Design Decision Topology Model for Pattern Relationship Analysis

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 are solutions to recurring design problems. Analyzing and managing the large and ever increasing number of design patterns is a problem. Non-uniform and incomplete pattern descriptions further complicate the task.

Existing literature defines different pattern relationship types and many relationships among patterns. These relationships are analyzed based on designer's experience and their formal basis is unclear. We propose a novel graph based model to capture the semantics of a design pattern using design decisions and their side-effects. The relationships are analyzed using various graph properties which enable automation of relationship analysis.

1. Introduction

A design pattern describes a particular recurring design problem that arises in a specific design context, and presents a well-proven generic scheme for its solution [9, 5]. Patterns are increasingly being used not only to capture and disseminate best practices, but also to turn named patterns into a shared vocabulary for expressing and communicating technical knowledge [9, 5, and 13]. The large number of existing and continuously increasing patterns (one source states that there are 250 patterns for Human-Computer interaction alone [19]) introduce new problems to designer who use them - like the management of a pattern knowledge base.

We propose a graph based model called Design Decision Topology Model (DDTM) to deal with the relationship analysis problem. The objective of this model is to reduce pattern *semantics* to *syntax*- a graph which delivers the pattern functionality (quality) through elementary functionality (quality) – nodes of the graph are elementary functional functionality and edges are dependencies. Conceptually, the *DDTM* technique is analogous to the *Decision view* [8, 16, 25] in the architecture domain. The utility of the *DDTM* for a pattern can be derived from the utility of Decision view for architecture. Researchers of architecture domain [8, 16, 25] propose *Decision views* to enable:

- Enriching architecture description
- Codifying crosscutting and intertwined design decisions present in multiple views.
- Traceability of quality requirements.
- Providing thumbnail or compact forms of the architecture.

Traceability and *Thumbnail* problems are considered important at pattern level also [6]. We apply architecture level techniques at pattern level to derive the DDTM of a pattern. This representation enriches pattern descriptions, helps analyze quality requirement traceability and relationships amongst patterns.

This model treats each pattern as a micro-architecture and defines the pattern as a topology of a set of design decisions. Using this model, different relationships are analyzed using graph properties. For example,

Patterns A and B are duplicates if Graph(A) ≡ graph(B),

Pattern A comprises-of patterns B and C if Graph(B) ⊂ Graph(A) AND Graph(C) ⊂ Graph(A).

The rest of the paper is structured as follows: Section 2 provides the required background terminology. In section 3, we discuss briefly how a pattern is described as a DDTM. In section 4, we demonstrate the tactic topology model which is a kind of DDTM. Section 5 discusses related work, and section 6 concludes the paper suggesting some future directions.

2. Terminology

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

- **Quality requirement** [27]: is a requirement which is not specifically concerned with the functionality

of the software. Quality requirements specify the external constraints the software should meet.

- **Quality Attribute** [2]: is a set of related quality requirements.
- **Design Decision** [15]: is a strategy that is applied to solve a particular part of the problem.
- **Tactic** [2]: A tactic is a design decision that influences the control of a quality attribute parameter. For example, the *Increase available resources* design decision (upgrading 512 MB RAM to 1 GB RAM) controls (minimizes) the response time parameter.
- **Implications/Side-effect** [3, 25]**:** 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, the *Increase available resources* tactic which is an alternative to achieve *Reduce response time* quality requirement imposes side-effects like *Increase in cost*, *Change in resource management (scheduling) policy* etc.

	Relationship Type	Synonyms	Description
1	*Is-Duplicate-of*	--	Patterns A and B provide same solution to same problem. [14]
2	*Is-an-Alternative-to*	*Similar-to*	A and B are similar patterns, solving the same problem, but proposing different choices. [26, 17]
3	*Comprises*	• Uses • Is-made-of • Decomposes -into	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. [26, 21, 17]
4	*Refines*	• Is-variant-of • Subsumes	Patterns A and B address same problem but pattern A provides more refined (with less side-effects) solution than B. [26, 17]

Table 1: Description of different Relationship types.

3. How to describe a pattern – the Design Decision Topology Model (DDTM)

Analyzing the relationships for a given set of patterns can be considered as 3-step process:

- Analyze design decisions of the patterns.
- Analyze the topology of the design decisions.
- Analyze the relationships from the topologies.

3.1. Analyze design decisions of the patterns.

The key-information of a pattern is usually embedded in the essential sections of the pattern-form/template; *Context*, *Problem*, *Solution, Consequences* sections are considered to be essential sections [9, 5, 6]. Additional sections such as *Implementation*, *Example* etc often appropriate to provide meaningful guidance on where a pattern applies and how to apply it [6]. Hence we use the key-information provided in *Problem*, *Solution, Consequences* sections as clues to analyze design decisions.

3.2. Analyze topology of design decisions.

DDTM provides a structure to the design decisions of a pattern by explicitly representing the dependency among them. DDTM primarily provides rationale for existence of a particular design decision. This information is modeled as edges among design decisions in our graph model. An edge A → B in DDTM means the side-effect of design decision A is resolved by design decision B.

The DDTM of a pattern can be viewed as a graph of design decisions. The *Intent* section specifies the primary design decision(s) of the pattern; they are modeled as source-nodes in the DDTM. Based on the implications of the current stage design decisions, the design decisions in the next stage are analyzed. For example, consider the *Master-slave* pattern [5]. The intent of *Master-slave* pattern is given as: *the master component distributes the work to slave components to support parallel computation*. It specifies *Work partitioning* as the primary design decision; and it is modeled as the source node in the DDTM of *Master-slave* pattern. One of the implications of *Work partitioning* design decision is the work distribution details need to be hidden from clients; hence *Restrict communication paths* design decision is used to overcome this implication. The continuation of the design process leads to an edge from *Work partitioning* to *Restrict communication paths* nodes in the DDTM of *Master-slave* pattern; the label on the edge represents the implication of *Work partitioning* design decision. Figure 1 shows a fragment of DDTM of *Master-slave* pattern.

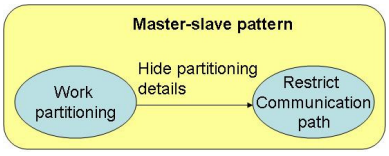

Figure 1: DDTM of Master-slave pattern.

	Relationship Type	Description
1	*Is-Duplicate-of*	Graph(P1) ≡ Graph(P2). (Graph equivalence property)
2	*Is-an-Alternative-to*	Source-Node(P1) = Source-Node(P2) AND Graph(P1) ≠ Graph(P2).
3	*Comprises*	Graph(P2) ⊂ Graph(P1). (Sub-graph property)
4	*Refines*	Source-Node(P1) = Source-Node(P2) AND Graph(P2) ⊂ Graph(P1).

Table 2: Graph rules of different Relationship types.

3.3. Analyze relationships from the topologies.

Various relationships among patterns can be analyzed using some graph properties, when patterns are described with DDTM. The graph model we propose is currently applicable to analyze four relationship types (see Table 1):
(1) Is-Duplicate-of,
(2) Is-an-Alternative-to,
(3) Comprises, and
(4) Refines.

Table 2 describes how these relationship-types map into properties of graphs.

4. Evaluation

This section demonstrates the utility of DDTM. We define *Tactic Topology Model (TTM)*, an instance of DDTM where the design decisions are tactics. Using TTM we describe some patterns such as *Abstract factory*, *Builder*, *MVC*, *Observer*, and *Publisher-Subscriber* to analyze relationships between them.

4.1. Tactics as design decisions of a pattern – Tactic Topology Model (TTM).

Conceptually, there can be a variety of design decisions that can model the semantics of patterns. Choosing tactics as design decisions has these benefits:

- DDTMs can be communicated easily when its design decisions refer to a standard body of knowledge such as tactics.
- Tactics are classified according to different quality attributes; this allows easy indexing of a tactic when its quality requirement is known.

Tactic Topology Model (TTM) is a kind of DDTM where the design decisions of a pattern are captured through tactics which are more elementary than patterns. Intuitively, if a pattern provides a solution to achieve multiple primitive quality requirements, a tactic provides a solution to achieve a single primitive quality requirement [2].

Bass et al define a catalogue of tactics [2, 4] for various quality attributes. This catalogue seems insufficient to precisely capture the semantics of the considered patterns. Also, Bass et al explicitly mention in [2] that "*the list of tactics is necessarily incomplete*". We defined an additional set of tactics to completely model the tactic topologies for the considered patterns. Table 3 lists the tactics along with the quality requirements they achieve and the quality attributes of the quality requirements.

Bass et al tactics		
Tactic	**Quality requirement**	**Quality Attribute**
Apply Polymorphism	Variant modules need to be exchangeable at runtime.	Substitutability
Restrict communication paths	Hide a set of modules/services.	Modifiability
Maintain Semantic Coherence	High-level decomposition of an application.	Modularity
Maintain Multiple views	Handle multiplicity in user-interface requirements.	Usability
Parameterize representative	Abstraction over variant modules.	Extensibility
Register at runtime	Dependents of an object are known at runtime.	Adaptability
Additional tactics		
Compose whole from parts	Represent module groups.	Maintainability
Notify modification	State change in one object requires state change in other objects.	Adaptability
Enumerate representatives	Abstraction over variant modules.	Extensibility

Table 3: Tactics used in pattern topologies.

4.2. Analyze tactics from pattern description.

The problem part of the pattern provides information of some of its design decisions in the form of its *benefits*. The described benefits or quality requirements are used as clues to recover some of its constituent tactics by mapping the pattern benefits upon the quality requirements of the tactics.

The UML structure in the solution part of the pattern also provides information of some of its design decisions. UML templates of tactics are used to analyze the tactics from pattern UML structure.

In this section, we provide details of analyzing the inherent tactics for the Observer pattern - tactic analysis for other patterns can be found at http://www.cse.iitk.ac.in/users/vkirankr/Patterns_to_Tactics.doc.

Table 4 illustrates the tactic analysis of Observer (a GoF [9]) pattern. Table 5 illustrates the tactic analysis of MVC (a POSA1 [5]) pattern.

It is to be observed from Tables 4 and 5 the differences in pattern description templates of GoF and POSA1 patterns. The template of GoF patterns contains *Intent*, *Motivation*, *Applicability*, *Structure*, *Participants*, *Collaborations*, *Consequences*, *Implementation*, *Sample Code*, *Known Uses*, and *Related Patterns* sections. Whereas, the POSA1 patterns template contain *Intent*, *Example*, *Problem*, *Solution*, *Structure*, *Dynamics*, *Implementation*, *Variants*, *Known Uses*, *Consequences*, *See Also* sections. Although these two templates differ in naming conventions, the two templates are nearly similar to each other. Henninger et al [14] also discusses the similarities and differences among GoF, POSA, and PLML templates.

Applicability section of Observer Pattern description		
Requirement	**Elaboration of requirement**	**Achieved through tactic(s)**
"*When a change to one object requires changing others*."	State change in one object requires state change in other objects.	Notify modification
"*You don't know how many objects need to be changed*."	Dependents of an object are known at runtime.	Register at runtime.
"*When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled*."	Variant modules need to be exchangeable at runtime.	Apply Polymorphism
Consequences section of Observer Pattern description		
"*Abstract coupling between Subject and Observer*."	Variant modules need to be exchangeable at runtime.	Apply Polymorphism
"*Support for broadcast communication*."	Variant modules need to be exchangeable at runtime.	Apply Polymorphism
UML diagram of Observer Pattern		
Tactic	**UML Textual form from [9]** (↑ means inheritance)	
Register at runtime	Subject.attach(), Subject.detach()	
Notify modification	Subject.notify()	
Apply Polymorphism	**Instance 1:** • Subject, ConcreteSubject • Subject ↑ ConcreteSubject **Instance 2:** • Observer, ConcreteObserver • Observer ↑ ConcreteObserver	

Table 4: Analysis of tactics for Observer pattern [9].

Problem section of MVC Pattern description		
Requirement	**Elaboration of requirement**	**Achieved through tactic(s)**
"*Different users place conflicting requirements on the user interface*."	Handle multiplicity in user-interface requirements.	Maintain Multiple views
"*Building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core*."	High-level decomposition of an application.	Maintain Semantic Coherence

"*The same information is presented differently in different windows, for example, in a bar or pie chart*."	Handle multiplicity in user-interface requirements.	Maintain Multiple views
"*Changes to the user interface should be easy, and even possible at run-time*."	State change in one object requires state change in other objects.	Notify modification
"*Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application*."	High-level decomposition of an application.	Maintain Semantic Coherence
"*The display and behavior of the application must reflect data manipulations immediately*."	State change in one object requires state change in other objects.	Notify modification
Consequences section of MVC Pattern description		
"*Multiple views of the same model*."	Handle multiplicity in user-interface requirements.	Maintain Multiple views
"*Synchronized views*."	State change in one object requires state change in other objects.	Notify modification
"*'Pluggable' views and controllers*."	Dependents of an object are known at runtime.	Register at runtime
"*Exchangeability of 'look and feel'*."	Variant modules need to be exchangeable at runtime.	Apply Polymorphism
Solution section of MVC Pattern description		
Solution description		**Tactic**
"*The MVC architectural pattern comprises three types of participating components: clients, model, views, and controllers*."		Maintain Semantic Coherence
"*There can be multiple views of the model*."		Maintain Multiple views
"*The separation of the model from view and controller components allows multiple views of the same model*."		Maintain Semantic Coherence
"*If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes. The model therefore notifies all views whenever its data changes*."		Notify modification
"*The change-propagation mechanism maintains a registry of the dependent components within the model*."		Register at runtime
"*Changes to the state of the model trigger the change-propagation mechanism*."		Notify modification
UML diagram of MVC Pattern description		
Tactic	**UML Textual form from [5]** (↑ means inheritance)	
Maintain Semantic Coherence	Model, View, Controller	
Maintain Multiple views	View	
Notify modification	Model.notify()	
Register at runtime	Model.attach(), Model.detach()	
Apply Polymorphism	• Observer, View, Controller • Observer ↑ View • Observer ↑ Controller	

Table 5: Analysis of tactics for MVC pattern [5].

4.3. Analyze topology of tactics.

Figures 2 through 6 illustrate the tactic topologies of Observer, Abstract factory, Builder, MVC, and Publisher-Subscriber patterns respectively. Table 6 provides the intent of these patterns for ready reference. Here, we discuss the topology analysis of Observer pattern; other topologies can be analyzed in the similar way.

The intent of the Observer pattern (refer table 6) specifies the design decision *Automatic update notification* as the primary design decision; hence *Notify modification* tactic is made as the source node in Observer TTM.

References to dependent modules are to be known and *all dependent modules need to implement a uniform interface* in order to notify an update to its dependents. *Register at runtime* and *Apply Polymorphism* tactics achieve the above two quality requirements respectively. Hence, the implication of *Notify modification* tactic provides the rationale for using *Register at runtime* and *Apply Polymorphism* tactics. The TTM of Observer pattern depicted in Figure 5 illustrates the tactic dependency.

4.4. Analyze relationships from topologies.

When patterns are represented as tactic topologies, the relationships among patterns can be analyzed by applying the graph rules given in Table 2.

It is to be noted that the label on the edges in TTM represent the rationale of the dependency between the two tactics. In order to improve understandability, the rationale for the same pair of tactics may differ from one pattern to other. But the rationale can be equalized at some higher level of abstraction. Hence, mismatch in the rationale does not affect the relationships among patterns.

Using these rules, the following relationships can be inferred from figures 5 through 9:

- Abstract factory (Figure 3) and Builder (Figure 4) have the same source node but TTMs are different. Hence from Table 2 Abstract factory is *is-an-Alternative-to* Builder pattern. Zimmer [26] comes to the same conclusion independently.
- The TTM of MVC pattern (Figure 5) includes TTM of the Observer pattern (Figure 2). Hence MVC *comprises-of* the Observer pattern. Avgeriou et al [1] and Buschmann et al [5] propound the same relationship.
- Publisher-Subscriber pattern (Figure 6) *refines* Observer pattern (Figure 2). This relationship is also mentioned by Avgeriou et al [1].

4.5 Utility of the TTM: The TTM is a decision view of a design pattern and is a useful description of how the pattern works.

Traceability: The direct linkage from a quality requirement to its corresponding UML fragment is missing in the pattern description. Analyzing tactics from the description bridges the gap between quality requirement and UML fragment and eases the traceability analysis. For example, consider a quality requirement of *Observer* pattern (Table 4) - *don't know how many objects need to be changed***.** This requirement is mapped to *Register-at-runtime* tactic. Analyzing the UML fragment of *Register at runtime* tactic - *Subject.attach()*, *Subject.detach()*, we obtain the traceability link between the quality requirement - *don't know how many objects need to be changed,* and its UML fragment - *Subject.attach()*, *Subject.detach()*.

Relationships: By comparing the TTMs of *Observer* and *MVC*, we were able to infer that *MVC* uses *Observer*. Similarly as shown in Section 4.4, *Publisher-Subscriber* refines *Observer*.

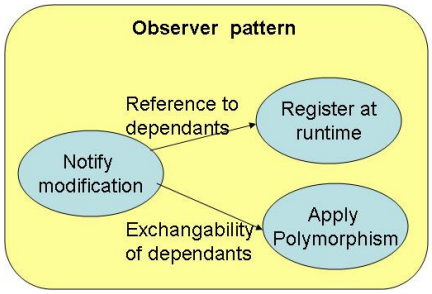

Figure 2: TTM of Observer pattern.

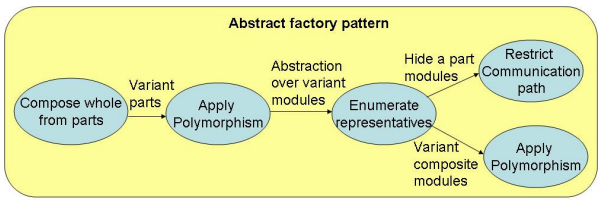

Figure 3: TTM of Abstract factory pattern.

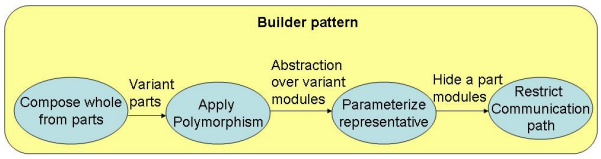

Figure 4: TTM of Builder pattern.

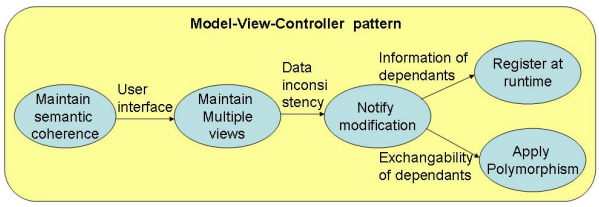

Figure 5: TTM of MVC pattern.

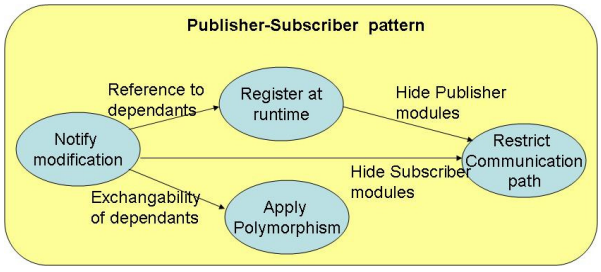

Figure 6: TTM of Publisher-Subscriber pattern.

5. Related Work

Pattern relationship analysis, a sub-problem of the pattern ontology problem, could help in managing the large number of patterns. Zimmer [26] defined three types of relationships and analyzed some relationships between design patterns. Similarly, Avgeriou et. al. in [1] analyzed relationships between architectural patterns. Those relationships were analyzed based on their experience with little or no formal support. Currently, many pattern relationships are specified informally through semantics-free "related to" relationship type [11, 12], but this level of abstraction is unsatisfactory for a designer. Noble in [21] defined and classified various pattern relationship types into primary and secondary categories based on designer needs. Kruchten in [17] defined various relationship types between design decisions.

Analyzing design alternatives (patterns) for a given set of requirements is considered as a knowledge-intensive task [31]. Providing tool support to manage a patterns knowledge base will significantly improve the productivity of the designer. VanHilst and colleagues [32] propose a novel knowledge formalization technique called *Multi-dimensional Concern Matrix* to model the knowledge of *Security* patterns along various stakeholder concerns. This paper also discusses various issues like *Primary dimensions*, *Secondary dimensions* etc to model pattern knowledge. The other advantages of this technique are: gaps in the problem space that lack pattern coverage can be identified easily, and the model is easily extensible to add new dimensions of concern. Our *DDTM* (or TTM) knowledge modeling technique is orthogonal to *Multi-dimensional Concern Matrix* technique. TTM addresses the queries related to relationships between tactics, patterns (and quality requirements) such as:

- What are the patterns which use *Restrict communication path* tactic to improve *Security*? – *Authentication proxy* pattern.
- What are the patterns which use *Compose from parts* tactic? – *Composite*, *Whole-part*, *Abstract factory, Builder* patterns.

It is to be noted that these queries can be modeled using multiple dimensions as: *Tactic*, *Relationship type*, *Quality requirement*.

Similar to our UML templates to analyze tactics, Riele [33] defined UML collaboration templates for patterns to analyze patterns from existing designs of frameworks such as *JUnit*, *Geo system*, *KMU Desktop*, and *JHotDraw*. Riele also defines *Design pattern density* metric to measure what percentage of the framework design can be analyzed through pattern instances. This metric is used to discuss the indications on framework maturity such as: *As the framework matures, the Design pattern density increases* etc.

Non-uniform pattern descriptions [14, 20, 7, 22] and the large number of patterns [14, 13, 19] complicate the problem of pattern relationship analysis. What we need is a uniform pattern description and a formal basis for the analysis of pattern relationships.

Uniform pattern description with natural language is clearly an impractical solution. Modeling pattern semantics through traditional UML semantics seems to be insufficient, modeling patterns precisely using UML is under research [28, 29, 30]. Describing patterns with formal approaches such as eLePUS [23], DiSCO [18], and BPSL [24] is best suited for code generation but not for relationship analysis [14]. Describing patterns as set of property-value pairs [10] enables automation of relationship analysis, but defining a set of properties which is consistent and complete for all the patterns is very difficult.

In recent years, software architecture researchers [25, 16 and 8] proposed that documenting design decisions as first class entities overcomes the *architecture knowledge vaporization* problem.
Currently, for pattern relationship analysis, different formal pattern descriptions exist, such as: eLePUS [23], DiSCO [18], BPSL [24], signs [22], mathematical structures [13], OWL-DL [13]. But these languages are not designer's languages and are hard to use for a designer. Our model helps the designer in ease of describing patterns, enabling the designer to use design decision vocabulary.

6. Conclusions and Future work

. We observe that a decision view of a pattern is useful to analyze relationships amongst patterns. A design decision topology model (DDTM), a decision view, reduces the semantics of a pattern to some syntactic properties of a graph. We propose a particular kind of DDTM, called Tactic Topology Model (TTM) which models design patterns using tactics. This TTM is constructed from the pattern description and its UML diagram. We discussed how our model helps analyze quality requirement traceability and relationships amongst patterns.

Analysis of our Pattern TTMs converges to the same conclusions (relationships between patterns) as described in literature. Our mechanism is amenable for automation and should enable discovery of new relationships.

The work presented in this paper, addresses a sub-problem of a designer's decision support system – ontology of design patterns.

7. Acknowledgements

We are very grateful to our shepherd Eduardo B. Fernandez who had tirelessly read and re-read many versions of the paper and improved both the form and content. Program Committee Member Eric Platon gave several useful and important suggestions which greatly helped. The shepherding process is indeed a very useful practice.

	Pattern	Intent
1	Observer.	Notify the updated state to its dependents automatically, when object state dependency exists between multiple objects.
2	Abstract factory	Separate the representation of a product family from the representation of products. Provide the common interface of product families to create its products and hide the representation of products.
3	Builder	Separate the representation of a complex object from representation of parts. Provide a common construction process to create different complex object representations.
4	Model-View-Controller.	An interactive application is divided into three components – model (core functionality and data), views (user interface), and controllers (handling user input). State changes in a view or in the model are notified to the other views.
5	Publisher-Subscriber.	One publisher notifies any number of subscribers about changes to its state. Publishers register themselves to a broker and subscribers discover publisher from broker.

Table 6: Some patterns from [5, 9].

8. References

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

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

[3] Bass. L., Klein. M. and Bachmann. F, "*Quality Attribute Design Primitives and the Attribute Driven Design Method*", In: Proceedings of the Product Family Engineering vol. 4, Springer-Verlag, Berlin.

[4] F. Bachmann, L. Bass, and R. Nord, "*Understanding and Achieving Modifiability in Software Architecture*", CMU/SEI-2007-TR-002.

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

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

[7] J. Dietrich, and C. Elgar, "*Towards a Web of Patterns*", Proc. Semantic Web Enabled Software Engineering (SWESE), 117-132, Galway, Ireland, 2005.

[8] 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).

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

[10] 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.

[11] S. Henninger, "*An Organizational Learning Method for Applying Usability Guidelines and Patterns*", in Engineering for Human-Computer Interaction (revised papers, EHCI 2001), vol. LNCS 2254, Springer, 2001, pp. 141-155.

[12] S. Henninger, and P. Ashokkumar, "*An Ontology-Based Infrastructure for Usability Design Patterns*", Proc. Semantic Web Enabled Software Engineering (SWESE), Galway, Ireland, pp. 41-55, 2005.

[13] S. Henninger, and P. Ashokkumar, "*An Ontology-Based Metamodel for Software Patterns*", In Proceedings of 18th Int. Conf. on Software Engineering and Knowledge Engineering 2006.

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

[15] 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*", In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5), Pittsburgh, Pennsylvania, 2005.

[16] A. G. J. Jansen and J. Bosch, "*Software architecture as a set of architectural design decisions*", In Proceedings of WICSA 5, pages 109-119, November 2005.

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

[18] T. Mikkonen, "*Formalizing Design Patterns*", Int'l Conf. Software Engineering, pp. 115-124, 1998.

[19] S. Montero, P. Díaz, and I. Aedo, "*A Semantic Representation for Domain-Specific Patterns*", Int'l Symp. on Metainformatics, U. K. Wiil, Ed., Springer-Verlag, LNCS 3511, 2005, pp. 129-140.

[20] J. Noble, "*Towards a Pattern Language for Object-Oriented Design*", Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific), 28, IEEE Comp. Soc., pp. 2-13, 1998.

[21] James Noble, "*Classifying relationships between object-oriented design patterns*", In Australian Software Engineering Conference (ASWEC), pages 98-107, 1998.

[22] James Noble, and Robert Biddle, "*Patterns as Signs*", Proceedings of the 16th European Conference on Object-Oriented Programming, p.368-391, June 10-14, 2002.

[23] S. Raje, and S. Chinnasamy, "*eLePUS-A Language for Specification of Software Design Patterns*", Proc. 2001 ACM Symp. Applied Computing, pp. 600-604, 2001.

[24] T. Taibi, and D. C. Ling Ngo, "*Formal Specification of Design Patterns - A Balanced Approach*", Journal of Object Technology, 2(4), pp. 127-140, 2003.

[25] Tyree, J. and Akerman, A, "*Architecture Decisions: Demystifying Architecture*", IEEE Software, vol. 22, no 2, pp. 19-27, (2005).

[26] Walter Zimmer, "*Relationships Between Design Patterns*", J. Coplien and D. Schmidt, editors, Pattern Languages of Program Design, pages 345_364. Addison-Wesley, 1995.

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

[28] G. Sunyé, F. Pennaneac'h, W.M. Ho, A. L. Guennec, and J. M. Jézéquel, "*Using UML action semantics for executable modeling and beyond*." In Proceedings of the 13th International Conference on Advanced Information Systems Engineering, pages 433-447, 2001.

[29] Robert B. France, Dae-Kyoo Kim, and Sudipto Ghosh, Eunjee Song, "*A UML-Based Pattern Specification Technique*". IEEE Transactions on Software Engineering, pages 193 – 206, 2004.

[30] J. K. H. Mak, C. S. T. Choy, and DPK Lun. "*Precise Modeling of Design Patterns in UML*." In Proceedings of the 26th International Conference on Software Engineering, pages 252-262, 2005.

[31] Pierre N. Robillard, "*The role of knowledge in software development*", Journal of Communications of the ACM, pages 87-92, 1999.

[32] 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.

[33] Dirk Riehle, "*Design pattern density defined*", Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, pages 469-480, 2009.