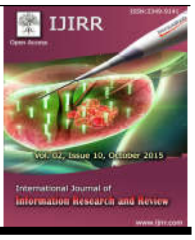
IJIRR

International Journal of Information Research and Review
Vol. 02, Issue, 10, pp. 1232-1236 October, 2015



Review Article

EMPIRICAL ANALYSIS OF DESIGN PATTERN METRICS FOR BUILDING MODEST FORMALIZED CATALOG

***Sriharsha, A.V. and Dr. Rama Mohan Reddy, A.**

Department of CSE, S V University, Tirupati, India

ARTICLE INFO ABSTRACT

Article History:

Received 15th, July 2015
Received in revised form
28th, August 2015
Accepted 29th, September 2015
Published online 31st, October 2015.

Keywords:
Formalization,
Design Patterns,
Software Architecture,
Software Metrics

Design patterns are characteristic structures of classes or objects which can be reused to achieve particular design goals in an elegant manner. As they are not available in the API on any design or development platform, it is very difficult for a developer to perceive the scope and application the design pattern. The formal specification of the design pattern helps in understanding the pattern symbolically. In this paper we propose the quantification of the aspects of a design pattern, earlier not limited to symbolic representation of the methods, attributes, relationships between the classes. By empirically analyzing the metrics it becomes very essential to incorporate the metrics to quantify the application of the class or the object that are embedded in the design pattern, as to build the modest formalized catalog.

Copyright © 2015 Sriharsha and Rama Mohan Reddy. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

INTRODUCTION

Christopher Wolfgang Alexander was the first eminent theorist to introduce patterns as a form of describing accumulated experiences in the field of architecture. He defines a pattern as a construct made of three parts: a context, a set of forces and a solution. The context reflects the conditions under which the pattern holds. The forces occur repeatedly in the context and represent the problem(s) faced. The solution is a configuration that allows the forces to resolve themselves (i.e., balances the forces). Alexandrian patterns comprise commonly encountered problems and their appropriate solutions for the making of successful towns and buildings in a western environment. Alexander called a set of correlated patterns a pattern language, because patterns form a vocabulary of concepts used in communications that take place between experts and novices. Since the mid-’90s, many software systems—including major parts of the Java and .NET libraries and many middleware platforms—have been developed with the conscious awareness of patterns. Sometimes developers applied these patterns selectively to address specific challenges and problems. Other times, they used patterns holistically to help construct software systems, from initially defining baseline architectures to finally realizing fine-grained details.

Corresponding author: Sriharsha, A.V.,
Department of CSE, S V University, Tirupati, India.

Knowledge and conscious application of patterns has become a valuable commodity for software professionals. Much has changed since Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (“the Gang of Four” or “GoF”) published Design Patterns, the most popular book on patterns. Communication software for next-generation distributed applications must be flexible and efficient. Flexibility is needed to support a growing range of multimedia datatypes, traffic patterns, and end-to-end quality of service (QoS) requirements.

Efficiency is needed to provide low latency to delay sensitive applications (such as avionics and call processing) and high performance to bandwidth-intensive applications (such as medical imaging and teleconferencing) over high speed and mobile networks. Many business information systems—such as those for accounting, payroll, inventory, and billing—are based on transactions. The rules for processing transactions are complex and must be flexible to reflect new business practices and mergers.

Business systems must also handle increasingly large volumes of transactions online. The meteoric growth of e-commerce on the Web has exposed many business-to-business systems directly to consumers. Despite these systems’ importance, relatively little has been written about their robust and secure analysis, architecture, or patterns.

The technology landscape has shifted, software design approaches have evolved and expanded, our understanding of development processes has matured, and we know more about documenting and applying patterns to software development. Ironically, Design Patterns is still so popular and influential that many software developers are unaware how much the field has matured or where to find pattern publications that cover a broader range of domains and technologies. The pattern community has long aimed to document and promote good software engineering practices.

This article summarizes the breadth and depth of the patterns in practice to help software developers and managers understand where the field has been and where it's headed, so that you can use patterns in your own projects. Patterns include Design Patterns, Analysis Patterns, and Architectural Patterns. Additionally, these patterns can be classified according to their intentional area of application. A Design Pattern Catalog consists of three categories of patterns they are Creational, Structural and Behavioral. All these patterns are implemented and seen in the software structure at the class level or at the object level. The design patterns of these categories are particularly classified for class level application and object level application. In the creational patterns the design patterns that are seen at the class level applications is Factory Method (FM) pattern, where the rest of the patterns are object level. Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Builder separates the construction of a complex object from its representation so that the same construction process can create different representations.

Prototype specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. Singleton ensures a class only has one instance and provides a global point of access to it. Adapter is however applies at class and object level, converts the interface of a class into another interface clients expect. Decoupling an abstraction from its implementation and so that two can vary independently is done by Bridge pattern. Combining the structures of structures and enable the clients treat them as individual objects and compositions of objects uniformly are contributed by the Composite design pattern. Decorator attaches additional responsibilities to an object dynamically. Façade provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to user. Flyweight is used in sharing to support large numbers of fine-grained objects efficiently. Proxy works as a surrogate or placeholder for another object to control access to it.

Interpreter builds representation of languages; Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure. Decoupling of the sender of a request to its receiver by giving more than one object a chance to handle the request is Chain of Responsibility. Command pattern encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Mediator defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Memento performs encapsulation without any scope violations. Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. State allows an object to alter its behavior when its internal state changes. The object will appear to change its class. Strategy describes the family of algorithms and encapsulates for interchangeability. Visitor represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operate. And after all the descriptive information about the design patterns, the design patterns are categorically studied in various classifications such as object type and class type patterns, creational structural and behavioral design patterns and further the patterns used for decoupling, variants, state-handling, control, virtual machine, convenience.

Metrics

The aim of any given suite of design metrics is to provide objective feedback as to the quality of the design being measured, suggesting aspects of object-orientation which may not have been applied appropriately. There may perhaps be a general lack of subclassing or polymorphism, a high level of unnecessary coupling, or an uneven spread of behavior amongst the classes. This highlighting of the 'weak' areas within a given design then implies a course of remedial action (generally a refactoring), so enabling software quality to be improved (in terms of characteristics such as reduced maintenance time or increased class reusability). The earlier in the lifecycle that a metric is available, the more useful it will be in terms of avoiding backtracking, Conversely, an early metric will be based on low information and it is therefore more ambitious to say that it is predictive of some finished system property. In addition, the granularity of metric available is clearly a function of lifecycle stage and tradeoffs may be required between reducing design and reducing implementation complexity (Tegarden *et al*., 1995).

This goal of producing a definitive limited set of perhaps a dozen orthogonal measures, which are generally predictive in all domains and available at a suitably early stage of the project lifecycle, has prompted much in the way of suggested measures. This is evident from the considerable number of metrics relevant to object-oriented design which have been proposed in the literature (Chidamber and Kemerer, 1994; Henderson Sellers, 1996; Lorenz and Kidd, 1994). Such measures are often initially conceived simply on the basis of intuitive appeal.

A process of refinement is then applied which may include adjustment to a ratio scale (usually the interval [0,1]), and checks for compliance with a chosen set of measurement axioms (Hitz and Montazeri, 1996; Weyuker, 1988; Shepperd and Ince, 1993). These actions serve as an initial filter on the proposed metric, prompting adjustments in order to remove some of its more obviously undesirable properties. However, proof of its actual predictive power has been dependent on correlation with finished system characteristics such as defect rate or percentage of classes reused.

Given the profusion of possible systems, and the use of different implementation languages, empirical studies must be seen to represent limited sampling and taken as a whole seem to produce an excess of (sometimes conflicting) correlations (Basili *et al.*, 1996; Binkley and Schach, 1996; Henry *et al.*, 1995). Other factors also confound metric application. The ambiguous definition of a metric will cause problems as its value will vary according to interpretation (Churcher and Shepperd, 1995; Shepperd and Ince, 1994). Furthermore, when presented with a set of class metric scores M(i) for a set of classes C(i), it is often unclear as to whether the average, minimum or maximum or indeed the standard deviation of M(i) should be limited (rule of application). Finally, concepts such as lack of cohesion have a number of completing metrics which could be used to quantify them (Bansiya *et al.*, 1999; Briand *et al.*, 1997; Li, 1998). The aim is therefore to demonstrate an approach which appears promising in terms of producing a more direct assessment of metric performance, and recommendations for metric selection and rule of applications. The method adopted is based upon an analysis of design metric interaction with well established design patterns structures.

Limitations of Object-Oriented Metrics

ISO/IEC international standard (14598) on software product quality states, "Internal metrics are of little value unless there is evidence that they are related to external quality." It need be noted that the validity of these metrics can sometimes be criticized (Taibi and Ngo, 2003a). Many things, including fatigue and mental and physical stress, can impact the performance of programmers with resultant impact on external metrics. "The only thing that can be reasonably stated is that the empirical relationship between software product metrics are not very likely to be strong because there are other effects that are not accounted for, but as has been demonstrated in a number of studies, they can still be useful in practice (Taibi and Ngo, 2004)."

Design Patterns and Metrics

It is seen that the application of both design metrics and design patterns is geared to a common purpose, namely the elimination of bad design practices. Metrics have generally operated at a higher level, inasmuch as they are usually calculated for an entire system or subsystem in order to provide general recommendations for improvement. A metric score for depth in inheritance tree, for example, may serve to highlight that there has been an excessive abstraction of classes. Design patterns operate at more local level, and do not merely imply what should change but also how. They indicate subsets of the design which are candidates for restructuring, and how this restructuring should be carried out. The application of design patterns in the construction of software is most notably described by Gamma *et al.* (1995). Design patterns represent 'descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context'. As such, they are deemed to provide a suitable test population for evaluating (syntactic) metric behavior. If established design patterns are accepted as being effective building blocks for design, then their usage should be compatible with the metric suite being applied. There is clearly a problem if refactorings to reduce metric scores destroy pattern structures, and vice versa.

In this case the metric, the pattern, or indeed both may prove to be suspect (Ververs and Pronk, 1995). The patterns discussed by Gamma are grouped into three categories, these being creational, structural and behavioural. Design metrics are largely concerned with static class structures rather than dynamic objects, and so the structural and behavioral categories are expected to be the most fruitful in terms of providing patterns for analysis. It is noted that the availability of a design patterns usually implies the existence of a corresponding 'non-pattern' solution, in other words the typical approach taken were the pattern not to be adopted (as opposed to an 'antipattern', Brown *et al.*, 1998). These non-patterns (or 'default' structures) will also be of use, in providing a comparison of the metric scores returned for them against those calculated for the pattern. Such an approach mirrors that adopted by Henderson-Sellers *et al.* (1996) in evaluating their proposed lack of cohesion in methods (LCOM) metric, in that the metric's scores are judged for aptness against a number of structural alternatives.

Metrics for class coupling

The specific category of metrics to be examined initially is those related to class coupling. The term coupling has been used to denote both peer-to-peer and inheritance-based coupling (Schach, 1996); the phrase 'internal coupling' has also been used to describe the intra module method-attribute accessing more commonly referred to as cohesion. The form of coupling to be considered here is peer-to-peer, and may be defined to exist where a given class requires knowledge of another (in terms of making use of its services or accessing its state). It is clear that there must be some communication between objects in order for the required overall behaviour of a system or framework to be implemented. Coupling reduction is not necessarily intended to eliminate module interaction altogether, but rather to reduce the level of 'surplus' connections. The evils of high levels of unnecessary coupling are well known. Classes become harder to understand and test in isolation (Cant *et al.*, 1994), there is an increased likelihood of bug propagation on modification, and there is reduced reusability as atomic units cannot easily be extracted from a system and used elsewhere. It is to the purpose of preventing such problems that coupling metrics have been formulated and applied. As with other aspects of object-oriented design, a number of competing metrics are available. Their form is seen to reflect the varying stages of design at which they are intended to be used.

In the early stages of a design the interaction between classes is likely to be described simply in terms of a binary relationship, whereas at a more detailed stage such connections may be expanded into a number of message sends with method argument lists. Examples of early stage coupling metrics are the coupling between objects (CBO) metric (Chidamber and Kemerer, 1994) and the coupling factor (COF) metric (Abreu *et al.*, 1995). Later stage metrics include the message passing coupling (MPC) metric (Li and Henry, 1993) and the parameters per method (PPM) metric (Lorenz and Kidd, 1994). The detail level at which design patterns are described implies that the binary relationship level will be more appropriate, therefore CBO and COF are selected as possible measures for analysis.

The table below summarizes the metrics discussed above. It illustrates, in general, whether a high or low value is desired from a metric for better code quality. However, one still must exercise judgment when determining the best approach for the task at hand.

Summary of Metrics

Metric	Desirable Value
Coupling Factor (COF)	Lower
Lack of Cohesion of Methods (LCOM)	Lower
Cyclomatic Complexity (CCOM)	Lower
Attribute Hiding Factor (AHF)	Higher
Method Hiding Factor (MHF)	Higher
Depth of Inheritance Tree (DIT)	Low (tradeoff)
Number of Children (NOC)	Low (tradeoff)
Weighted Methods Per Class (WMC)	Low (tradeoff)
Number of Classes (NCL)	Higher
Lines of Code (LOC)	Lower

Certain Predicates

Preliminary understanding of the formalization of classes starts with certain assertive predicates made to the background knowledge of the design pattern schools. Following are the assertions assumed as a preamble of the formalization of the classes.

Assertions

A Pattern Family reflects a philosophical school of thought about pattern evolution
Object is implementation or instantiation of the class (which repeats its encapsulated members)
Class is a particular functionally specified group of members (methods and functions)

We use the following predicate calculus for design the formula of the problem in the domain.

Description	Formalization
The Pattern Catalog belongs to a Pattern Family	(Ax) pattern(x) ➔ belongs(x, patternfamily)
The Patterns used to solve the software development problem are in a Pattern Catalog	(Ay) problem(y) ➔ has(y, pattern)
	(A pattern) (E problem) patternfamily (pattern) ➔ solves(pattern, problem)
Every software developer-designer understands pattern	(A pattern) designer (pattern) ➔ understands (pattern)
	(A x) (E y) (designer (x)^ problem (y)) ➔ understands (x, y) x : pattern; y : problem (specifications)
Each pattern is described by relative group of objects and classes	(Ay) pattern(x,y) ➔ has({x,y}, pattern) x : classes; y : objects

Formalization

Even lacking completeness in their jargon and vocabulary, the existing specification languages promulgate different mathematical sources and incorporate different ingredients, their reflection "*how should patterns be formalized?*" shall perceived meticulously to promote modesty in formalization of design patterns. Formal specification of design patterns is not meant to replace the existing textual/graphical descriptions but rather to complement them to achieve well-defined semantics, allow rigorous reasoning about them and facilitate tool support. Formal specification of design patterns can enhance the understanding of their semantics. Since formalization of classes implies the formalization of structural aspects of the design patterns, which should project knowledge of certain important symbols of the design pattern with respect to formalization of the class.

The structural aspect of design patterns consists of variable symbols, connectives (mainly $\wedge$), quantifiers (mainly $\exists$) and predicate symbols acting upon variable symbols. Variable symbols represent classes, attributes, methods, objects and untyped values while the predicate symbols represent permanent relations. The domain (set) of primary entities that are classes, attributes, methods, objects, and untyped values is designated receptively ***C***, ***A***, ***M***, ***O***, and ***V***.

The following table depicts the primary permanent relations, their domain and their intent. These relations straightforwardly derive from object-oriented technology concepts. It is the smallest set (in terms of number of elements) on top of which any other permanent relation can be built. For a particular class the above the BPSL (Balanced Pattern Specification Language [Toufik Taibi et. al]) notation can be applied as follows:

$\exists$subject, concrete-subject, observer, concrete-observer $\in \boldsymbol{C}$;
subject-state, observer-state $\in \boldsymbol{A}$;
attach, detach, notify, get-state, set-state, update $\in \boldsymbol{M}$;
o, s $\in \boldsymbol{O}$;
d$\in \boldsymbol{V}$;

The modesty of the specification lies in describing its capabilities through metrics. The CK-OO design metrics proposes ranges of values for the various elegant properties of class and object design. Such properties also may be quantified and symbolized along with the BPSL notation. For convenience of symbolization let us assume ***m*** is the variable for the metric. For each metric for the class ***C*** in the following illustration, the range of values (minimum value and the maximum value) may be described.

$\exists$subject, concrete-subject, observer, concrete-observer $\in C_1, C_2, C_3$;
subject-state, observer-state $\in A_1, A_2, A_3$;
attach, detach, notify, get-state, set-state, update $\in M_1, M_2, M_3$;
o, s $\in O_1, O_2, O_3$;
d$\in V_1, V_2, V_3$;

For all the $\boldsymbol{C_1}$, $\boldsymbol{C_2}$, $\boldsymbol{C_3}$ in a hierarchy and the attributes $\boldsymbol{A_1}$, $\boldsymbol{A_2}$, $\boldsymbol{A_3}$ methods $\boldsymbol{M_1}$, $\boldsymbol{M_2}$, $\boldsymbol{M_3}$ whose instances are objects, the metrics for the collection of the classes with their constituent members shall also be represented.

Let us assume all $\boldsymbol{C_1}$, $\boldsymbol{C_2}$, $\boldsymbol{C_3}$ group into the categorical identity called ***C*** and the CK-OO design metrics for the hierarchy ***C*** is as follows. The following representation illustrates that the classes in the ***C*** are associated with the metrics the Attribute Hiding Factor, Lines of Code and Weighted Methods Per Class.

$\exists$AHF, LOC WMC $\in C$;
$C_{AHF} \in$ Pr(Max(Range))
$C_{LOC} \in$ Pr(Min(Range))
$C_{WMC} \in$ Pr(Min(Range))

Table 1. Primary Permanent Relations and their Intent

Name	Domain	Intent
Defined-In	MxC	Indicates that a method is defined in a certain class.
	AxC	Indicates that an attribute is defined in a certain class.
Reference-to-one (-many)	CxC	Indicates that one class defines a member whose type is a reference to one (many) instance(s) of the second class.
Inheritance	CxC	Indicates that the first class inherits from the second.
Creation	MxC	Indicates that a method contains an instruction that creates a new instance of a class.
	CxC	Indicates that one of the methods of a class contains an instruction that creates a new instance of another class.
Invocation	MxM	Indicates that the first method invokes the second method.
Argument	CxM	Indicates that a reference to a class is an argument of a method.
	VxM	Indicates that an untyped value is an argument of a method
Instance	OxC	Indicates that an object is an instance of a certain class.

The above formal specification illustrates the quantification of the metrics of the group of classes C.

DISCUSSION

Formalization of the Design pattern includes basically formalization of the classes of the design. For a developed system formalization is mere symbolization and quantification of the system properties (static and dynamic). For the system proposed to design the formalization is about its design specification which includes user requirement specification and the functional specification of the system. The Formalization is the Formal Specification of the system which represents the symbols and quantifiers of the system.

The DPML or the BPSL whichever the existing formalization languages used for the design pattern formalization use only symbolization of the design pattern aspects. In the proposed modest formalization methods of the design patterns the quantification of the symbols used to represent the design pattern are also considered. It is very important to note that formalization or the formal specification for a software design should also mentions the capacities in which the software artifacts should be used for the solving of software development problem.

Conclusion

Patterns are gaining increasing acceptance and usage. They capture design experience in such a way that they become a learning aid for novice designers. However, the inherent benefits of patterns cannot be fully exploited by the existing informal means of specifying them. Formal specification of patterns allows precise specifications and facilitates tool support.

Formal specification of design patterns is in tended to complement existing textual and graphical descriptions in order to eliminate ambiguity, allow rigorous reasoning about patterns and facilitate automation of the activities related to them. As patterns represent abstractions, any formal language meant to specify them should strive to achieve simplicity for a better understandability, accuracy for a precise semantics and completeness to avoid loss of semantics.

REFERENCES

Alexander, C. 1979. The timeless way of building. Oxford University Press.

Buschmann, F., Henney, K. and Schmidt, D. 2007. Pattern-Oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing, John Wiley & Sons.

Buschmann, F., Henney, K. and Schmidt, D.C. 2007. Pattern-Oriented Software Architecture Vol. 5: On Patterns and Pattern Languages, John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. 1996. Pattern-Oriented Software Architecture: Vol. 1 A System of Patterns. West Sussex, England: John Wiley and Sons Ltd.

Coad, P. 1992. Object-Oriented Patterns, in: Communications of the ACM, September, p. 152-159.

Coplien, J. and Harrison, N. 2005. Organizational Patterns of Agile Software Development, Prentice Hall.

Fowler, M. 1997. Analysis Patterns - Reusable Object Models. Menlo Park: Addison-Wesley.

Fowler, M. 1999. Refactoring: Improving the design of existing code. Addison-Wesley.

Fowler, M. 2003. Patterns of Enterprise Application Architecture, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R.and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Rumbaugh, J., Jacobson, I., & Booch, G. (1998). The unified modeling language reference manual. Addison-Wesley Professional.

Schumacher *et al.*, M. 2006. Security Patterns: Integrating Security and Systems Engineering, John Wiley & Sons.

Taibi, T. and Ngo, D.C.L. 2003a. Formal specification of Design patterns-A balanced approach. *Journal of Object Technology*, 2(4), 127-140.

Taibi, T. and Ngo, D.C.L. 2003b. Formal specification of Design patterns: A comparison. In Proceedings of the 1st ACS/IEEE International Conference on Computer Systems and Applications (AICCSA) (pp. 77-86). IEEE Computer Society Press.

Taibi, T. and Ngo, D.C.L. 2004. Towards a balanced specification language for distributed object computing patterns. IASTED International Journal of Computers and Applications, 26(1), 63-70.

Utas, G. 2005. Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems, John Wiley & Sons.

Voelter, M., Kircher, M. and Zdun, U. 2004. Remoting Patterns, John Wiley & Sons.

****