Pattern-oriented Knowledge Model for Architecture Design

Kiran Kumar, Prabhakar T.V.
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur, India
{vkirankr, tvp}@iitk.ac.in

Abstract. Software design patterns document the most recommended solutions to recurring design problems. Selection of the best design pattern in a given context involves analysis of available alternatives, which is a knowledge-intensive task. Pattern knowledge overload (due to the large number of design patterns) makes such analysis difficult. A knowledge base to generate available alternatives can alleviate the problem. In this paper, we propose a pattern-oriented knowledge model which considers four dimensions of the pattern knowledge space: Pattern to Tactic relationship, Pattern to Pattern relationship, Pattern to Quality-attribute relationship and Pattern to Application-type relationship. We perform analysis of these relationships for patterns in the two popular pattern catalogues viz GoF and POSA1.

1 Introduction

The problem of software design (Figure 1) can be stated as how to identify and choose design alternatives [5, 26] †. Since patterns document best practices built on tried and tested design experience, they play an important role in design and documentation. Some of the benefits of using patterns are discussed in [8, 9, 19, 20] - these include: Ease of knowledge transfer between designer and developer, Ease of early analysis of design decision consequences, Well-defined support for forward-engineering, Ease of recovering design decisions from different views etc. Hence, design alternatives are commonly extracted from pattern catalogues. Based on that we have a three-stage process that allows us to build a pattern-oriented design view that gives designers an orientation of the design direction to follow.

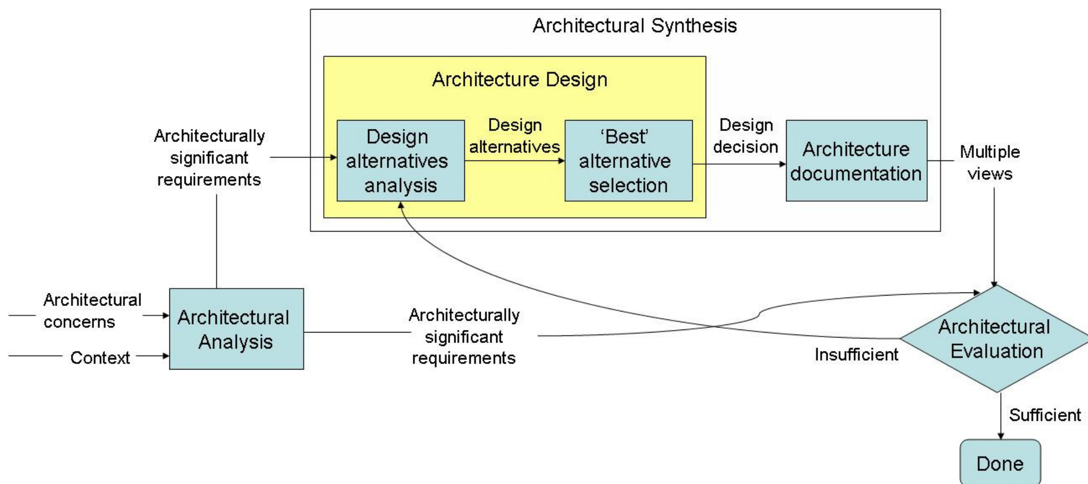

Figure 1. Integrated view of Architecture design and development processes.

† *Len Bass [5] stresses the role of design alternatives analysis during architecture design process. Hofmeister and colleagues [26] abstracted the process commonality from five different architecture development processes.*

Since the design decision (best design alternative) at a particular design context is bound to one of the analyzed set of alternatives, missing an important design alternative can sometimes impact the selected design decision. For example, consider the following design context: *Reduce response time of a web-based system where network speed is bottleneck*. As a solution, as many as three design alternatives can be generated: *Client-side caching*, *Increase server resources* and *Increase parallelism at server*. Since network speed is a bottleneck, *Client-side caching* seems to be a better option than the other two. Suppose during the alternative analysis phase *Client-side caching* is not considered, *Increase server resources* or *Increase parallelism* will emerge as solutions, which may not be desirable.

Analysis of the design alternatives is a knowledge-intensive task; Pattern knowledge overload [23, 24, 29, 41] hardens this analysis. Sometimes designers choose recently used design decisions when a thorough alternative analysis is not possible. Under these circumstances, designers can benefit by a competent knowledge base to generate available alternatives. The tools which integrate such knowledge base as one of their components are termed *Design Assistants* and are current the subject of active research [3, 6, 45, 42].

When architecture design knowledge is codified appropriately, the alternatives analysis problem can be modeled as an information retrieval problem. In this paper, we focus on codifying an important part of patterns knowledge which includes essential design concepts such as: *Patterns*, *Tactics*, *Quality requirements*, *Quality attributes*, *Application type*. Figure 2 illustrates the concepts and relationships of our knowledge model.

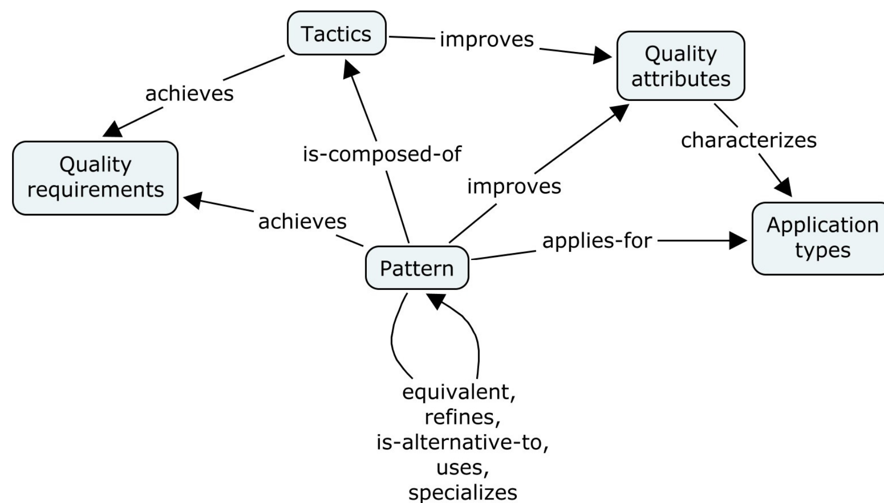

Figure 2. Concepts and relationships of Pattern Oriented Knowledge Model.

In addition to being intuitive, our knowledge model provides other benefits such as *expressiveness* [25], *visualization* [35] *and* can be easily built using *editors* like Cmap [51], Protégé [52], VUE [53] etc. We term our knowledge model, *Pattern Oriented Knowledge Model* (POKM).

One basic difference of our analysis and others' is that *we analyze patterns from a bottom-up perspective;* our analysis is based on an underlying tactics based formal model of patterns. When compared with Booch's knowledge model [6], we focus on analyzing concepts like *Pattern relationships*, *Quality Requirements* and *Quality Attributes* of patterns. The knowledge model of VanHilst et al. [45] focus on the security aspects of different applications; hence their knowledge model contains some specialized concepts like *Code source*, *Architecture layer*. When our knowledge model is compared with VanHilst and colleagues, we focus on analyzing concepts like *Primitive quality requirements* and *Quality attributes* of patterns; also, the pattern catalogue we analyze differ from theirs. When Zimmer's knowledge model [50] is compared with our knowledge model, we use a different analysis method to analyze relationships between patterns; we also focus on inter-catalogue pattern relationships. Tichy's knowledge model [43] focuses on analyzing commonality in *Quality requirement* of patterns; we analyze all the underlying *Quality requirements* and *Quality attributes* of a pattern and we classify them as *Primary* and *Secondary Quality attributes*. Table 1 compares our knowledge model with existing popular design knowledge models.

Table 1. Comparison of our knowledge model with existing popular design knowledge models.

Concepts	Knowledge models				
	Booch [6]	**VanHilst et al. [45]**	**Zimmer [50]**	**Tichy [43]**	**our KM**
Application type	✓	✓			✓
Lifecycle stage	✓	✓			
Technology	✓				
Design view	✓				
Quality Response		✓			
Code source		✓			
Constraint		✓			
Architecture layer		✓			
Pattern relationships			✓		✓
Quality Requirements				✓	✓
Quality Attributes					✓
Tactics					✓
Formal analysis approach					✓
Patterns catalogue	Various sources	Security patterns	GoF patterns	POSA1, PLoP patterns	GoF, POSA1 patterns

The rest of the paper is structured as follows: Section 2 provides the required background terminology. Section 3 presents some necessary details of the decision view of architecture. In section 4, we provide some supporting arguments to our classification of tactics as building blocks of patterns. In section 5, we discuss the details of our analysis and present our analysis results; we also discuss the usefulness of our knowledge model with different design queries. Section 6 discusses related work and section 7 concludes the paper.

2 Terminology

In this section, we review some of the software architecture terminology used in this paper.

- **Quality requirement** [31]: is a requirement which is not specifically concerned with the functionality of the software. Quality requirements specify the external constraints the software should meet. *Fault detection*, *Reduce response time*, *Protect confidential data* etc are some examples of quality requirements.
- **Quality Attribute** [4]: is a set of related quality requirements. *Availability*, *Performance*, *Security*, *Usability* etc are some examples of quality attributes.
- **Design Alternative** [5]: is one of many possible strategies that realize the given set of requirement(s). For example, *Active redundancy*, *Passive redundancy* and *Spare* are different design alternatives to reduce downtime of the system.
- **Design Decision** [5]: is a design alternative that is chosen or applied to realize the requirement(s). For example, *Active redundancy* is the design decision used to ensure minimal downtime of the system.
- **Tactic** [4]: A tactic is a design decision that influences the control of a quality attribute parameter. For example, *Increase available resources* design decision (upgrading 512 MB RAM to 1 GB RAM) controls (minimizes) the response time parameter.
- **Implication/Consequence** [44]: A design decision comes with many implications. For example, a design decision might introduce a need to make other decisions, create new requirements, or modify existing requirements; pose additional constraints to the environment. For example, *Increase available resources* tactic which is one way to achieve *Reduce response time* quality requirement imposes side-effects like *Increase in cost*, *Change in resource management (scheduling) policy* etc.

- **Tactic Topology Model (TTM)** [30]: A graph based representation of semantics of a pattern, where tactics are the nodes of the graph and edges are the dependencies (based on consequences) between the tactics. The TTM of *Observer* pattern can be seen in Figure 4, Section 5.1.

3 Decision view of software architecture

Decision view provides a higher abstraction-level description to the architecture than its module view [12, 28]. Decision view provides a first class representation of design decisions and various relationships among them [28, 12]. The metamodel of a decision view defines the attributes of a design decision and the set of relationships among design decisions. The Bosch et al. [28] decision view metamodel consists of *dependency* and *refines* relationships. The Kruchten [32] decision view metamodel provides a richer set of relationships such as *constrains*, *subsumes*, *comprises* etc; this metamodel also provides various attributes for a design decision such as *scope*, *state*, *cost* etc.

Remco and colleagues [39] analyze the core metamodel of the decision view. *Dependency* relationship is considered as one of the important relationships in the core metamodel. The dependency relationship primarily provides rationale for existence of a particular design decision. Information of design decision dependencies becomes necessary during architecture evolution. As the architecture evolves some design decisions need to be removed; dependency relationship allows safe-undo of a design decision [28, 12, 38] i.e., when a design decision is removed, all its dependant design decisions also need to be removed.

The design decision dependency can be captured in two types of relationships: *Constrains* and *Traces-from*. The *Constrains* relationship represents the dependency between two design decisions; the *Traces-from* relationship represents the dependency between a context and a design decision. Kruchten [32] defines the *Constrains* relationship as follows: "*Decision B is tied to Decision A, if decision A is dropped, then decision B is dropped*" and *Traces-from* relationship is defined as follows: "*Design decisions trace from upstream technical artifacts: requirements*" [32].

Harrison and colleagues [19, 20] propose that patterns can be used to codify design decisions of an application. They also mention that patterns capture one class of design decisions that are related to quality improvement. Other types of design decisions such as those related to technology (such as selecting specific technology) and organization (such as company guidelines or project team setup) may not be captured using design patterns [19].

4 Synergistic relationships between Architectures, Patterns and Tactics

The decision view of an artifact can be understood from a different perspective as well: *Decision view represents the relationship(s) amongst the building blocks of an artifact*. It is a well-agreed that patterns are building blocks of architecture [8, 9, 19, 20] †. Hence, the decision view of architecture can be built using patterns. To build the decision view of patterns (discussed in section 5.1), we need to identify its building blocks. We propose tactics as building blocks for patterns. First, we analyze the relationship among architecture, patterns and tactics w.r.t. the following attributes – *Similarity*, *Granularity*, *Abstraction level*, *Quality attributes* and *Level of reusability*. Then, we check whether the relationship between architecture and patterns also holds true between patterns and tactics, to say that tactics can be used as building blocks of patterns. The relationships are described as follows:

- **Similarity**. One fundamental similarity among Architecture, Pattern and Tactic is that *at some level of abstraction all three of them can be considered as an effective design-solution for the given design-problem/requirement(s)*. One implication of this similarity is that the documentation mechanisms of one artifact are applicable for other artifacts as well. It can be seen that the patterns are documented based on various views like structural view, dynamic view, etc. (e.g. GoF [15], POSA1 [8]). Although tactics currently lack view-based documentation, these can be generated.

† *Tactics are also building blocks of architecture. For the sake of simplicity, we relax this fact here.*

- **Granularity**. The relationship among Architecture, Patterns and Tactics w.r.t. this attribute can be better understood using a *module view perspective*. The module view of architecture of a system can be realized as composition of module views of multiple patterns and tactics. The module view of a pattern can be realized as a composition of module views of multiple tactics. Thus, we can consider that architecture is composed of patterns and tactics and a pattern is composed of tactics.
- **Abstraction level**. Patterns and tactics form a *library of knowledge* which can generally be applied in several applications independent of the domain [34]. Many patterns and tactics provide architecture templates rather than concrete architecture fragments as their solution [9]; when domain-specific patterns are considered, their solution is close to the architecture fragments in that domain [14]. During architecture design phase, these templates are instantiated into concrete architecture fragments using application-specific details of the requirements. Patterns and tactics are categorized as part-of *application-generic knowledge* and the architecture is categorized as part-of *application-specific knowledge* [34].
- **Quality attributes**. Tactics achieve a primitive quality requirement of a quality attribute [4]. Patterns generally address requirements of multiple quality attributes [8, 9]. Since architecture is influenced by concerns of various stakeholders [4], architecture addresses multiple quality attributes.
- **Level of reusability**. In general, an artifact is selected for reuse whose properties match maximally with the required properties. When requirements of a system occurs in the context of an existing system e.g. product-line application, the initial design reuses existing architecture [5], here the level of reusability is at architecture level. The next level of reusability is at a pattern level [5], since pattern requirements have a finer grain than architecture and coarser than tactics.

Table 2 shows the relationship among the three artifacts. It can be noticed that the relationships between architecture and patterns are similar to those between patterns and tactics. We can say that tactics can be useful for describing patterns.

Table 2. Relationship among Architecture, Pattern and Tactic.

Attribute	Relationship	Interpretation
Similarity	Architecture ≈ Pattern ≈ Tactic	At some level of abstraction, they can be considered as *Problem-Solution* pair.
Granularity	Architecture > Pattern > Tactic	Architecture realizes into multiple patterns and a pattern realizes into multiple tactics.
Abstraction level	Architecture <= Pattern = Tactic	Patterns and Tactics are part-of application-generic knowledge, and architecture is part-of application-specific knowledge.
Quality Attributes	Architecture > Pattern > Tactic	A tactic generally addresses one QA, some patterns address multiple QAs and architecture addresses multiple QAs.
Level of reusability	Architecture > Pattern > Tactic	Reusability at architecture level is preferable than pattern level and reusability at pattern level is preferable than tactic level.

5 Quad-dimensional Knowledge model

Typically, before starting the design the application type/domain is identified and understood. Based on that, the designer identifies the most important quality attributes and the list of design pattern alternatives. Additionally, when choosing a pattern, the designer also has access to the tactics that compose that pattern and relationships between patterns. Following are some of the design queries of this form:

- What are the patterns that use *Rollback* tactic to recover faults in *Financial systems*? – *Memento* pattern
- What are the patterns that specialize *Proxy* pattern to improve *Scalability*? – *Mediator* pattern

To support the queries of above type, we designed a pattern-oriented knowledge model, composed of four dimensions (illustrated in Figure 3), that provide the designer with a view of how to best develop its application. VanHilst et al. [45] define the dimension on a knowledge model as follows: "*A dimension is a distinct list of concerns along a single axis, with a simple concept and a set of distinctions that define the categories*". We follow the same interpretation. Each dimension of the knowledge model is discussed in following subsections.

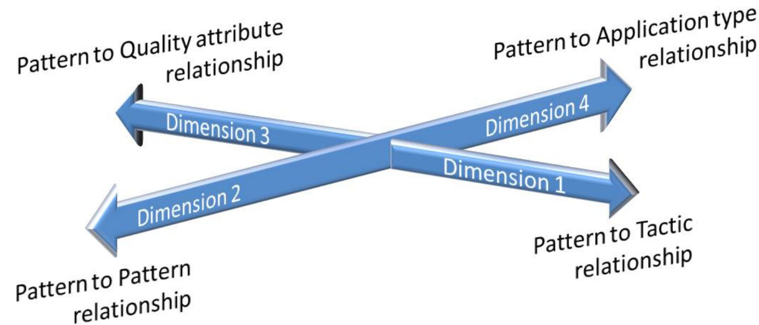

Figure 3: Four dimensions of our knowledge model.

5.1 Pattern to Tactic relationship

As discussed in section 4, the design decisions of a pattern can be captured using tactics which are more primitive solutions than patterns. Intuitively, if a pattern provides a solution to achieve multiple primitive quality requirements, a tactic provides a solution to achieve single primitive quality requirement [4]. For example, consider *Observer* pattern which provides solution to the following four quality requirements:

- *State change in one object requires state change in other objects*,
- *Dependents of an object are known at runtime*,
- *Abstract interface of variant modules is used for coupling* and
- *Variant modules need to be exchangeable at runtime*.

From a tactics perspective, *Observer* pattern is composition of *Notify modification*, *Register at runtime*, *Interface parameterization* and *Apply Polymorphism* tactics, since the above quality requirements are achieved by these four tactics respectively.

The constituent tactics of a pattern can be analyzed from the pattern description. In [30], we discuss the analysis procedure to analyze the tactics and Tactics Topology Model (decision view) of a pattern. In the Booch design process [7], fundamental design decisions are classified into five types: *Mechanism design decisions*, *Module design decisions*, *Service design decisions*, *Parameters design decisions*, *Association design decisions*. This classification can be used as a checklist while analyzing the constituent tactics of a pattern. Following our tactic analysis process, we analyzed the patterns in two popular pattern catalogues – *GoF* [15] and *POSA1* [8] and recovered the tactics of those patterns from their description. Due to space limitations, the decision views of all patterns cannot be presented here, for discussion purpose, we present the decision view of one pattern †. Figure 4 illustrates the decision view of *Observer* pattern.

† *The decision view analysis document for GoF and POSA1 patterns can be found at* http://www.cse.iitk.ac.in/users/vkirankr/Pattern_to_Tactics_Analysis.pdf.

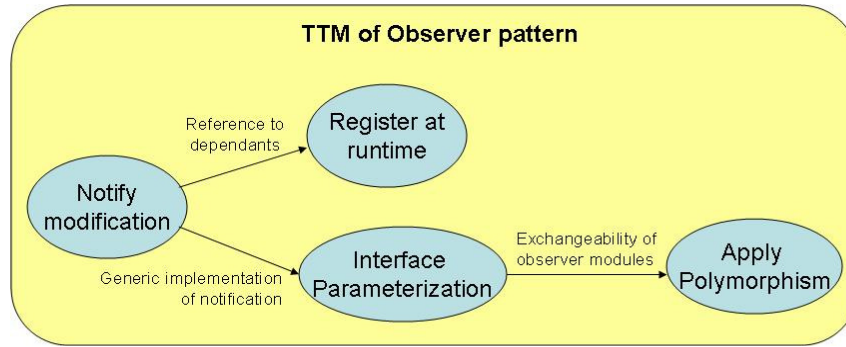

Figure 4. Tactic Topology Model of Observer pattern.

The interpretation of *Observer* pattern decision view (Figure 4) is as follows:

- The semantics of *Observer* pattern can be codified using four tactics: *Notify modification*, *Register at runtime*, *Interface parameterization* and *Apply Polymorphism*.
- *Notify modification* tactic is considered as primary tactic, since this tactic achieves the quality requirement closer to the context of *Observer* pattern.
- *Notify modification* tactic creates two quality requirements as its consequences: *Reference to dependants* and *Generic implementation of notification*. The tactics *Register at runtime* and *Interface parameterization* achieve these quality requirements respectively. Hence we consider the following two dependencies: Notify modification *constrains* Register at runtime and Notify modification *constrains* Interface parameterization. In decision view shown in Figure 4, *constrains* relationship is represented as an edge between these tactic nodes; the edge label represents the consequence or rationale for dependency.
- In a similar way, one consequence of *Interface parameterization* tactic is *Exchangeability of variant observer modules*. *Apply Polymorphism* tactic achieves this quality requirement, hence there is constrains relationship between *Interface parameterization* and *Apply Polymorphism* tactics.
- As discussed in section 4, a decision view supports safe-undo of a design decision during the systems evolution. It is to be noticed that when an *Observer* pattern is used, if the *Notify modification* tactic needs to be removed, other three dependant tactics also need to be removed, because their application context is dependant on *Notify modification* tactic.

Bass and colleagues define a catalogue of tactics [4] for various quality attributes. This catalogue seems insufficient to capture precisely the semantics of the considered patterns. Also, Bass et al. explicitly mention in [4] that "*the list of tactics is necessarily incomplete*". We defined an additional set of tactics to model precisely the tactic topologies for the considered patterns. Table 3 presents our tactics catalogue along with the quality requirements they achieve and their quality attributes. The quality requirements of the GoF and POSA1 patterns are mapped to the quality requirements of tactics (16 of Bass et al. and 20 additional given by us).

Table 3. Tactics capturing the Quality requirements of GoF and POSA1 patterns.

Bass et al. tactics		
Tactic	**Quality requirement**	**Quality Attribute**
Restrict communication paths	Hide a set of modules/services.	Modifiability
Union of services	Intuitive interface of complex object.	Usability
Aggregation of services		
Maintain multiple views	Handle multiplicity in user-interface requirements.	
Maintain hierarchy of views		
Apply polymorphism	Variant modules need to be exchangeable at runtime.	Substitutability
Interface parameterization	Abstraction based on variation points.	Reusability
Generalize service commonality		
Checkpoint	Persistence of consistent object state.	Availability
Rollback	Recovery of object state faults.	

Tactic	Quality requirement	Quality Attr.
Register at runtime	Dependents of an object are known at runtime.	Adaptability
Parameter based behavior selection	Support modification of the behavior at runtime.	
Multiple abstraction levels	High-level decomposition of an application.	Modularity
Multiple specialized modules		
Work partitioning	Efficient execution of computationally-intensive task.	Performance
Credentials based access	Authenticated access to the object.	Security
Added tactics		
Compose whole from parts	Design complex object from smaller parts.	Composability
Object cloning	Allow self object creation and initialization.	Performance
Object sharing	Reduce data duplication.	
Count number of references	Delete object when there are no references to it.	
Smart reference	Additional actions when an object is accessed.	Intelligence
Null value based object creation	Additional functionality to control object creation	
Object pool search based creation		
Heuristics based chaining	Nondeterministic selection of strategies	
Preprocessing module	Add new functionality without affecting existing object structure.	Extensibility
Add an individual module		
Subclass delegation		
Static access type	An object needs to be accessible from well-known access point.	Accessibility
Interface mapping	Overcome mismatch in interface signature.	Integrability
Chaining	Integration of independent modules.	
Container interface	Add/remove parts of composite at runtime.	Adaptability
Notify modification	State change in one object requires state change in other objects.	
Library operation	Common functionality encapsulation.	Reusability
Grammar	Represent statements in the language.	Usability
Traversal	Sequential access to the elements of aggregate object.	
Remote messaging	Support for distributed object communication.	Scalability

5.2 Pattern to Pattern relationship

In [30], we discussed how pattern relationships can be analyzed using graph properties when semantics of patterns are modeled using *Tactic Topology Model* (TTM). Table 4 presents the description and graph predicates for the five relationships used in our POKM. A brief discussion of the relationships is as follows:

- *Is-Similar-to* relationship is analyzed using *graph equivalence* property.
- *Is-an-Alternative-to* relationship is analyzed in two steps. The source node in a TTM resembles the context quality requirement of the pattern. Hence, to infer whether two patterns are addressing the same problem, one of the two following conditions need to be satisfied: *source nodes of the two patterns need to be same*, or *source node of one pattern is alternative of source node of other pattern*. When it is known that the two patterns are addressing the same problem, we need to check whether they propose different choices, this is inferred using *graph non-equivalence* property.
- *Uses* relationship is analyzed using *proper subgraph* property.
- *Refines* relationship is analyzed in two steps. Firstly, we need to ensure that both patterns provide same initial solution; this condition is formulated as *source nodes of the two patterns need to be same*. Secondly, we check whether a pattern extends the solution of other pattern using *proper subgraph* property.
- *Specializes* relationship is based on *graph homomorphism* property. First, we transform graph of a pattern using *generalization* (inverse of *special case*) relationship. Next, we check whether the graph of other pattern is *subgraph* of generalized graph.

Table 4. Description of pattern relationships in our POKM.

Relationship	Description / Graph predicate
Is-Similar-to	**Description**: Patterns A and B provide same solution to similar problem. [24]
	Graph predicate: Graph(P1) $\equiv$ Graph(P2).
Is-an-Alternative-to	**Description**: Patterns A and B solve the same problem, but propose different choices. [32, 50]
	Graph predicate: (Source-node(P1) = Source-node(P2) **OR** *is-alternative*(Source-node(P1), Source-node(P2))) **AND** Graph(P1) $\neq$ Graph(P2).
Uses	**Description**: When building a solution for the problem addressed by pattern A, one sub-problem is similar to the problem addressed by B. Therefore, the pattern A uses the pattern B in its solution. [32, 50]
	Graph predicate: Graph(P2) $\subset$ Graph(P1).
Refines	**Description**: Pattern A provides more wider/detailed solution than B. [32, 50]
	Graph predicate: Source-node(P1) = Source-node(P2) **AND** Graph(P2) $\subset$ Graph(P1).
Specializes	**Description**: The solution of pattern A indicates a special case of solution of pattern B. [32]
	Graph predicate: *generalization*: Graph(P1) $\rightarrow$ Generalized-graph(P1) **AND** Graph(P2) $\subseteq$ Generalized-graph(P1)

We applied the relationship predicates in Table 4 to analyze relationships among GoF and POSA1 patterns; Figure 5 illustrates the relationship analysis result (GoF and POSA1 patterns are denoted with different colors). Comparing our pattern relationship result with existing results, such as [50] and [2], we find that there is some amount of mismatch between the results. We figure out that following are some of the primary reasons:

- Since we recover tactics strictly based on essential sections of pattern description, our decision views can be considered as more restricted form. Sometimes we may not recover all the underlying tactics of a pattern because pattern description may not always provide all the details to implement the pattern.
- We believe that design experience also plays an important role during tactic recovery analysis. Experienced designers can analyze the given description from various perspectives with their design experience and perform tactic analysis at more fundamental level. Since we compare our result with highly experienced designers' result, some level of mismatch occurs.
- Also, their analysis details are unavailable or very brief. Hence, the scope of improving our result to their results remains limited.

In our analysis, we also found some unidentified relationships when compared with [50] and [2] such as:

- Mediator *is-similar-to* Client-Dispatcher-Server.
- Flyweight *is-alternative-to* Singleton.
- Interpreter *uses* Builder.
- Bridge *refines* Decorator.
- Blackboard *specializes* Pipes-and-Filters.

Some of the well-known pattern relationships are also identified such as:

- Layers *is-alternative-to* Pipes-and-Filters.
- Model-View-Controller *uses* Observer.
- Publisher-Subscriber *refines* Observer.
- Microkernel *specializes* Layers.

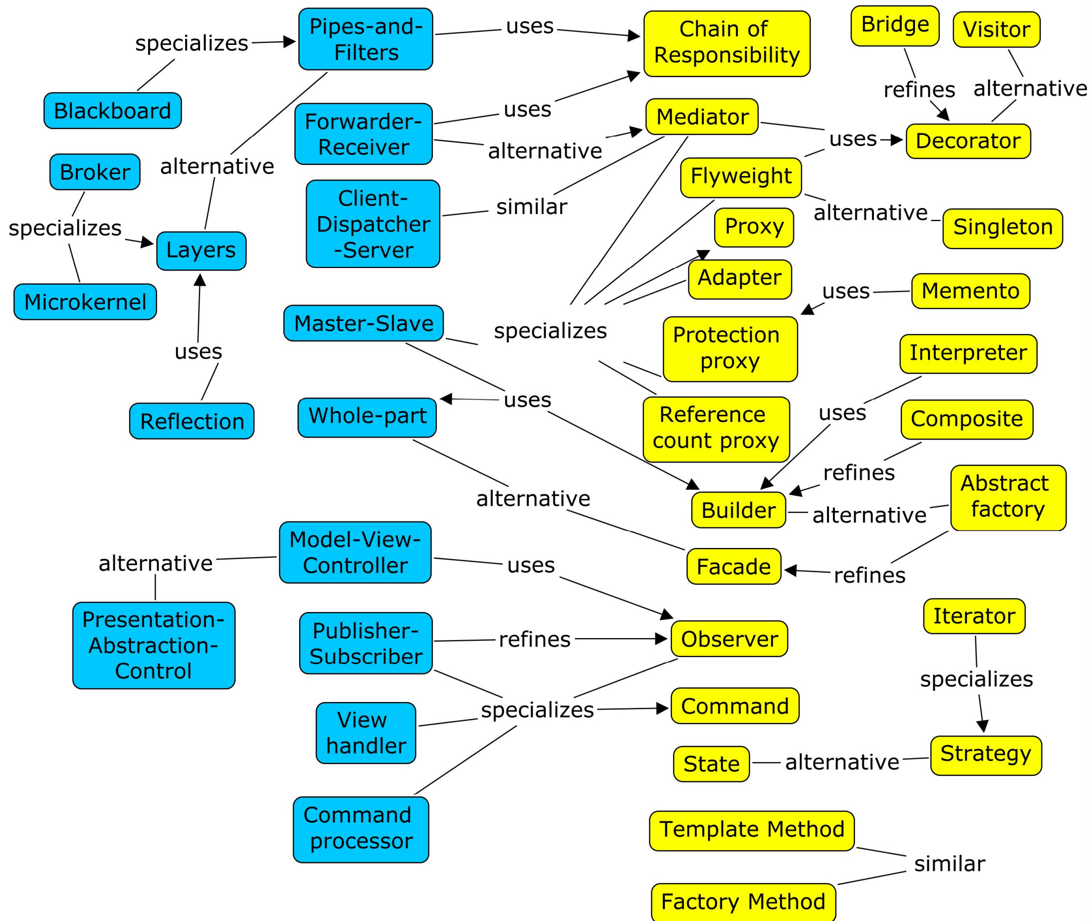

Figure 5. Pattern relationships for GoF and POSA1 patterns.

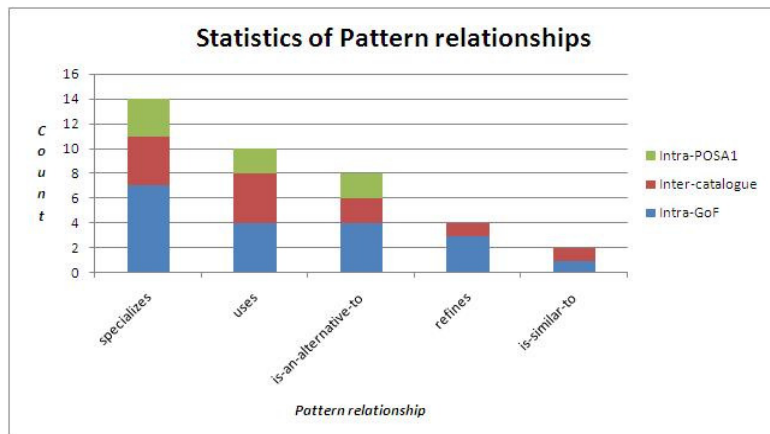

Figure 6. Statistics of pattern relationships.

Following are some of the conclusions that can be inferred from Figure 6:

- Majority of pattern relationships are captured by *specializes*, *uses* and *is-alternative-to*.
- The high frequency of *specializes* relationship shows that many patterns achieve different design problems with similar underlying architecture.
- Number(*Intra-GoF* relationships) > Number(*Inter-catalogue* relationships) > Number(*Intra-POSA1* relationships).

5.3 Pattern to Quality Attribute relationship

When patterns are represented as a constituent set of tactics, the quality attributes of a pattern can be analyzed through the quality attributes of tactics. Table 3 lists the quality attributes of tactics used in our analysis. One simple method to obtain quality attributes of a pattern is by the union of all the quality attributes of its tactics. For example, consider Observer pattern, when this method is applied we obtain three quality attributes such as: *Adaptability*, *Reusability* and *Exchangeability* tactics. This method does not explicitly represent following information:

- Observer pattern is a more appropriate alternative to improve *Adaptability* of the system rather than improve *Reusability* or *Exchangeability* of the system.
- With Observer pattern, the quality attributes *Reusability* or *Exchangeability* cannot be improved solely without improving *Adaptability* quality attribute.

In order to explicitly represent such information for a pattern, we add an additional level of refinement to the above method. Using the TTM of a pattern, we can easily classify the tactics of a pattern into two types (discussed in section 5.1): *Primary tactic* (root node tactic in TTM) and *Secondary tactics* (non-root node tactics in TTM). With this classification, we can also classify the quality attributes of a pattern into two types: *Primary quality attribute* (quality attribute of primary tactic) and *Secondary quality attributes* (quality attribute of subsequent tactics).

Reconsidering the *Observer* pattern, its quality attributes can now be classified as: *Adaptability* is primary quality attribute and *Reusability* and *Exchangeability* are secondary quality attributes. We applied TTM based quality attribute analysis to other GoF and POSA1 patterns to obtain this primary and secondary quality attributes for each of the patterns. Figure 7 presents the primary quality attributes of GoF and POSA1 patterns; in this figure, the patterns are grouped based on their primary quality attributes. Table A1 (in appendix) presents the secondary quality attributes of GoF and POSA1 patterns after normalization. The quality attributes used in our analysis can be referred from [1, 13, 36, 48].

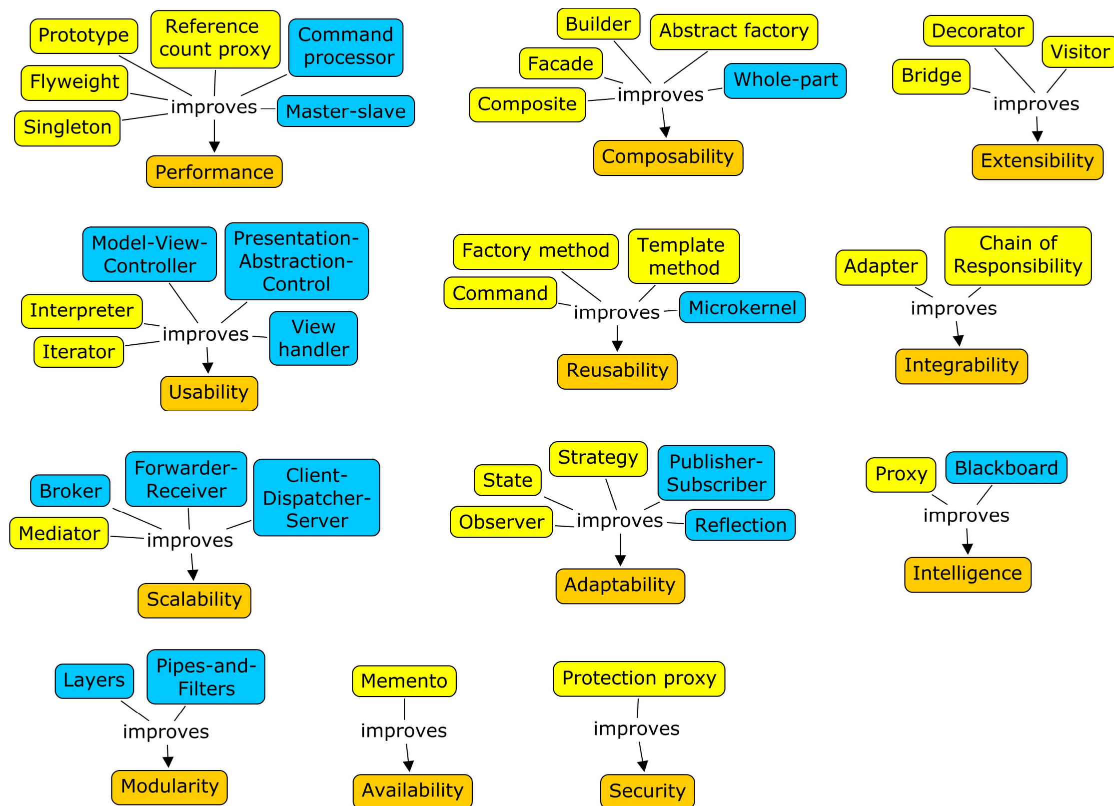

Figure 7. Primary quality attribute of GoF and POSA1 patterns.

Following are some of the conclusions that can be inferred from Figure 7:

- The quality requirements of *Performance*, *Composability* and *Usability* seem to be well-addressed, whereas quality requirements of *Availability* and *Security* are almost not addressed.
- Code-centric quality attributes like *Extensibility*, *Integrability* are especially addressed by GoF patterns; whereas organization-centric quality attributes like *Modularity* are especially addressed by POSA1 patterns.
- For quality attributes like *Performance*, *Composability* GoF patterns provide more alternatives than POSA1; whereas for quality attributes like *Scalability*, *Usability* POSA1 patterns provide more alternatives than GoF.

5.4 Pattern to Application Type relationship

The properties of an application can be characterized by the quality attributes it achieves; different applications focus on different quality attributes. For example, a *Product-line application* focuses on *Reusability* quality attribute, whereas *Gaming system* focuses on *Intelligence* and *Adaptability* quality attributes. Hence, we relate patterns to application-types using quality attributes they achieve.

Based on the primary quality attributes of the patterns, we selected six relevant application types such as: *Financial system*, *Operating system*, *Gaming system*, *Web service* and *Product-line application* from two application-type catalogues [6] and [40]. We then related the quality attributes to the appropriate application-type; Figure 8 illustrates our quality-attribute to application-type relationship analysis result. Combining the pattern to primary quality attribute knowledge given in Figure 6 (a) with Figure 8, we obtain the pattern to application-type relationship; Table A2 (in appendix) presents this analysis result after normalization. We concede that this taxonomy of application types may not be comprehensive, but this list illustrates a dimension that is important for a *Design Assistant*.

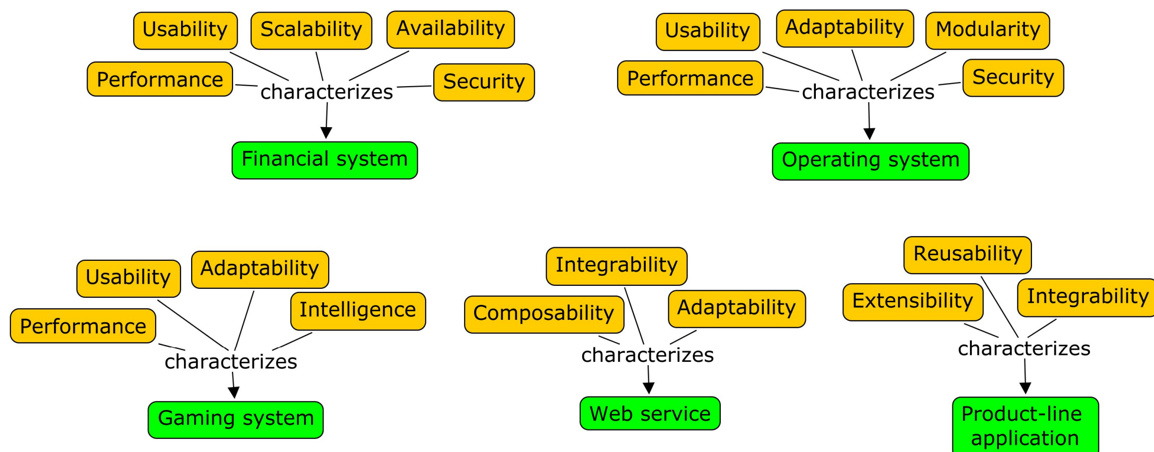

Figure 8. Pattern to Application Type relationship for GoF and POSA1 patterns.

5.5 Usefulness of our knowledge model

The usefulness of the knowledge base is to generate available pattern alternatives for a given design context. Table 6 presents a set of thirteen design queries and the pattern alternatives for each query. The pattern alternatives are generated based on our analysis results (subsections 5.1 to 5.4.) on GoF and POSA1 patterns.

Table 6. Some design queries and their pattern alternatives of our knowledge model.

	Design query	**Pattern alternatives**
1	What are the patterns that *use Notify-modification* tactic? (refer section 5.1)	Observer, Model-View-Controller, Publisher-Subscriber
2	What are the patterns that are *built-using Compose whole from parts* tactic? (refer section 5.1)	Composite , Whole-part, Façade, Abstract factory, Builder
3	What are the patterns *similar-to Mediator* pattern? (refer section 5.2)	Client-Dispatcher-Server
4	What are the patterns *alternative-to Layers* pattern? (refer section 5.2)	Pipes-and-Filters
5	What are the patterns that *use Layers* pattern? (refer section 5.2)	Reflection
6	What are the patterns that *refine Observer* pattern? (refer section 5.2)	Publisher-Subscriber
7	What are the patterns that *specialize Layers* pattern? (refer section 5.2)	Broker, Microkernel
8	What are the patterns that *improve Performance* quality attribute? (refer section 5.3)	Master-Slave, Singleton, Flyweight, Prototype, Reference count proxy, Command processor.
9	What are the patterns that *primarily improve Performance* and *improve Reusability* as well? (refer section 5.3)	Master-Slave, Flyweight, Prototype.
10	What are the patterns that *apply-for Web service* application? (refer section 5.4)	Reflection, Observer, Publisher-Subscriber, Composite , Whole-part, Façade, Abstract factory, Builder, Adapter, Chain of Responsibility
11	What are the patterns that *apply-for Adaptability* aspects of *Web service* application? (refer section 5.4)	Reflection, Observer, Publisher-Subscriber
12	What are the patterns which *use Object-sharing* tactic to improve *Performance* quality attribute? (refer section 5.1 and 5.3)	Singleton, Flyweight
13	What are the patterns that *specialize Layers* pattern for *Product-line* application? (refer sections 5.2 and 5.4)	Microkernel

6 Related work

Wu and Kelly [47] extend the Bass et al. tactic set [4] by adding 17 tactics for Safety quality attribute; these tactics are classified into three categories – *Failure Avoidance*, *Failure Detection* and *Failure containment*. They also propose a template similar to pattern description template to describe a tactic in a structured way.

Harrison and Avgeriou [21] discuss that many general patterns cannot be directly applied when designing reliable systems because these patterns do not address the fault-tolerance issues in their solutions; in this case, the pattern solution needs to be further refined to incorporate fault-tolerance tactics. They discuss how the existing pattern solutions can be transformed to incorporate fault-tolerance tactics; they also analyzed the difficulty levels to implement tactics into patterns.

Khomh et al. [54, 55] performed an empirical analysis relating GoF patterns to the following three quality attributes: *Expandability*, *Understandability* and *Reusability*. They interviewed 20 designers regarding the impact of patterns on the above three quality attributes; the impact is assessed in three levels: *positive*, *neutral*, and *negative*. We analyzed the GoF patterns using a richer set of quality attribute which includes other quality attributes such as *Performance*, *Adaptability* etc. We analyzed the quality attributes of a pattern using the constituent tactics of a pattern, some of the tactics in a pattern neutralize the side-effect caused by other tactic which can also be considered as positive impact. The mapping between Khomh et al. quality attributes and our quality attributes is as follows: *Expandability* maps to *Extensibility* and *Composability* quality attributes, *Understandability* maps to *Modifiability* and *Substitutability* quality attributes, *Reusability* maps to *Reusability* and *Integratability* quality attributes. Comparing our result with their empirical result, we find a strong correlation between our conclusions and theirs: 78% (54/69).

Zhao and colleagues [49] discuss 19 dimensions of software engineering knowledge such as: *process knowledge*, *pattern knowledge*, *version knowledge*, *technology knowledge* etc. Their definition of *pattern knowledge* is similar to one of the dimension in our knowledge model - Pattern to Pattern relationships. Our knowledge model differs from theirs as follows: in their knowledge model, each dimension is independent of other dimensions; whereas in our knowledge model, each dimension is related to patterns. For example, consider *Application domain knowledge*, their definition for this knowledge includes codifying the business requirements; whereas in our knowledge model, this knowledge includes codifying various pattern alternatives that are applicable for an application domain.

Santonu and Kunal [42] discuss an industrial application of design knowledge models like ours and others like [6], [45], [50], [43] etc. They discuss that in reality, to optimize budgets, the project team often involves inexperienced designers; in this case, the inexperienced designers can be assisted with knowledge-based design assistant tools during architecture design. They also explain the *producer-consumer* relationship between design experts and inexperienced designers – design experts codify the design knowledge with appropriate knowledge models and the *Design assistant* tool supports the inexperienced designers during architecture design phase.

The applications of ontologies in various software engineering activities are discussed in [18] and [16]. Tom Gruber [17] defines Ontology as *representation of domain as set of concepts and relationships between those concepts*. Under this definition, our knowledge model can also be considered as ontology-based knowledge model. Also, our knowledge model satisfies the concept-instance property of the ontology; for example, in our knowledge model, *Quality Attribute* is a concept and *Reusability*, *Performance* etc are its instances.

Linguistics based pattern properties analysis is also under research: Hironori and colleagues [33, 46] propose Natural Language Processing (NLP) based methodology for pattern relationship analysis from pattern description; Hasso and Carlson [22] classify patterns using NLP. We use decision view as underlying model for our analysis. One direct benefit of NLP based methodology over our methodology is automated solution for pattern analysis. We believe that the state-of-the-art of NLP technology is insufficient for rigorous pattern analysis. Since NLP is an active research area, as the maturity of NLP technology increases, the maturity of NLP-based techniques also increase accordingly.

In recent years, in addition to existing pattern description, compact high-level representations of a pattern are also gaining interest. Hseuh et al. [27] propose a 6-tuple representation to specify the essence of the pattern description: <*Functional Requirement* (FR), *Nonfunctional requirement* (NFR), *Impact on different quality metrics*, *Structure realizing FR* (S-FR), *Structure realizing NFR* (S-NFR), *Transformation function from S-FR to S-NFR*>. Buschmann et al. [9] propose a rule-based (if-then) representation; the rule-based representation offers a clear separation between problem and solution parts of the pattern. In this representation, the problem and solution of a pattern are decomposed into multiple simpler parts; the problem part is represented as a Boolean formula and solution as a sequence of steps. Our Tactic Topology Model (decision view) can also be considered as a compact representation of a pattern, because the fewer constituent tactics denote what the pattern is and what it is not. In other words, constituent tactics denote what quality requirements are addressed and what quality requirements are not addressed in a pattern.

ArchE [3, 10] is a research prototype design assistant developed by Bachmann and colleagues at SEI. Currently, this tool is based on reasoning frameworks or mathematical models of the quality attributes. For example, *Rate Monotonic Analysis*, *Queuing Theory* etc are reasoning frameworks for *Performance* quality attribute. One constraint in using reasoning framework is that designer needs to specify (accurately) the current state of architecture and required state of architecture using a set of quality attribute parameters/metrics. At initial stages of architecture design, this information may not be available or hard to

analyze these values. In such cases, designer needs assistance for quality requirements in abstract form. Our knowledge model is suitable in this case. An orthogonal dimension of knowledge can be added to ArchE using ontology-based knowledge models.

Architecture knowledge management is classified into types: *application-generic knowledge management* and *application-specific knowledge management* [34]. Our knowledge dimensions fall under the application-generic knowledge management category. Application-specific knowledge management involves managing the knowledge of a specific application during the initial development or evolution of that application. This involves managing design decisions such as: *Structural design decisions*, *Deployment design decisions*, *Integration design decisions*, *Presentation design decisions*, *Technology selection design decisions* etc [44]. In addition to managing design decisions, other views of the application such as *Logical view*, *Process view* etc also needs to be maintained. Rambabu and Prabhakar [11] discuss different attributes to annotate architectures. This annotation helps the architect in searching efficiently the previous architectures to find an architecture suitable for reuse or to find the design decisions used to resolve similar design problems.

The competency level of an ontology is evaluated using competency questions - these are a set of queries the ontology can be able to answer [29, 37]. Combining our knowledge model with other different knowledge models like [6], [45], [50], [43] etc can improve the competency level of a design assistant so that many of the common design queries can be answered. Lee and Kruchten provide efficient visualization support for browsing the ontology based knowledge models [35].

7 Conclusions

Software design patterns document the most recommended solutions to recurring design problems. Since the design decision at a particular design context is bound to one of the analyzed set of alternatives, missing an important alternative can sometimes impact the selected decision. Analysis of alternative patterns for a given set of requirements is a knowledge-intensive task; pattern knowledge overload hardens the alternative analysis. Providing a knowledge base to analyze pattern alternatives can alleviate this problem to a greater extent. When architecture design knowledge is codified appropriately, design alternative analysis problem can be modeled as an information retrieval problem. We used the classic concepts-and-relationships model to codify knowledge of patterns in four different dimensions - *Pattern to Tactic relationship*, *Pattern to Pattern relationship*, *Pattern to Quality-attribute relationship* and *Pattern to Application-type relationship*.

When compared to others, one basic difference is that in our knowledge model we analyze patterns from the decision view perspective. Different formal approaches based on mathematical structures exist to describe a pattern formally; we focus on intuitive graph models for pattern description. We discussed the usefulness of our knowledge model with various design queries along with their pattern alternatives. Our contributions for the GoF and POSA1 patterns knowledge can be summarized as follows:

- we analyzed the decision views for each of these patterns,
- we analyzed five types of relationships among these patterns by applying different graph rules on decision views of these patterns,
- we analyzed the primary and secondary quality attributes for each these patterns based on their decision views and
- for different application types, we analyzed the set of patterns applicable for them based on the primary quality attributes of the patterns.

Our analysis results can be used in at least two ways: to build a competent knowledge base to assist the designer during analysis phase and to train the novice designers. Since the competency level of a knowledge base is evaluated using different competency questions, combining our knowledge model with other existing knowledge models can improve the competency level of a knowledge base.

With this analysis as experience, we intend to broaden our future research in two directions: adding additional dimensions to our knowledge model and extending the analysis of this knowledge model to other POSA patterns, Enterprise patterns, Berkeley OPL etc.

8 Acknowledgements

We are very grateful to our shepherd Pedro Monteiro who had tirelessly read and re-read many versions of the paper and improved both the form and content. The comments and suggestions given by our shepherd have been very effective and helped a lot in improving the quality of our paper. We also thank our program committee member Rosana Teresinha Vaccare Braga. The shepherding process is indeed a very useful practice.

9 References

[1] Ahmad Kayed, Nael Hirzalla, Ahmad A. Samhan, Mohammed Alfayoumi, "*Towards an Ontology for Software Product Quality Attributes*," iciw, pp.200-204, 2009 Fourth International Conference on Internet and Web Applications and Services, 2009.

[2] P. Avgeriou and U. Zdun, “*Architectural patterns revisited - a pattern language*”, In Proceedings of 10th European Conference on Pattern Languages of Programs 2005.

[3] F. Bachmann, L. Bass, & M. Klein. “*Preliminary Design of ArchE: A Software Architecture Design Assistant*”. (CMU/SEI-2003-TR-021, ADA421618). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[4] L. Bass, P. Clements and R.Kazman, “*Software Architecture in Practice*”, Second Edition. Addison-Wesley 2003.

[5] L. Bass, “*Generate and Test as a Software Architecture Design Approach*”. 8th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2009.

[6] G. Booch: On Design, *Handbook of Software Architecture* blog, March 2006, http://www.booch.com/architecture/blog.jsp?archive=2006-03.html

[7] G Booch, R Maksimchuk, M Engle, B Young, J Conallen and K Houston, “*Object-oriented analysis and design with applications*”. Third Edition, Addison-Wesley 2007.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, “*Pattern-Oriented System Architecture: A System of Patterns*”, John Wiley & Sons, 1996.

[9] F. Buschmann, K. Henney, D. C. Schmidt, “*Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*”. Wiley & Sons, 2007.

[10] A. Diaz-Pace, H. Kim, L. Bass, P. Bianco and F. Bachmann. “*Integrating Quality-attribute Reasoning Frameworks in the ArchE Design Assistant*”. In 4th International Conference on the Quality of Software Architecture (QoSA), University of Karlsruhe (TH), Germany, 2008.

[11] Duddukuri R and Prabhakar T.V. “*On archiving architecture documents*”. In Software Engineering Conference, 2005. APSEC ’05. 12th Asia-Pacific, pages 351 – 358, December 2005.

[12] Dueñas, J.C. and Capilla, R, “*The Decision View of Software Architecture*”, Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), Springer-Verlag, LNCS 3047, pp. 222-230 (2005).

[13] G Elias and Rashmi Jain, “*Exploring Attributes for Systems Architecture Evaluation*”, Proceedings of the Conference on Systems Engineering Research, 2007.

[14] M. Fowler, “*Analysis patterns: reusable object models*”, Addison-Wesley, 1997.

[15] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “D*esign Patterns: Elements of Reusable Object-Oriented Software*”, Addison-Wesley, 1994.

[16] D Gaševic, N Kaviani and M Milanovic, “*Ontologies and Software Engineering*”. Handbook on Ontologies, pages 593-615, 2009.

[17] T. Gruber, *What-is-an-ontology*?, http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.

[18] H.J. Happel, S. Seedorf, "*Applications of Ontologies in Software Engineering*", In Proc. of International Workshop on Semantic Web Enabled Software Engineering, 2006.

[19] N. B. Harrison, P. Avgeriou and U. Zdun. "*Using patterns to capture architectural decisions*". IEEE Software, 24(4): pages 38–45, 2007.

[20] N. Harrison and P. Avgeriou, "*Leveraging Architecture Patterns to Satisfy Quality Attributes*". First European Conference on Software Architecture Madrid, Spain, September 24-26, 2007, Springer Lecture Notes in Computer Science.

[21] N. Harrison and P. Avgeriou, "*Incorporating Fault Tolerance Tactics in Software Architecture Patterns*", RISE/EFTS Joint International Workshop on Software Engineering for REsilieNt systems, November 17-19, 2008, Newcastle upon Tyne (UK), ACM CS Press.

[22] S. Hasso and C.R. Carlson, "*Linguistics-based Software Design Patterns Classification*", In Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Science (HICSS-37). IEEE Computer Society Press, 2004.

[23] S. Henninger and P. Ashokkumar, "*An Ontology-Based Metamodel for Software Patterns*". Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering, USA, 2006.

[24] S. Henninger, V. Corrêa, "*Software Pattern Communities: Current Practices and Challenges*", Pattern Languages of Programs (PLoP 07), (submitted), 2007.

[25] M. Hepp, "*Ontologies: State of the Art, Business Potential and Grand Challenges*". Ontology Management, pages 3-22, 2008.

[26] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran and P. America, "*Generalizing a model of software architecture design from five industrial approaches*", 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5), Pittsburgh, Pennsylvania, 2005.

[27] Hsueh, N.L., Chu, P.H., Chu, W.: "*A Quantitative Approach for Evaluating the Quality of Design Patterns*". J. Systems and Software (2008).

[28] A Jansen and J Bosch, "*Software Architecture as a Set of Architectural Design Decisions*". 5th Working IEEE/IFIP Conference on Software Architecture, pages 109-120, 2005.

[29] H Kampffmeyer and S Zschaler, "*Finding the Pattern You Need: The Design Pattern Intent Ontology*". Proceedings of Model Driven Engineering Languages and Systems, pages 211-225, 2007.

[30] Kiran Kumar and Prabhakar TV, "*Design Decision Topology Model for Pattern Relationship Analysis*", Asian Conference on Pattern Languages of Programs, 2010.

[31] G. Kotonya, I. and Sommerville. "*Requirements Engineering: Processes and Techniques*". John Wiley & Sons, 1998.

[32] P. Kruchten, "*An ontology of architectural design decisions in software intensive systems*", In 2nd Groningen Workshop on Software Variability, pages 54-61, December 2004.

[33]A. Kubo, H. Washizaki and Y. Fukazawa, "*Extracting relations among security patterns*," in Proceedings of the Second Workshop on Software Patterns and Quality, 2007.

[34] Lago P., Avgeriou P, 1st Workshop on *SHaring and Reusing ARchitectural Knowledge* (FinalWorkshop Report). ACM SIGSOFT Software Engineering Notes 31(5), 32–36, (2006).

[35] L. Lee and P. Kruchten, "*A Tool to Visualize Architectural Design Decisions*". Quality of Software Architectures, pages 43-54, 2008.

[36] Liam O'Brien, Paulo Merson, Len Bass, "*Quality Attributes for Service-Oriented Architectures*", sdsoa, pp.3, International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007), 2007.

[37] Noy, N.F., McGuinness, D.L.: "*Ontology development 101: A guide to creating your first ontology*". Technical Report KSL-01-05, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 94305, USA (March 2001).

[38] A Ran and J Kuusela, "*Design Decision Trees*". 8th International Workshop on Software Specification and Design, pages 172-175, 1996.

[39] Remco C. de Boer, R Farenhorst, P Lago, Hans van Vliet, Viktor C and A Jansen, "*Architectural Knowledge: Getting to the Core*", proceedings of the 3rd International Conference on the Quality of Software Architectures (QoSA 2007), Springer LNCS 4880, pp. 197-214, 2008.

[40] Robert L. Glass, Iris Vessey, "*Contemporary Application-Domain Taxonomies*," IEEE Software, vol. 12, no. 4, pp. 63-76, July 1995.

[41] JM Rosengard and MF Ursu, "*Ontological representations of software patterns*". Proceedings of Knowledge-Based Intelligent Information and Engineering Systems, pages 31-37, 2004.

[42] Santonu Sarkar and Kunal Verma, "A*ccelerating technical design of business applications: a knowledge-based approach*". 3rd India software engineering conference, pages 43-50, 2010.

[43] Tichy, W.F., "*A catalogue of general-purpose software design patterns*". In Proceedings of the Technology of Object-Oriented Languages and Systems, IEEE Computer Society, 1997.

[44] Tyree J. and Akerman A, "*Architecture Decisions: Demystifying Architecture*", IEEE Software, vol. 22, pages 19-27, 2005.

[45] VanHilst Michael, Fernandez Eduardo B and Braz Fabricio, "*A Multi-dimensional Classification for Users of Security Patterns*", Journal of Research and Practice in Information Technology, pages 87-97, 2009.

[46] H. Washizaki, A. Kubo, A. Takasu and Y. Fukazawa: "*Relation Analysis among Patterns on Software Development Process*", Proc. 6th International Conference on Product Focused Software Process Improvement, LNCS Vol.3547, 2005.

[47] Wu, W., Kelly, T.: "*Safety Tactics for Software Architecture Design*". In: Proceedings of the 28th International Computer Software and Applications Conference, IEEE Computer Society, Los Alamitos (2004).

[48] Zain Balfagih, Mohd Fadzil Hassan, "*Quality Model for Web Services from Multi-stakeholders' Perspective*," icime, pp.287-291, 2009 International Conference on Information Management and Engineering, 2009

[49] Y. Zhao, J. Dong and T. Peng, "*Ontology Classification for Semantic-Web-Based Software Engineering*". IEEE Transactions on Services Computing, pages 303-317, 2009.

[50] W Zimmer, "*Relationships Between Design Patterns*", J. Coplien and D. Schmidt, editors, Pattern Languages of Program Design, pages 345-364, 1995.

[51] http://cmap.ihmc.us/ accessed on February 2010

[52] http://protege.stanford.edu/ accessed on October 2009

[53] http://vue.tufts.edu/ accessed on May 2010

[54] F Khomh, Y-G Gueheneuc, "Perception and Reality: What are Design Patterns Good For?" Proceedings of the 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), July 31st 2007, Berlin, Germany. Springer-Verlag.

[55] Foutse Khomh, Yann-Gael Gueheneuc, Do Design Patterns Impact Software Quality Positively? Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), du 1-4 avril 2008, Athènes, Grèce. IEEE Computer Society Press.

10 Appendix

Table A1. Secondary quality attributes of GoF and POSA1 patterns.

Patterns	Substitutability	Reusability	Modifiability	Extensibility	Usability	Adaptability	Modularity	Composability	Integrability	Security	Accessibility
Flyweight	✓	✓	✓	✓							
Master-slave	✓	✓	✓		✓			✓			
Model-View-Controller	✓	✓	✓			✓	✓				
Pipes and Filters	✓	✓	✓						✓		
Bridge	✓	✓	✓								
Chain of Responsibility	✓	✓	✓								
Blackboard	✓	✓	✓								
Mediator	✓	✓		✓		✓					
Client-Dispatcher-Server	✓	✓		✓		✓					
Publisher-Subscriber	✓	✓		✓							
Composite	✓	✓			✓	✓					
Builder	✓	✓			✓						
Visitor	✓	✓			✓						
Forwarder-Receiver	✓	✓				✓			✓		
Interpreter	✓	✓						✓			
Prototype	✓	✓									
Decorator	✓	✓									
Iterator	✓	✓									
Observer	✓	✓									
State	✓	✓									
Strategy	✓	✓									
Abstract factory	✓		✓		✓						
Reflection	✓		✓			✓					
Layers	✓		✓								
Microkernel	✓		✓								
Broker	✓					✓					
Factory method	✓										
Command	✓										
Template method	✓										
Command processor	✓										
View Handler	✓										
Adapter			✓	✓							
Proxy			✓	✓							
Protection proxy			✓	✓							
Reference count proxy			✓	✓							
Façade			✓		✓						
Whole-part			✓		✓						
Presentation-Abstraction-Control			✓				✓				
Singleton			✓		✓						✓
Memento				✓						✓	

Table A2. Different application types and their suitable GoF and POSA1 patterns.

Pattern	Financial system	Operating System	Gaming system	Web service	Product-line application
Prototype	✓	✓	✓		
Singleton	✓	✓	✓		
Flyweight	✓	✓	✓		
Reference count proxy	✓	✓	✓		
Master-slave	✓	✓	✓		
Command processor	✓	✓	✓		
Interpreter	✓	✓	✓		
Iterator	✓	✓	✓		
Model-View-Controller	✓	✓	✓		
Presentation-Abstraction-Control	✓	✓	✓		
View Handler	✓	✓	✓		
Protection proxy	✓	✓			
Mediator	✓				
Memento	✓				
Broker	✓				
Forwarder-Receiver	✓				
Client-Dispatcher-Server	✓				
Observer		✓	✓	✓	
State		✓	✓	✓	
Strategy		✓	✓	✓	
Reflection		✓	✓	✓	
Publisher-Subscriber		✓	✓	✓	
Layers		✓			
Pipes and Filters		✓			
Proxy			✓		
Blackboard			✓		
Adapter				✓	✓
Chain of Responsibility				✓	✓
Abstract factory				✓	
Builder				✓	
Composite				✓	
Façade				✓	
Whole-part				✓	
Factory method					✓
Bridge					✓
Decorator					✓
Command					✓
Template method					✓
Visitor					✓
Microkernel					✓