the number of subjects (N), the mean(M), the standard deviation (SD), and the standard error of mean (SEM) of the samples. The *t*-test is performed to test if the null hypothesis can be rejected.

6.1. Pattern integration

The integration of architectural patterns within a software architecture often requires new communication links and results in removing certain pattern participants or some participants being merged into a single element. The appropriate integration depends on how correctly and explicitly these tasks are performed by software architects. Table V shows the mean, standard deviation, and standard mean error for the control and treatment groups. The SD value for the control group is slightly higher than the treatment group indicating less variation in the individual scores of the treatment group from the mean value. There is a significant difference between the resulting mean values for the two groups (control group = 5 and treatment group = 6.6), which shows the better performance of the treatment group for integrating architectural patterns as compared with the control group.

Table V. Statistical results for different variables.

	Group	N	M	SD	SEM
Patterns integration	Control	16	5	1.26	0.32
	Treatment	18	6.6	1.14	0.27
Design comprehensibility	Control	16	4.9	1.39	0.35
	Treatment	18	6.3	1.16	0.27
Design decisions	Control	16	5.2	1.56	0.39
	Treatment	18	6.5	1.48	0.35
Architecture decomposition	Control	16	4.9	1.53	0.38
	Treatment	18	6.2	1.33	0.31

The Sig. value from Levene's test is greater than 0.05, as shown in Table VI, which shows almost equal variances among the control and treatment groups. We perform the *t*-test to analyze the data gathered for the "pattern integration" aspect. Table VI shows the statistics of the data. The *t*-test with 32 df generates *p*-value equal to 0.003, which is considered to be statistically significant. The *p*-value 0.003 shows more than 99 per cent confidence that the treatment group performed better as compared with the control group.

6.2. Design comprehensibility

The completeness and clarity of the resulting software architecture adds to the comprehensibility of the software architecture. Table V shows the mean, standard deviation, and standard error mean values for the "design comprehensibility" variable.

There is a significant difference between the mean values calculated for both groups (control group = 4.9 vs treatment group = 6.3), which provides an indication that the treatment group performed better than the control group.

As a prerequisite to run the *t*-test, Levene's test is performed to check the equality of variances among both groups. The Sig. value from Levene's test is greater than 0.05, as shown in Table VI,

DOI: 10.1002/spe

Table VI. *t*-test results for different variables.

		Levene's test for equality of variances	*t*-test for equality of means			
		Sig.	*t*	df	*p*	Mean diff.
Pattern integration	Equal variance assumed	0.58	3.25	32	0.003	1.6
Design comprehensibility	Equal variance assumed	0.56	2.99	32	0.01	1.4
Design decisions	Equal variance assumed	0.91	2.53	32	0.02	1.32
Architecture decomposition	Equal variance assumed	.61	2.31	32	0.03	1.3

which shows equal variances among the control and treatment groups [17]. We perform the *t*-test to analyze the data gathered after the experiment. Table VI shows the statistics of the data. The *p*-value equals 0.01, which is considered to be statistically significant. The *p*-value 0.01 shows the probability that 1 in 100 randomization of subjects can lead to different results. The *t*-test value of 2.99 indicates that the treatment group performed better as compared with the control group as documented in [12].

6.3. Design decisions

Architectural patterns are considered an important mean for documenting design decisions [17]. We evaluate the effectiveness of using pattern participants relationships for documenting design decisions. Table V shows the mean, standard deviations, and standard error mean values for the treatment and control groups. The treatment group has scored a higher mean value as compared with the control group (treatment group = 5.2, control group = 6.5). The Levene's test and *t*-test are performed to verify the significance of difference in mean values.

The Sig. value from Levene's test in Table VI shows equal variances among the control and treatment groups. We perform the *t*-test to analyze the data gathered for the "design decisions" variable. Table VI shows the statistics of the data. The *p*-value equals 0.02, which is considered to be statistically significant [12]. The *p*-value 0.02 shows the probability that 2 in 100 randomization of subjects can lead to different results [12], which shows that the results are statistically significant and would allow us to reject the null hypothesis. We can be confident that the treatment group performed better as compared with the control group.

6.4. Architecture decomposition

An important aspect for modeling architectural patterns is decomposition of software architecture into manageable components and sub-components. Effective integration of architectural patterns can result in well-partitioned software architecture [4]. Table V shows the mean, standard deviation, and standard mean error of data collected for the control and treatment groups. There is a significant difference in the resulting mean value for the two groups (control group = 4.9 and treatment group = 6.2), which shows the better performance of the treatment group for decomposing the software architecture as compared with the control group.

The Sig. value from the Levene's test is 0.61, which shows high level of homogeneity in variance between both groups [17]. We perform *t*-test to analyze the data gathered for the "design decomposition" variable. Table VI shows the statistics of the data. The *p*-value equals 0.03, which is considered to be statistically significant [12]. The *p*-value 0.03 indicates the probability that 3 in 100 randomization of subjects can lead to different results [12], which is statistically negligible, and we can be confident that the treatment group performed better as compared with the control group. The *t*-test

DOI: 10.1002/spe

value of 2.31 indicates that the treatment group mean is greater than the control group mean as documented in [12].

6.5. Data set reduction

As the subjects were specifically asked to merge architectural patterns in their assignment, the exclusion criteria were based on the use of at least four patterns or any data point that is more than two standard deviations away from the mean [12]. Figure 16 in Appendix A shows the data plot graphs for overall mean scores obtained by individual subjects with respect to the four architectural aspects considered in this study. Among 36 subjects considered in this study, the data gathered from two students were excluded from the study based on the preceding defined criteria and subjects inclusion/exclusion criteria documented in [31]. Appendix A further discusses the exclusion of outliers in this study.

6.6. Hypothesis testing

Figure 15 shows the average score for both groups w.r.t the four aspects considered in this study. All four aspects considered in this study have p-values in the range 0.003 to 0.03, which are considered statistically significant to reject the null hypotheses.

Whereas there are more than one dependent variable used to test the hypotheses, it is obvious from the results that the treatment group managed to more effectively combine architectural patterns within software architectures, compared with the control group.

7. INTERPRETATION

7.1. Evaluation of qualitative data and implications

We performed analysis of the qualitative data received from the expert reviewers and participants, in addition to the statistical analysis performed in the previous section. The qualitative data were gathered in two forms: feedback from the participants in the post-questionnaires, and expert reviewers feedback regarding individual software architecture documents. The analysis of the qualitative data can provide additional information to assist with the interpretation of quantitative results as presented here:

- There were two major design problems identified by the expert reviewers in the control group that were a direct consequence of "inappropriate" integration of architectural patterns, as we concluded from the textual feedback given by reviewers. In one design document, the architect modeled patterns as "black boxes" providing no connections among pattern participants. In

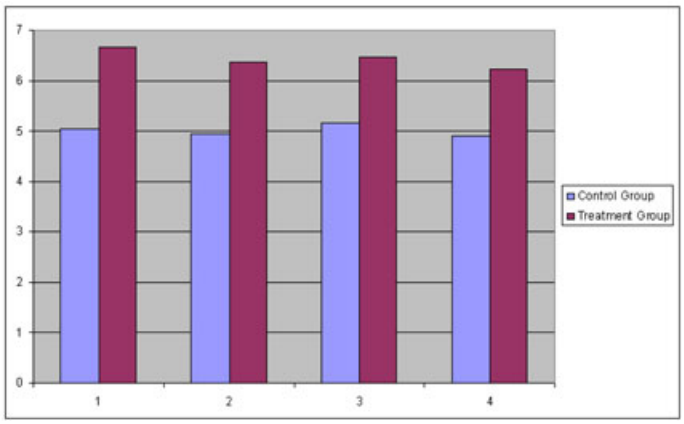

Figure 15. Average scores obtained by the control and treatment groups.

DOI: 10.1002/spe

another case, the architecture was considered too generic to fit the system context. In comparison, the treatment group managed to better address the design problems by coming up with more comprehensible software architectures as compared with the control group.
- By mapping the post-experiment questions about understanding of the listed candidate architectural patterns, it was noticed that the participants in the treatment group with better understanding of the listed candidate patterns managed to produce better quality architecture as compared with the participants with similar level of expertise in the control group. This leads us to a possible conclusion that pattern understanding alone is not enough to produce high quality software architecture, but the effective integration of patterns improves the quality of software architecture.

7.2. Limitations of the study

Threats to internal validity:

Internal validity is the degree to which the values of dependent variables can be attributed to the experiment variables, for example, balancing groups, use of statistical method, and so on.

- In order to avoid bias in allocating participants to the treatment group, we assigned participants to each group randomly based on their expertise level. For instance, if there were six subjects with the same level of expertise(i.e., experience, education background, etc.), three were randomly picked for the treatment group, and the other three for the control group.
- Another threat to validity is the selection of appropriate statistical method for data evaluation. We addressed the issue by sending data to an expert in the field of statistics and by studying alternate statistical methods to pick one that best fits the nature of data gathered after the experiment, that is, interval scale, scores ranging from 1 to 10, and so on.
- *External reviewers bias:* There is a possibility that external reviewers may be biased in grading one or more architectural aspects considered in this study. This is because the expert reviewers may interpret a selected architectural aspect differently, for example, the design comprehensibility aspect may be interpreted differently by different reviewers. In an effort to reduce the impact of reviewers bias on final results, the selected aspects were discussed with the reviewers to seek their feedback. A brief description of the aspects was then provided to three reviewers.

Threats to external validity:

- There was a risk that the participants may have different educational background, which was not the case in our experiment. All participants had educational background in software engineering and computer science. This means that our results are more generalizable to "people" with technical background than those with a non-technical background.
- *Generalization:* The subjects who participated in the experiment (graduate students) are unlikely to be representative of experienced industrial software architects. However, Sjoberg *et al*. in [33], have also suggested that graduate students of computer science be considered as semiprofessionals and, hence are not so far from practitioners. The experiment results encourage us to further exploit the use of pattern relationships for integrating architectural patterns in industrial experiments.
- *Time constraint:* We believe that software architecture design is a lengthy and complex activity and not all of the architectural aspects (i.e., architectural views, detail component partitioning, etc.) can be addressed in a limited time frame. However, subjects were asked to perform a limited task, to integrate architectural patterns within software architecture. In the design of the experiment, we considered 2 h and 15 min sufficient for subjects to come up with reasonable architecture. The decision to allocate 2 h and 15-min time slot for each group was verified by a pre-experiment pilot run of the study.
- There is a risk that the three expert reviewers may significantly differ in rewarding grades to a specific software architecture. To avert this risk, we performed inter-rater agreement test to identify major differences in grades. The inter-rater correlation test was used to identify the

DOI: 10.1002/spe

degree of homogeneity in grades. The Tables in Appendix B provide the results of performing the inter-rater correlation test, which shows acceptable level of difference in homogeneity of grades.

8. CONCLUSIONS AND FUTURE WORK

This paper presented an approach to support practitioners in the task of integrating architecture patterns by documenting a list of relationships at the level of pattern participants. The approach was validated through a controlled experiment. Four aspects were taken into consideration for integrating architectural patterns: pattern integration, design comprehensibility, design decisions, and architecture design decomposition. The subjects, which were provided pattern participants relationships managed to more effectively integrate architectural patterns within software architectures as compared with participants, which were not provided such information. The results from our experiment show that a more rigorous documentation of relationships among architectural patterns can help inexperienced architects to come up with higher quality software architectures. We can further make the following comments: (i) understanding of architectural patterns can not guarantee by itself a good application of patterns in an architecture unless architectural patterns are effectively integrated; (ii) the four aspects considered in this study for analyzing the quality of software architectures are only a few of many architectural aspects, all of which require more empirical research.

As future work, we are in the process of developing an Eclipse-based pattern modeling tool. The tool supports modeling architectural patterns and pattern variants within software architectures. We plan to define the pattern participants relationships as part of a UML metamodel and extend the tool towards selecting and integrating related architectural patterns. We would also like to cover several other different kinds of design patterns like security patterns, reliability patterns, usability patterns, and so on. We believe that expanding the idea of pattern participants relationships to large sets of patterns will lead to a library of architectural patterns and how they interact, providing useful help to software architects for integrating a variety of architectural patterns.

APPENDIX A

Figure 16 shows the final grades assigned to individual architectures on a scale of 1 to 10. The final grades are calculated as average score for four variables used in this work. Grade 2.1 from the control group and grade 2.3 from the treatment group are considered outliers and, hence are not considered when performing statistical analysis in this work.

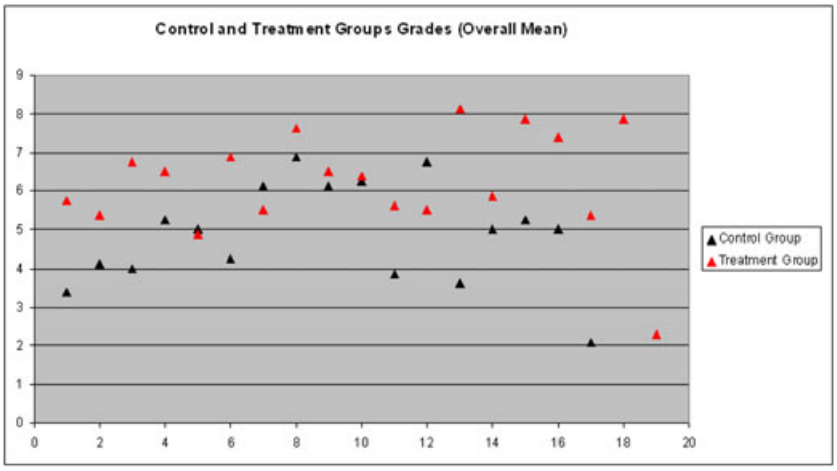

Figure 16. Identification of outliers in the control and treatment groups (Average score).

DOI: 10.1002/spe

APPENDIX B

The agreement among the external reviewers in assigning the grades to individual architectural aspects is calculated according to the kappa statistics as shown in Figure 17 (cases validity), Figure 18 (different combinations of agreement/disagreement between reviewers), and Figure 19 (kappa value). The combination of agreement/dissagreement between the external reviewers, as shown in Figure 18, is performed according to SPSS tool guidelines [12]. For instance, the value of 18 in Figure 18 shows the total number of instances where reviewers had consensus in assigning a low grade to an architecture.

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
Case1 * Case2	124	100.0%	0	.0%	124	100.0%

Figure 17. Case processing summary.

		Case2			
		1	2	3	Total
Case1	1	18	5	0	23
	2	3	51	5	59
	3	3	7	32	42
	Total	24	63	37	124

Figure 18. Cross tabulation data for matching and non-matching cases.

		Value	Asymp. Std. Error[a]	Approx. T[b]	Approx. Sig.
Measure of Agreement	Kappa	.701	.056	10.685	.000
	N of Valid Cases	124			

Figure 19. Symmetric measures – kappa statistics.

The results of the inter-rater analysis are Kappa $= 0.7$ with approx. sig. value less than 0.001. This measure of agreement is considered statistically significant. The Kappa values of at least 0.6 and preferably higher than 0.7 are considered significant before claiming a good level of agreement [12].

APPENDIX C

In this section, we present an example to design part of a warehouse management system [3]. A key requirement for the development of such a system is the communication middleware that offers business process management. The goal of the communication middleware is to simplify application development by providing uniform view of network services and separate core application functionality from communication complexities such as connection management, data transfer, even and request demultiplexing, and concurrency control, and so on. Some of the key nonfunctional requirements deemed in the resulting software architecture are scalability, portability, flexibility, and distribution. It must be noted that a software architecture design activity involves several steps like requirement analysis, prioritization of key drivers, selection of appropriate patterns, verification, validation and so on. For the sake of simplicity, we do not document all architecture design activity steps and focus only on the pattern integration process alongside the major design decisions. Also, the warehouse management system is a large scale system, and, in this section, we present only a part of the architecture to demonstrate the working of pattern participants relationships.

DOI: 10.1002/spe

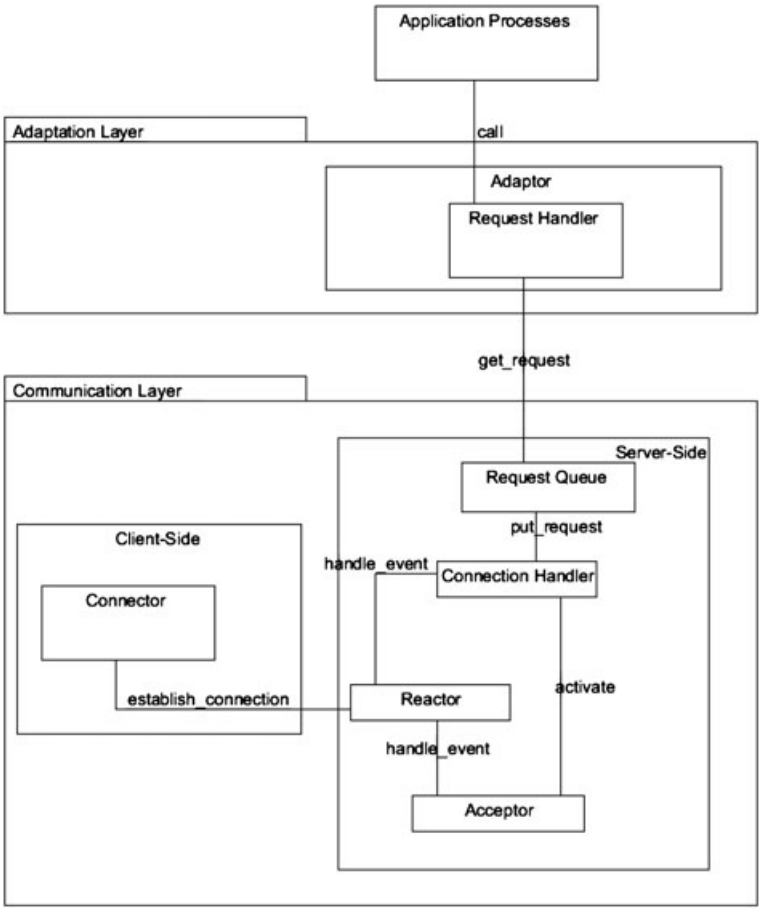

Figure 20. Example architecture design.

We selected the Layers, Client–Server, Reactor, Acceptor–Connector, and Request Handler patterns. We considered these patterns suitable to effectively address the scalability, portability, flexibility, and distribution requirements. As a first step, the communication middleware is implemented as a layered structure: the adaptation layer and the communication layer. When integrating the Reactor and Acceptor–Connector patterns, only the Handler participant from the Reactor pattern will be used to handle events. Such an integration between the Reactor and Acceptor–Connector patterns is done using the *absorbParticipant* relationship as discussed in Section 3.3. In the core communication layer implementation, a component initiates an event loop using the Reactor pattern. When a request event occurs, the Reactor demultiplexes the request to the appropriate event handler. The Reactor then calls the handle event method on the Connection Handler, which reads the request and passes it to Adaptation layer. The *importParticipant* relationship is used to import the participants of the Reactor and Acceptor–Connector patterns into the participants of the Client–Server pattern. Moreover, the Adaptation layer imports the Adaptor pattern using *importPattern* relationship. This layer then demultiplexes the request to the appropriate *call* method. Figure 20 shows the resulting software architecture for combining the earlier listed architectural patterns using pattern participants relationships.

ACKNOWLEDGEMENTS

We would like to thank Neil Harrison, Matthias Galster, and Simon Gieseke for reviewing the architecture documents and providing their valuable feedback to draw conclusions from this study. We are also thankful to Peter van Saten for helping us analyze the statistical data collected after this experiment. Moreover, we

DOI: 10.1002/spe

thank the software patterns course graduate students from the University of Groningen, the Netherlands for their participation in the experiment.

REFERENCES

1. Abowd GD, Allen R, Garlan D. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology, ACM Press* 1995; **4**(4):319–364.
2. Avgeriou P, Zdun U. Architectural patterns revisited - A pattern language. *Technical Report*, 2005.
3. Buschmann F, Henney K, Schmidt DC. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Wiley Series in Software Design Patterns: Chichester, 2007.
4. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. *Pattern-Oriented Software Architecture*, Vol. 1. Wiley and Sons: Chichester, 1996.
5. Janeiro J, Barbosa SDJ, Springer T, Schill A. Enhancing user interface design patterns with design rationale structures SIGDOC '09. *Proceedings of the 27th ACM International Conference on Design of Communication*, ACM, 2009; 9–16.
6. MacDonald S. Design patterns in enterprise. *CASCON '96: Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, 1996; 25.
7. Porter R, Coplien JO, Winn T. Sequences as a basis for pattern language composition. *Science of Computer Programming* 2005; **56**:231–249. Elsevier North-Holland, Inc.
8. Pree W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley: Reading, MA, 1995.
9. Jedlitschka A, Briand LC. The role of controlled experiments working group results. *Proceedings of the 2006 International Conference on Empirical Software Engineering Issues*, Springer-Verlag, 2007; 58–62.
10. Bass L, Clements P, Kazman R. *Software Architecture in Practice 2nd Edition*. Addison Wesley: Boston, 2003.
11. Buschmann F, Henney K, Schmidt DC. Past, present, and future trends in software patterns. *IEEE Software. IEEE Computer Society* 2007; **24**:31–37.
12. Cronk BC. *How to Use SPSS: A Step-By-Step Guide to Analysis and Interpretation Pyrczak Pub*, 4th ed.: Los Angeles, 2006.
13. Eden AH. Precise Specification and Automatic Application of Design Patterns. *International Conference on Automated Software Engineering*, IEEE Press, 1997.
14. Booch G. Handbook of Software Architecture: Gallery, 2010. http://www.booch.com/architecture/architecture.jsp?part=Gallery.
15. Golden E, John BE, Bass L. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, ACM, 2005; 460–469.
16. Giesecke S, Marwede F, Rohr M, Hasselbring W. A style-based architecture modelling approach for UML 2 component diagrams. *Proceedings of the 11th IASTED International Conference Software Engineering and Applications (SEA'2007)*, ACTA Press, 2007; 530–538.
17. Levene H. Robust Tests for Equality of Variances. In *Contributions to Probability and Statistics*, Olkin I (ed.). Stanford University Press: Palo Alto, California, 1960; 278–292.
18. Harrison N, Avgeriou P. Pattern-driven architectural partitioning: balancing functional and non-functional requirements. *ICDT '07: Proceedings of the Second International Conference on Digital Telecommunications*, IEEE Computer Society, 2007; 21.
19. Harrison N, Avgeriou P, Zdun U. *Using Patterns to Capture Architectural Decisions*. IEEE Software, 2007. 38–45.
20. Hamza H, Fayad M. *Towards A Pattern Language for Developing Stable Software Patterns - Part I*. PLoP, 2003.
21. Patterns and Pattern Languages of Program, 2010. http://hillside.net.
22. Janeiro J, Barbosa SDJ, Springer T, Schill A. Enhancing user interface design patterns with design rationale structures. *SIGDOC '09: Proceedings of the 27th ACM International Conference on Design of Communication*, ACM, 2009; 9–16.
23. Kamal AW, Avgeriou P. Mining relationships between the participants of architectural patterns. *4th European Conference on Software Architectures*, ECSA, 2010.
24. Kitchenham BA, Hughes RT, Linkman SG. Modeling software measurement data. *IEEE Transactions on Software Engineering IEEE Press* 2001; **27**:788–804.
25. Gardner ME. Which is the correct statistical test to use? *British Journal of Oral and Maxillofacial Surgery* 2008; **46**:38–41.
26. Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology ACM Press* 2002; **11**:2–57.
27. Fayad M, Altman A. Introduction to software stability. *Communications of the ACM* 2001; **44**(9):95–98.
28. Mikkonen T. Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering*. IEEE Computer Society: Kyoto, 1998; 115–124.
29. Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 2000; **26**(1):70–93.
30. Jedlitschka AP. Reporting guidelines for controlled experiments in software engineering. *ACM/IEEE Proceedings*, 2005; 95–104.

DOI: 10.1002/spe

31. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 2009; **14**:131–164. Kluwer Academic Publishers.
32. Pavliv L, Herièko M, Podgorelec V, Rozman I. Improving design pattern adoption with an ontology-based repository. *Informatica* 2009; **33**:189–197.
33. Sjøberg DIK, Arisholm EM. Conducting experiments on software evolution. *ACM. Proceedings of the 4th International Workshop on Principles of Software Evolution*, Vienna, Austria, 2001.
34. Schumacher M, Fernandez E, Hybertson D, Buschmann F. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley and Sons: New York, 2006.
35. Dustdar S, Gall H. Architectural concerns in distributed and mobile collaborative systems. *SEKE '02: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, ACM, 2002; 521–522.
36. Schmidt DC, Stal M, Rohnert H, Buschmann F. *Patterns for Concurrent and Distributed Objects*. J. Wiley and Sons Ltd: Santa Barbara, California, 2000.
37. Thisted RA. What is a *P*-value? In *Departments of Statistics and Health Studies*. The University of Chicago: South Maryland Avenue, Chicago, 2010.
38. Object Management Group (OMG) UML 2.0 Superstructure. *Final Adopted Specification*, 2003.
39. Zimmer W. Relationships between design patterns. In *Pattern Languages of Program Design*, Coplien JO, Schmidt DC (eds). Reading. MA: Addison-Wesley, 1995; 345–364.

DOI: 10.1002/spe