Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation

Harrison, Neil B.; Avgeriou, Paris

Published in:
EPRINTS-BOOK-TITLE

Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation

Neil B. Harrison[1,2] and Paris Avgeriou[2]

[1] Department of Computing and Networking Sciences, Utah Valley State College, Orem, Utah, USA, harrisne@uvsc.edu
[2] Department of Mathematics and Computing Science, University of Groningen, Groningen, the Netherlands, paris@cs.rug.nl

Abstract

Architecture patterns are an important tool in architectural design. However, while many architecture patterns have been identified, there is little in-depth understanding of their actual use in software architectures. For instance, there is no overview of how many patterns are used per system or which patterns are the most common or most important for particular domains. In addition, little is known of how architecture patterns ay interact with each other. We studied architecture documentation of 47 systems to learn about their architecture patterns. Most systems had two or more architecture patterns, and certain patterns were prominent in different application domains. We identified several patterns that are commonly used together, and are beginning to learn how such combinations may impact system quality attributes. This information can be used to help designers select architecture patterns, can help people learn both architectures and patterns, and can be useful in architectural reviews.

1. Introduction

Architecture patterns are an established tool for designing and documenting software architectures. They are proven approaches to architectural design. Numerous architecture patterns have been identified, based on extensive experience with real systems.

Because the impact of patterns on software architecture is so critical, it is important to understand better how patterns are used in practice, in real systems. There are several things it is important to learn:

How many architecture patterns are usually applied in a system? Is there an optimal number of patterns and what are the effects on the architecture?

Which architecture patterns are most commonly used, especially in certain application domains? This can help people understand which patterns to learn and analyze. Architects that have experience in certain domains usually know the most appropriate patterns for that domain even if it is only implicitly. However it is important to make this knowledge explicit: to map the patterns to the domains they are most frequently used, and to reason about this mapping.

How can patterns help us understand the impact of an architecture on the system's quality attributes? The consequences of individual patterns are in general documented but combinations of patterns make the impact more complex; which pairs are important?

Grady Booch has collected architecture documentation of many software systems [5]. This forms a rich library of diverse systems, representing many different domains. We analyzed the architecture documentation of each of the systems in the library to find the architecture patterns that were apparent from the documentation. We confirmed that patterns are a dominant part of software architectures. We learned which patterns were most common in different domains. We also observed combinations of patterns used, and have begun to study the significance of pattern combinations.

This paper describes the results of this ongoing study. In the next section, we describe the gallery of architecture documentation and our method of analyzing it. Section 3 describes the results, and section 4 lists some limitations of the study. Section 5 describes several uses of the study, and we conclude with future work and conclusions.

2. The architecture gallery

The collection of software architectures is the product of work by Grady Booch in his efforts to create a handbook of software architecture [5]. He states, "The

0-7695-3092-3/08 $25.00 © 2008 IEEE
DOI 10.1109/WICSA.2008.18

primary goal of the *Handbook of Software Architecture* is to fill this void in software engineering by codifying the architecture of a large collection of interesting software-intensive systems, presenting them in a manner that exposes their essential patterns and that permits comparisons across domains and architectural styles." Part of this work has been to collect documentation of architecture of systems for the purpose of analysis. The architecture diagrams have been collected into a gallery of representative systems.

The architecture gallery consists of architectural diagrams from 47 different software systems from a wide variety of applications. The diagrams were all box-and-line diagrams, and each diagram had its own semantics for the boxes and lines. We mapped the diagrams to the 4+1 architecture views from Kruchten [17] as shown in Table 1. This view model was chosen because it is widely known, but there can be other mappings according to other view models, such as the ones described in [7]. Most diagrams included elements from several of the 4+1 views; however, in most cases, we were able to identify one view that was more prominent than the others. The development and process views were the most commonly used views. We believe this is partly due to the nature of the documents from which they came – the audience was software professionals, not managers or customers. So they would tend to be more interested in views that support implementation.

Table 1: Most Prominent of the 4+1 Architecture Views per system in the Architecture Diagrams

View	Frequency
Logical	6
Development	18
Process	11
Physical	8
No View Dominant	4

We classified the systems into seven application domains. (Some systems could fall into more than one domain; in that case, we chose the primary domain.) The application domains were as follows:

1. Embedded Systems: These were systems built to run in special purpose machines, often on specialized hardware. Examples included a pacemaker and a fire alarm. We found 11 such systems.
2. Dataflow and Production: These systems support assembly-line type operations, as well as processing and responding to streaming data. Examples include manufacturing support, automated cargo routing, process monitoring, and air traffic control. There were 10 systems in this domain.
3. Information and Enterprise Systems: These were data processing applications. There were 9 systems.
4. Web-based Systems: These were web applications and servers. There was some overlap between this category and the Information and Enterprise Systems, because some of them are web-based. The systems here were those where web interaction was the primary function. There were 5 such systems.
5. CASE & Related Developer Tools: Tools designed to help software developers comprised this category. There were 4 such systems.
6. Games: There were 4 interactive game systems.
7. Scientific Applications: These systems were marked by their diversity: they included a system for chemical calculations and a speech system. There were 4 scientific applications.

We studied the architectures in the following manner:

1. We began by studying an architecture diagram, looking for architecture patterns that were used. The most obvious clue was the use of the name of an architecture pattern in the diagram. In 12 out of the 47 architectures, architecture patterns were explicitly named.
2. We used the names of the boxes and the lines between them to understand their functions and their collaborations with other components. We looked for mappings to components and connectors found in the architecture patterns.
3. Where possible, (in 37 out of 47 cases) we followed up on the references to additional documentation that were associated with each pattern diagram in [5] to confirm the patterns we found. In all cases, the documentation consisted of architectural descriptions from which the diagrams were taken.
4. We consulted the architecture pattern literature [2, 6, 22, 23, 24] to confirm the presence of patterns and to look for other patterns.
5. Finally, we made another pass through all the architecture diagrams to confirm the patterns selected, and to resolve the patterns that were uncertain. If it was still unclear, we omitted the pattern from the final list.

Note that we did not consider the domain classification as we identified the patterns; in fact we classified the systems into domains after we studied them. This was done in order to avoid bias that might creep in by assuming that systems in a certain domain should have certain patterns. In addition, we attempted to avoid bias by studying them in the order they are given on the website – alphabetical order.

3. Findings

Analysis of the data led to a number of insights about typical pattern usage. They are described in the following sub-sections.

3.1. Pattern Density

Patterns were very prevalent in the diagrams. Every system had at least one pattern. A total of 110 patterns were identified. The average number of patterns used per system was 2.36, and the mode (most frequent) was 2 patterns. The distribution of pattern density was as follows:

Table 2: Pattern Density per System

Number of Patterns Found	Number of Systems
1	10
2	22
3	9
4	4
5	0
6	1
7	0
8	1

We note that over 85% of the systems used between one and three architecture patterns.

3.2. Pattern Frequency

Twenty different architecture patterns were identified. Their frequency is shown in Table 3. It is noted that not all architecture patterns have standardized names that are universally adopted. We used the names from the classification in [2], except for the patterns State Transition [24] and Half Sync, Half Async [22].

Table 3: Pattern Frequency

Pattern Name	Frequency
Layers	18
Shared Repository	13
Pipes and Filters	10
Client-Server	10
Broker	9
Model-View-Controller	7
Presentation-Abstraction- Control	7
Explicit Invocation	4
Plug-in	4
Blackboard	4
Microkernel	3
Peer to Peer	3
C2	3
Publish-Subscribe	3
State Transition	3
Interpreter	2
Half Sync, Half Async	2
Active Repository	2
Interceptor	2
Remote Procedure Call	1
Implicit Invocation	1

The most popular patterns (higher frequencies in Table 3) did not come as a surprise, as they also receive the greater attention in the software architecture literature. It is more interesting and useful to investigate the patterns in particular domains.

3.3. Pattern Density by Domain

We found that the domains showed very little difference in the median number of patterns per system: all were close to 2 patterns per domain. The table below shows which patterns were most prominent in the different domains. With the exception of the scientific domain (see note below), the table included patterns that appear twice or more in a domain.

Table 4: Prominent Patterns per Domain

Domain	# Sys	Pattern	Freq
Embedded Systems	11	Pipes & Filters	4
		Client-Server	3
		Presentation Abstraction Control	3
		State Transition (note 1)	3

		Explicit Invocation (note 2)	2
		Shared Repository	2
		Layers	2
Dataflow and Production	10	Layers	6
		Shared Repository	3
		Presentation Abstraction Control	2
		Broker	2
		Plugin	2
Information and Enterprise Systems	9	Shared Repository	6
		Layers	4
		Model View Controller	4
		Presentation Abstraction Control (note 3)	2
		Broker (note 4)	2
Web-Based Systems	5	Broker (note 5)	2
		Layers	2
		Pipes & Filters (note 6)	2
		Explicit Invocation	2
CASE & Related Developer Tools	4	Layers	4
		Broker	2
Games	4	Model View Controller	2
		Pipes & Filters	2
		Blackboard	2
Scientific Applications	4	Pipes & Filters	1
		Shared Repository	1
		Client-Server	1
		Presentation Abstraction Control	1
		Blackboard	1
		Layers	1
		Plugin	1

Notes about the table are as follows:

Embedded Systems:

Embedded systems often have important realtime constraints. In addition, they usually have limited resources such as memory and external storage. This makes the Pipes and Filters pattern a good match: it is often efficient, and can be configured to use little intermediate storage. The systems are often state-based. It appears that some embedded systems are getting more powerful, and have some capabilities for human interfaces and storage, which accounts for the Client-Server, Shared Repository, and Layers patterns.

1. The State Transition pattern [24] is probably even more prevalent than indicated here. In particular, one of the systems was an ATM, which is likely state-driven. However, the pattern was not apparent in that system's diagram.
2. Explicit Invocation is useful where the components have tight coupling. The constraints of embedded systems tend to lead to tight coupling, so this pattern is appropriate here.

Dataflow and Production

Like embedded systems, dataflow and production systems often have some realtime concerns, but do not have the limitations of embedded systems, such as limited memory and hard realtime constraints. They can be more general purpose and more interactive; usability and extensibility may be significant attributes. Therefore, patterns related to data processing, such as Layers, Shared Repository, and Presentation Abstraction Control are prominent.

Information and Enterprise Systems

In these systems, data processing is central, hence the dominance of Shared Repository. Maintainability, extensibility, usability and security are important. This justifies again the use of Shared Repository as well as Layers. User interface needs are supported by Presentation Abstraction Control and Model View Controller.

3. Presentation Abstraction Control and Model View Controller are alternate approaches to improve the modifiability of GUI-based systems; they were observed 6 times in these systems.
4. The presence of Broker indicates that some of these systems are distributed.

Web-Based Systems

The web-based systems in this sample were generally limited to the server side. Key quality attributes would include performance, extensibility, security, and

availability. The pattern distribution is quite flat, with Broker, Layers, Pipes and Filters, and Explicit Invocation all showing equal prominence. These are all well suited to web systems. This category is somewhat small, but several information systems and CASE tools are web based as well.

5. It is a bit surprising that Broker and Client-Server were not observed in most web-based systems. However, many web-based systems have thin clients, where the client is architecturally insignificant (e.g., the user's web browser).
6. This might be a bit surprising for web-based systems. However, many web-based systems have significant performance needs; our sample included the Google search engine, for example.

CASE and Related Developer Tools

Our sample included only four such systems. Each of them used the Layers pattern, which supports maintainability, extensibility, security, and flexibility; all important to these tools. The Broker pattern probably indicates that two of the systems support distributed operation. Overall, these systems used few patterns; perhaps they are too specialized and diverse to incorporate very many patterns.

Games

The patterns in games systems show the different important aspects of gaming systems. User interface is very important, hence the Model View Controller pattern. Performance is also critical, so the Pipes and Filters pattern was used in some systems. It was particularly interesting to see that the game systems were developed in different years (1996, 2000, and two in 2004), Over time, the systems became more complex, and more patterns were observed. The last two used the Blackboard pattern, surely to support artificial intelligence.

Scientific Applications

The scientific applications were a very diverse group, and no patterns were prominent. With a larger sample of scientific systems, the domain might be divided into sub-domains, which might have some prominent patterns. We see evidence of artificial intelligence (Blackboard) as well as high-performance computing (Pipes and Filters.)

3.4. Pattern Interactions

One of the areas of research in architecture patterns is how the patterns interact with each other. In particular, architecture patterns have well-understood and often significant impacts on various quality attributes of a system. Because the quality attributes can be key characteristics of a system, it is important to understand how architectural decisions (including the use of patterns) impact them [12, 15]. While the impact of individual architecture patterns on quality attributes has been studied [3, 6, 8, 13, 20, 24], little is known about the combined impact of architecture patterns. The impact, though, of multiple patterns can also be significant. One pattern may impact a quality attribute in a positive way, while another pattern may impact the same quality attribute in a different way. On the other hand, both patterns may impact the quality attribute in similar ways. If the impact is positive, this can be very good; if it is negative, the combined impact could spell disaster for the system. For example, the Layers pattern tends to have a negative impact on performance, as does Presentation Abstraction Control. If used together, the performance impact could be significant.

Avgeriou and Zdun have cataloged 25 different architecture patterns [2]. In addition, there are certainly other architecture patterns as well [4, 10]. This gives hundreds of different possible pairs of architecture patterns to examine. Therefore, it makes sense to identify the pairs of patterns most commonly used together, and examine their interactions. The most frequent pairs of patterns are listed in Table 5.

The majority of the systems we studied had two or fewer patterns: only 15 out of the 47 had more than two patterns. This means examination of pairs of patterns will be fruitful, while it is lower priority to examine triplets of patterns or groups of even higher degree. In addition, even where more than two architecture patterns are used in a system, it will certainly be helpful to study the pairwise interactions of patterns. The most common pairs of architecture patterns we identified were as follows:

Table 5: Frequency of Architecture Pattern Pairs

Pattern Pair	Frequency
Layers – Broker	6
Layers – Shared Repository	3
Pipes & Filters – Blackboard	3
Client Server – Presentation Abstraction Control	3

Layers – Presentation Abstraction Control	3
Layers – Model View Controller	3
Broker – Client-Server	2
Shared Repository – Presentation Abstraction Control	2
Layers – Microkernel	2
Shared Repository – Model View Controller	2
Client Server – Peer to Peer	2
Shared Repository – Peer to Peer	2
Shared Repository – C2	2
Peer to Peer – C2	2
Layers – Interpreter	2
Layers – Client Server	2
Pipes & Filters – Client Server	2
Pipes & Filters – Shared Repository	2
Client Server – Blackboard	2
Broker – Shared Repository	2
Broker – Half Sync/Half Async	2
Shared Repository – Half Sync/Half Async	2
Client Server – Half Sync/Half Async	2

There were a large number of pattern pairs that appeared once; 67 in all. However, the vast majority of these pairs, 51, came from the systems with four or more patterns. These represent less than 15% of the systems we examined, so these pattern pairs are expected to be unusual.

Many of the most common pairs seem to logically go together, in particular, the following pairs:

1. The Layers – Broker connection is very common among distributed applications. A server is often implemented with Layers. The Broker pattern coordinates communication from remote clients. The patterns also have a symbiotic relationship with respect to security: layered systems often have an outer authentication layer, while the Broker can provide some protection against intrusion attacks. Brokers may be implemented in layers [27], and layered applications are common in distributed communication (e.g., the ISO/OSI stack.)
2. The Broker and Client-Server patterns are a natural pair for distributed applications. The Client-Server pattern defines the relationship between providers of services and consumers of the services (clients.) Brokers coordinate the communication between them. Broker is also used to implement efficiency, availability, and security, which are common concerns in client-server systems.
3. A Shared Repository is obviously designed to allow multiple access, but may need security and some access restrictions. Layers are often used to provide an interface to a Shared Repository, and can thus provide the necessary security and other screening of requests.
4. Many interactive applications are designed as layered systems. Management of the user interface is often encapsulated in a separate component, becoming the outer layer. The user interaction can be managed using the Model View Controller pattern, with the bulk of the layered system becoming the Model, and the interface (outer layer) managed with the View and Controller. Alternatively, the layered system can present an abstract representation of the user interface functionality for use with the Presentation Abstraction Control pattern. Each of these alternate user interface patterns was paired with the Layers pattern three times.
5. The Presentation Abstraction Control pattern was also paired with Client-Server three times. Different clients may require different presentations of the functionality from the server. This can be accomplished by using the Presentation component, with the Abstraction and the Control residing with the server.

Other common pairs of patterns have less apparent relationships, and further research is required to determine their relationships. Possibly the combination of patterns in some cases cannot be generalized, but specific variants of patterns were combined to satisfy individual system requirements.

In the pairs described above, some of the symbiosis appears to be related to quality attributes, such as performance, usability, and security. It appears that in many cases, the patterns provide structure to implement some functionality, but one pattern may have particular impact on important quality attributes of the system. For example, the Client-Server pattern defines the basic structure, and the Broker pattern adds some functional capabilities, but is particularly strong in the quality attributes. It may be that some patterns are selected primarily for their functional capabilities, while others are selected in order to support quality attributes. (We have observed this once in a system design experiment, in which participants selected the Layers pattern for functionality, and the Broker pattern for availability and security [14].) While research into the interaction among architecture patterns is in its

infancy, this may be an indicator of how architecture patterns work together.

Since the most common number of patterns in a system was two, it is instructive to look at the pairs of patterns that appeared as pairs in these systems. These are likely to be smaller, less complex systems than those with more than two patterns. Only a few were used more than once. They are listed below, with the frequency of appearance.

Table 6: Pairs of Patterns in Systems with Two Architecture Patterns

Pattern Pair	Frequency
Layers – Broker	4
Layers – Model View Controller	2
Client-Server – Presentation Abstraction Control	2
Layers – C2	1
Pipes & Filters – Shared Repository	1
Active Repository – Presentation Abstraction Control	1
Pipes & Filters – Blackboard	1
Client-Server – Plugin	1
Layers – Presentation Abstraction Control	1
Shared Repository – Presentation Abstraction Control	1
Broker – Plugin	1
Layers – Plugin	1
Layers – Microkernel	1
Broker – Client-Server	1
Pipes & Filters – Interpreter	1
Pipes & Filters – Shared Respoitory	1
State Transition – Shared Repository	1
State Transition – Layers	1

We note that the Layers pattern is particularly prominent: 11 out of 23 pairs involve Layers. It is paired with 7 different patterns. While this is to be expected due to the frequency of the Layers pattern overall, it does underscore the compatibility of Layers with other patterns for relatively simple systems (where only two architecture patterns are used,) as well as for all systems.

On the other hand, the second most common pattern, Shared Repository, is paired with few patterns in two-pattern systems. It was most common in larger systems, such as information and enterprise systems.

We compared the pattern interactions we discovered to the pattern interactions suggested in published literature. Not much has been published about pattern interactions, except for the work in [2], which describes a rich set of interactions. The relationships are varied, including use, is-a, variants, realizes, and alternatives. In a single architecture, only the "use" relationship is visible: all the others concern selection of a pattern (e.g., selecting between two alternate patterns.) From all the pattern pairs in Table 6, only three match the "use" relationships described in [2]. The reason we found many more pairs is likely because we were examining complete architectures which must create complete solutions. This introduces rich possibilities for multiple patterns to be used. Further research will be useful to determine which of the pattern pairs are most useful.

4. Limitations

There are several limitations in this study which should be taken under consideration. In some cases, further research is warranted.

4.1. Pattern Identification

We saw that identification of the patterns depends on several different factors. These factors can significantly increase or decrease the number of patterns identified in a system. These factors assume that the architecture documentation is being examined without the help of the original (or even current) architects – obviously, the ideal is that the architect personally explains the architecture and the patterns used. Without the benefit of communication with the architect, the following factors should be considered.

Additional documentation beyond the architecture diagrams was helpful in identifying some patterns, but it did not add as much as one might expect. Part of the study illustrates this: Fifteen of the systems came from Fayad et al [11]. The diagrams were analyzed first, and then the book was consulted to verify the analysis. In the subsequent analysis, four systems had patterns added, two had questionable patterns removed, and nine stayed the same.

Some of the diagrams were easier to understand than others, and where the diagrams were difficult to understand, it was difficult to discover the patterns. It was difficult to figure out what made some diagrams more difficult than others, but it appeared that the more abstract the diagram was, the harder it was to understand. In order to increase visibility of the patterns in diagrams, architects might annotate the module with the patterns used, use the pattern name in the name of the module, or capture the patterns used with an architectural decision capture tool (see [16].)

It stands to reason that the more experience an analyst has with the architecture patterns, the more likely he or she is to find patterns. We found this to be true: a novice and an expert studied the same patterns, and the novice found a subset of the patterns found by the expert. In addition, it is likely that familiarity with certain patterns by the analyst leads to their ready identification. This illustrates the proverb that if all you have is a hammer, everything looks like a nail. Less well-known patterns are likely to be found less often. They are also less likely to be used in the first place. For example, the C2 pattern was identified in two systems , and an author of the C2 pattern was involved in both systems [19, 21, 26]. Other people might not recognize or use it.

An important consideration is how accurate is the pattern identification, and what are the consequences of wrong answers. It is impossible to judge completely without a detailed architecture review, including conversations with the architects. However, we were able to assess the accuracy of pattern identification based only on the diagram versus analysis of both the diagram and supporting documentation. We examined 15 systems that were described in Fayad et al [11] first based only on the diagrams, and then augmented by the documentation. We found that of the patterns identified, 60% were identified correctly, 27% were missed, and 13% were over-identified (identified, but not really present.) This gives us moderate confidence in pattern identification from the diagrams alone, and good confidence if additional documentation is used.

The consequences of missing patterns would be that less information about the utility of patterns might be available. The consequences of over-identification might be that a pattern is used in a domain where it is not appropriate. We have attempted to minimize this possibility by focusing on patterns that are found frequently in domains, as described above.

4.2. Pattern Density

While the majority of the systems contained either one or two patterns, several contained more; as many as 8 patterns. Since patterns are vehicles of system comprehension, multiple patterns can aid in understanding all the important components of the system. However, too many patterns can actually hinder analysts from identifying the patterns and their functions. This happens when patterns begin to overlap – when system components fill functions in multiple patterns.

On the other hand, the absence of any identifiable patterns in an architecture can also hinder comprehension – one cannot relate it to any established approach to the architecture. Fortunately, all the systems contained patterns, though we have observed a subsystem that had no patterns, and it was difficult to comprehend.

There may be a “sweet spot” for the number of architecture patterns in an architecture – the optimal number of patterns to aid design as well as comprehension. Our data indicate that most systems have between one and three patterns; since this is common practice it may be considered as a good design heuristic. However. it may be worthwhile to study whether this is the optimal number of patterns in a system; it may depend on factors such as system size and the problem domain.

4.3. Sample Coverage

How well does this sample represent the body of software in the world? The coverage is broad, but not very deep. In particular, the sample does not attempt to represent a profile of the world’s software, which is undoubtedly skewed in a few areas, such as (currently) web applications. In this case, we believe that breadth of study is more helpful than depth, since a wider body of knowledge is being created.

4.4. Pattern Variations

As patterns are used, they are often changed from the “pure” definition given in the pattern literature. It appeared that many of the patterns were used with variation. However, we found it difficult to identify whether a pattern was used in its “pure” form, or whether it was used with variation – such detail was not to be found in either the diagrams or the supporting documentation. Variations to patterns are often made to accommodate the quality requirements of the system, such as performance, security, reliability, etc. Some of the most common variations are published in the patterns; mainly in Buschmann et al. [6]. However, the usage of pattern variants may well make the pattern’s use in the system more difficult to understand. By the same token, the variant use of patterns almost certainly made it harder for us to identify the patterns, and may have obscured some patterns altogether. But this makes sense: if a pattern’s use is changed so much that the pattern becomes difficult to recognize, then one can argue that the pattern is no longer being used at all.

5. Practical Uses

There are several potentially important uses for this information. It is a contribution to the idea of a software architecture handbook, used to guide architects and developers in proven practices of software development. Booch notes, "It is a sign of maturity for any given engineering discipline when we can name, study, and apply the patterns relevant to that domain." Software maintenance consumes much of the total cost of software development, and one of the most expensive activities in maintenance is understanding the software.

Architecture patterns are a valuable tool in creating the system architecture. Numerous architecture methods accommodate the use of patterns. In addition, architectural design methods that focus on architecture patterns are being developed [12]. This work can make these processes more effective by showing system designers which patterns are commonly used, particularly in certain domains. It can also point to common combinations of patterns.

Architecture patterns are also emerging as an important tool in the software architecture review process. A prominent part of software architecture development is the architecture review [1, 8, 18]. We are developing review methods that focus on patterns, with an emphasis on the impact of the patterns on the system's quality requirements. This work contributes by identifying the most common patterns, their frequency in certain domains and most frequent combinations of patterns.

This study has educational applications as well. Software architecture, including architecture patterns, are beginning to be taught in graduate and undergraduate computer science and software engineering curricula [25]. This work can give guidance about which patterns are most commonly used, in which application domains, and which patterns work well together as pairs.

6. Future Research

Additional studies of architectures, including more from Booch's architecture gallery can be added to further validate this work.

More detailed study of pattern-based evaluation of architectures can be done. In particular, it would be useful to analyze a system's architecture for patterns and get feedback from the architects about the patterns found. We have begun to do this as part of the pattern-based architecture review process we are developing, and early results confirm its accuracy and usefulness.

The area of patterns' impact on system quality attributes is an important area of study. Studies have been done [9, 20] and are ongoing [13] to understand the impact of individual patterns. The impact of combinations of patterns, and more generically, approaches (see [8]), appears to be particularly important to quality attributes. Further study into the frequency of pattern combinations and their impact on quality attributes will make it possible to study the impact of the most common pattern combinations. This is one of the most important areas of long term study.

This work underscores the importance of studying pattern variants. Existing pattern variants should be studied both at a structural level and for their impact on quality attributes. In addition, undocumented pattern variants should be studied and common undocumented variants should be documented.

We noted that the architecture diagrams represented different views of the systems, and most incorporated elements of more than one of the 4+1 views. It is possible that architecture patterns are more readily apparent in different views. We intend to study which views match certain patterns better in the sense that the patterns become more visible and explicit. Ideally one specialized view per pattern would solve this problem, but the combination of patterns can complicate things.

Finally we have observed the difficulty in identifying patterns in system diagrams due to the lack of pattern names or individual pattern element names. We intend to look into what are the best practices that architects follow in documenting the use of specific patterns in architecture diagrams. In specific we are looking for simple yet efficient ways to "annotate" diagrams with information on patterns and their variations used.

7. Conclusions

Architecture patterns are an important tool of software architects. This work confirms that nearly every system contains one or more of the known architecture patterns; in most cases two or more patterns. It begins to shed light on which architecture patterns are most commonly used. It also identifies commonly used pairs of architecture patterns. This information can be used to help architects design systems, students learn architecture methods, and reviewers identify quality attribute-related issues associated with architecture patterns.

8. Acknowledgments

We would like to thank Grady Booch for promoting a software architecture handbook, and for assembling the collection of software architectures.

9. References

[1] Abowd, G. et al., "Recommended Best industrial Practice for Software Architecture Evaluation", *Technical Reoprt CMU/SEI-96-TR-025*, CMU Software Engineering Institute, January 13, 1997.

[2] Avgeriou, P. and Zdun, U. "Architectural Patterns Revisited – a Pattern Language", *10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, Irsee, Germany.

[3] Bass, L,, Clements, P. and Kazman, R. *Software Architecture in Practice* Addison-Wesley, 2003.

[4] Bianco, P., Kogtermanski, R., L., and Merson, P. *Evaluating a Service-Oriented Architecture* ,(CMU/SEI-2007-TR-015) Pittsburgh, PA: Software Engineering Institute, 2007.

[5] Booch, G. "Handbook of Software Architecture: Gallery" http://www.booch.com/architecture/architecture.jsp?part=Gallery accessed 10 September 2007.

[6] Buschmann F. et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[7] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R. & Stafford, J. *Documenting Software Architectures: Views and Beyond* Addison-Wesley, 2002.

[8] Clements, P., Kazman, R. and Klein, M. *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, SEI Series, 2002.

[9] Cloutier, R. *Applicability of Patterns to Architecting Complex Systems*, Doctoral Dissertation, Stevens Institute of Technology, 2006.

[10] Cockburn, A. "Hexagonal Architecture", http://alistair.cockburn.us/index.php/Hexagonal_architecture, accessed September 11, 2007.

[11] Fayad, M. and Johnson, R. *Domain-Specific Application Frameworks*. Indianapolis, Indiana: Wiley, 2000.

[12] Harrison, N and Avgeriou, P. "Pattern-Driven Architectural Partitioning - Balancing Functional and Non-functional Requirements", *First International Workshop on Software Architecture Research and Practice (SARP'07)*, July 2007 – San Jose, CA, IEEE Computer Society Press.

[13] Harrison, N and Avgeriou, P. "Leveraging Architecture Patterns to Satisfy Quality Attributes", *First European Conference on Software Architecture*, Madrid, Sept 24-26, 2007, Springer LNCS.

[14] Harrison, N. Avgeriou, P. and Zdun, U. "Focus Group Report: Capturing Architectural Knowledge with Architectural Patterns", *11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, Irsee, Germany.

[15] Harrison, N., Avgeriou, P. and Zdun, U. "Architecture Patterns as Mechanisms for Capturing Architectural Decisions", *IEEE Software,* 24(4) 2007.

[16] Jansen, A. and Bosch, J. "Evaluation of Tool Support for Architectural Evaluation", *Proc. of the 19th IEEE Intl Conference on Automated Software Engineering (ASE2004),* Sept 2004, Linz, Austria, pp. 375-378.

[17] Kruchten, P. "The 4+1 View Model of Architecture", *IEEE Software* 12(6) 1995.

[18] Maranzano, J. et al., "Architecture Reviews: Practice and Experience", *IEEE Software*, 22(2) 2005, pp. 34-43.

[19] Medvidovic, N. et al., "Software Architectural Support for Handheld Computing", *IEEE Computer*, Sept 2003, p 69.

[20] O'Brien, L., Bass, L., and Merson, P. *Quality Attributes and Service-Oriented Architectures,*(CMU/SEI-2005-TN-014) Pittsburgh, PA: Software Engineering Institute, 2005.

[21] Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. & Wolf, A. "An Architecture-Based Approach to Self-Adaptive Software." *IEEE Intelligent Systems*, May/June 1999, p. 58.

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

[23] Shaw, M. "Toward Higher-Level Abstractions for Software Systems", *Proc. Tercer Simposio Internacional del Conocimiento y su Ingerieria* (Oct 1988) 55-61. Reprinted in *Data and Knowledge Engineering*, 5 (1990) pp. 19-28.

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

[25]SEI: "Resources for Educators in Software Architecture" www.sei.cmu.edu/architecture/educators.html, accessed 27 November 2007.

[26] Taylor, R., Medvidov, N. et al., "A Component- and Message-based Architectural Style for GUI Software", *IEEE Trans. Softw. Eng.* 22(6):390-406, 1996.

[27] Voelter, M., Kircher, M. and Zdun, U. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, Wiley, 2004.