International Journal of Latest Research in Engineering and Technology (IJLRET)
ISSN: 2454-5031(Online)
www.ijlret.com‖ Volume 1 Issue 3‖August 2015 ‖ PP 52-57

Modest Formalization of Software Design Patterns

A.V.Srihasha[1], Dr. A. Rama Mohan Reddy[2]
[1](PhD Research Scholar, Department of CSE, S V University, Tirupati, India)
[2](Professor, Department of CSE, S V University, Tirupati, India)

ABSTRACT: Formalization is the document form of formalism, where the practical compositional elements are represented by the symbols and variables. The Software Requirement Specification is documented in such a way that it breaks the deliverables into smaller components. Design patterns are among the most powerful methods for building large software systems. Patterns provide well-known solutions to recurring problems that developers face. Predicate logic is used for describing the formal specification of the design patterns. In this paper we urge to explain that formal specification of design patterns is very essential before they are implemented in any platform, further the formal specification of the design pattern is derived into a formula with respect to the application of the domain. In this paper we state some of the illustration to understand the concept of the formal specification and formula and we call this Modest Formalization of Software Design Patterns.

KEYWORDS – modesty, formalization, design patterns, software architecture, calculus.

I. INTRODUCTION

In art theory, formalism is the concept that a work‘s artistic value is entirely determined by its form–the way it is made, its purely visual aspects, and its medium. Formalism emphasizes compositional elements such as color, line, shape and texture rather than realism, context, and content. The philosopher Nick Zangwill of Glasgow University has defined formalism in art as referring to those properties "*that are determined solely by sensory or physical properties—so long as the physical properties in question are not relations to other things and other times*." The philosopher and architect Branko Mitrovic has defined formalism in art and architecture as "*the doctrine that states that the aesthetic qualities of works of visual art derive from the visual and spatial properties*." A formal analysis is an academic method in art history and criticism for analyzing works of art: "In order to perceive style, and understand it, art historians use '*formal analysis*'. This means they describe things very carefully. These descriptions, which may include subjective vocabulary, are always accompanied by illustrations, so that there can be no doubt about what exists objectively".

Formalization is the document form of formalism, where the practical compositional elements are represented by the symbols and variables. However, the theoretical impact on formalization has often been obscured in empirical investigations; the concept of building the basic idea of a system remains unchanged. Formalization (as efficiency) is likely to contribute to effectiveness early even in an organization's history. Formalization is defined high level at the implementation and so each component has to be clearly defined in its role of specialization.

II. SOFTWARE ENGINEERING PERSPECTIVES

Software Requirement Specification assures the project management stakeholders and client that the development team has really understood the business requirements documentation properly. The Software Requirement Specification is documented in such a way that it breaks the deliverables into smaller components. The information is organized in such a way that the developers will not only understand the boundaries within which they need to work, but also what functionality needs to be developed and in what order. These two points are particularly important in the process of software development. If a development team does not understand that there are certain constraints on their work, as for example the code must be tightly written so that it will compile and run quickly, then problems will creep later on when the code might deliver the functionality required. Understanding what order the functionality will be developed in means that the developers have the "big picture" view of the development. This gives them an opportunity to plan ahead which saves both project time and cost. As for some of the important characteristics to be followed in SRS of a Software Development

activity, accuracy, clarity, completeness, consistency, prioritization of requirements, verifiability, modifiability, traceability, etc., the formalization improves the specification's readability and understandability.

Software Development process can be divided into smaller, interacting sub processes. Generally software development can be seen as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation.

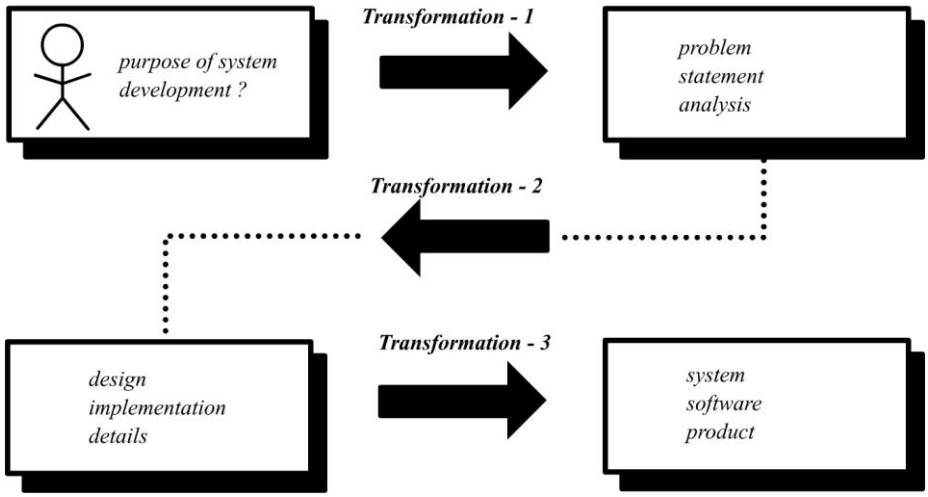

Fig II. (1a): Transformations in Software Development Process.

III. SOFTWARE DESIGN PATTERNS

Design patterns are among the most powerful methods for building large software systems. Patterns provide In 1987, Ward Cunningham and Kent Beck were working with Smalltalk and designing user interfaces. They decided to use some of Alexander's ideas to develop a small five pattern language for guiding novice Smalltalk programmers. They wrote up the results and presented them at OOPSLA'87 in Orlando in the paper "Using Pattern Languages for Object-Oriented Programs". Soon afterward, Jim Coplien (more affectionately referred to as "Cope") began compiling a catalog of C++ idioms (which are one kind of pattern) and later published them as a book in 1991, Advanced C++ Programming Styles and Idioms. From 1990 to 1992, various members of the Gang of Four had met one another and had done some work compiling a catalog of patterns. Discussions of patterns abounded at OOPSLA'91 at a workshop given by Bruce Andersen (which was repeated in 1992). Many pattern notables participated in these workshops, including Jim Coplien, Doug Lea, Desmond D'Souza, Norm Kerth, Wolfgang Pree, and others. In August 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado, the first meeting of what is now known as the Hillside Group. Another patterns workshop was held at OOPSLA'93 and then in April of 1994, the Hillside Group met again (this time with Richard Gabriel added to the fold) to plan the first PLoP conference. Thereafter the GoF book is published and the history of Design Patterns is classified. Current pattern representations are textual. They include the Gang-of-Four (GoF) form, the Coplien form, and the Alexandrian form. The GoF form (Gamma et al., 1994) includes sections for intent, motivation, structure, participants, and collaborations. The emphasis of this format is on the structure of the solution. However, the discussion of the forces is spread out over multiple sections, which makes it challenging for a developer to get an overview of when to apply a particular pattern and the consequences of using it.

IV. FORMALIZATION OF PATTERNS

Design patterns are among the most powerful methods for building large software systems. Patterns provide well-known solutions to recurring problems that developers face. There are several benefits of using patterns if they are applied correctly. Although design patterns are only over a few decades old, the science of patterns is becoming established, allowing for consistent communication. By using well-known patterns

reusable components can be built in frameworks. Providing frameworks for reusability and separation of concerns is the key to software development today.

First-order logic (aka. first-order predicate calculus) is a formal system used in mathematics, philosophy, linguistics, and computer science. It is also known as first-order predicate calculus, the lower predicate calculus, quantification theory, and predicate logic. First-order logic uses quantified variables over (non-logical) objects. This distinguishes it from propositional logic which does not use quantifiers. The adjective "first-order" distinguishes first-order logic from higher-order logic in which there are predicates having predicates or functions as arguments, or in which one or both of predicate quantifiers or function quantifiers are permitted.[3] In first-order theories, predicates are often associated with sets. In interpreted higher-order theories, predicates may be interpreted as sets of sets.

Programming languages has many intense characteristics that fit well into the formal syntax and semantics in order to be executed on a computer. In general for the early phases of a software project, however, the intended behaviour of a program has to be specified in an abstract way, using some kind of specification language. Formal specification languages can be used successfully for non-trivial pieces of software, like Z and LARCH. These formal specification languages force the specifier to express him- or herself in terms of mathematical logic.

In 1974, Jean-Raymond Abrial published "Data Semantics" [1] and further used notation that would later be taught in the University of Grenoble until the end of the 1980s. While at EDF (Électricité de France), Abrial wrote internal notes on Z. The Z notation is used in the 1980 book Méthodes de programmation [2].

V. FORMAL SPECIFICATION OF DESIGN PATTERNS

In specifying structural aspects of Design patterns, we investigated a formal specification method using general first-order logic to represent each Design pattern structure as a logic theory (Dong et al., 2000). To illustrate the problem, let us consider the Composite pattern and the Iterator pattern from (Gamma et al., 1995) as examples. The structural aspect of the Composite and Iterator patterns is depicted in Figure 1. The Component class is an abstract class which defines the interfaces of the pattern. The Composite and the Leaf classes are concrete classes defining the attributes and operations of the concrete components. The Composite class can contain a group of children, whereas the Leaf class cannot. The Composite pattern is often used to represent part-whole hierarchies of objects. The goal of this pattern is to treat composition of objects and individual objects in the composite structure uniformly. In the Iterator pattern, the Iterator class is an abstract class which provides the interfaces of the operations, such as First, Next, IsDone, CurrentItem, to access the elements of an aggregate object sequentially without exposing its underlying representation. The ConcreteIterator class inherits the operation interfaces from the Iterator class and defines concrete operations which access the corresponding concrete aggregate. The Aggregate class defines a common interface for all aggregates that the Iterator accesses. The ConcreteAggregate class defines an operation to create the corresponding concrete Iterator.

The representations of the Composite pattern and the Iterator pattern contain predicates for describing classes, state variables, methods, and their relations. More precisely, the following sorts denote the first-class objects in a pattern: class and object. We also make use of sorts bool and int. The signature for the Composite pattern is:

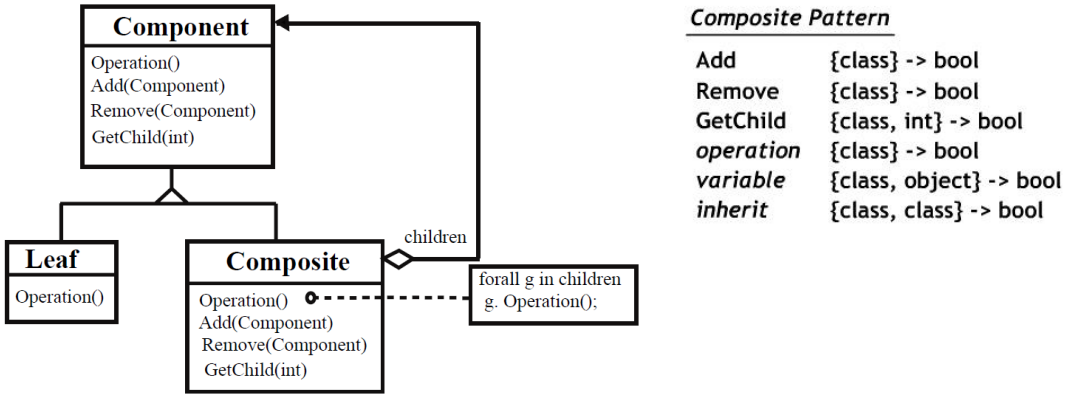

Fig IV. (1a): Composite Design Pattern and its Signature.

The signature of the Iterator pattern is:

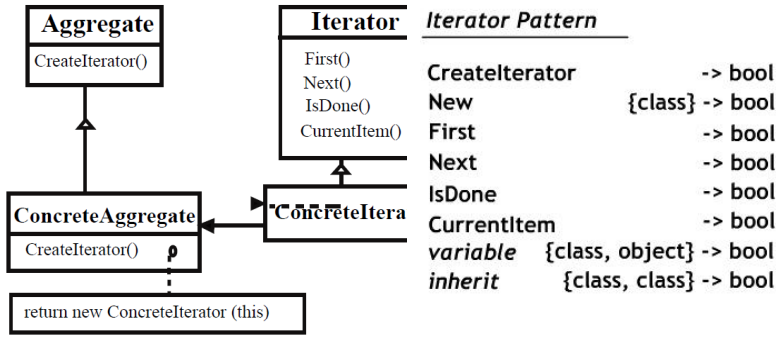

Fig IV. (1b): Iterator Design Pattern and its Signature.

The above Fig IV.(1a) and IV.(1b) contains (partial) theories associated with the two patterns. $\boldsymbol{\Theta_C}$ denotes the theory of the Composite pattern and $\boldsymbol{\Theta_I}$ denotes the theory of the Iterator pattern. The theory $\boldsymbol{\Theta_C}$ is divided into three class groups and one relation group. The first group defines the abstract class Component and four method interfaces. The second group corresponds to the Leaf class. The third group contains theories about the Composite class, which include the definition of a state variable and the operations applied to it. The last group defines two inheritance relations. The first class in each inheritance relation is the parent class and the second class is the child class. The theory $\boldsymbol{\Theta_I}$ is divided into five groups. The first four groups contain theories about four classes in the pattern. The last group contains two inheritance relations.

$\boldsymbol{\Theta_C}$	$\boldsymbol{\Theta_I}$
AbstractClass(Component) Operation(Component) Add(Component) Remove(Component) GetChild(Component, int)	AbstractClass(Aggregate) CreateIterator
	Class(ConcreteAggregate) CreateIterator→New(ConcreteIterator)
Class(Leaf) Operation(Leaf)	AbstractClass(Iterator) First Next IsDone CurrentItem
Class(Composite) Variable(Component, Children) Operation(Composite)→[∀g[Children(g)→Operation(g)]] ∀v [Add(v)→Children(v)] ∀v [Children(v)→Remove(v)] ∃v [Children(v) ∧ GetChild(v, int)]	Class(ConcreteIterator) Variable(Aggregate, aggregates)
Inherit(Component, Leaf) Inherit(Component, Composite)	Inherit(Aggregate, ConcreteAggregate) Inherit(Iterator, ConcreteIterator)

Table V (1): Theories associated with the two patterns Composite and Iterator.

VI. MODEL

Design patterns are represented in many programming languages that support Object Oriented paradigm. Design patterns are represented in terms of object-oriented design primitives in a predicate like formats. Each design primitive consists of two parts: *name* and *argument*. The *name* part contains the name of a feature or a relationship in object-oriented design, such as class or inheritance. The *argument* part contains general information about a feature or a relation such as the information on the participants of an inheritance relationship. Some of the examples are given below table. A higher level of abstraction is provided by introducing pattern primitive operators. Pattern primitive operators are represented in terms of design primitive operators and they allow general object-oriented schemas such as delegation, aggregation, and polymorphism to be defined. Pattern primitive operators can capture the subpatterns, which occur frequently in the declarative representation of Design patterns. They can also be used to change, transform, or make the declarative representation evolve. This operator can assist with the evolution of the pattern schema and also with the application of this pattern.

Class(C): **C** is a class.
Inherit(A, B): B is a subclass of **A.**
Attribute(C, A, V, T): V is the name of an attribute in class **C** with type **T**. **T** is optional. **A** describes the access right of this attribute, that is, public, private, or protected.
Method(C, A, F, R, P_1, T_1, P_2, T_2, ...): F is a method of a class **C**. **A** describes the access right of this method, that is, it can be public, private, or protected. **R** describes the return type. The method's parameters and their types are **P_1**, **T_1**, **P_2**, **T_2**, ..., respectively, and this part is optional. The return type **R** is also optional if the method has no parameters.
Member(E_1, S_1, E_2, S_2, ...): E_1 is an element of set S_1. E_2 is an element of set S_2, and so on. When universal quantification forall and member are used together, it enumerates set S_1, S_2, ..., S_n simultaneously, that is, the first elements of all sets are enumerated first, then the second elements.

Table V I(1): Model Formalisms for Object Oriented Concepts.

VII. PREDICATE LOGIC

In mathematical logic, predicate logic is the generic term for symbolic formal systems like first-order logic, second-order logic, many-sorted logic, or infinitary logic (*An infinitary logic is a logic that allows infinitely long statements and/or infinitely long proofs*). This formal system is distinguished from other systems in that its formulae contain variables which can be quantified. The two common quantifiers are the existential ∃ ("there exists") and universal ∀ ("for all") quantifiers. The modest approach of this formalization is a gradual development of the formula for the application of design patterns in the software development.

The design patterns of a design pattern school are set up in a catalog which is a searchable data structure for finding suitable design pattern application for the software development problem in the domain. The organization of the catalog contains formal specifications of the design patterns. In general, the formal specification of object oriented concepts is induced into an application driven representation of classes and objects as domain specific formulae. In this context of designing the modest formalization for design patterns, we embark on the approach for designing the formal specification for the design patterns and later their implementation indications are directed to develop the formula, further to directly use them in the software development. Certain assertions are made before the generation of formula for the formal specifications derived from the modest thought.

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	(A*x*) **pattern**(*x*) ➔ *belongs*(*x*, **patternfamily**)
The Patterns used to solve the software development problem are in a Pattern Catalog	(A*y*) **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	(A*y*) **pattern**(x,*y*) ➔ *has*({x,*y*}, **pattern**) **x : classes; y : objects**

VIII. CONCLUSION

Formalization is a very essential activity to be done for every software development problem, in order to understand the quantified application of the resources. Formalization with respect to a platform or a development tool limits the functional applications and derivations of the formal representation. Thus in this paper we urge to develop a formal specification for the generic design and the formula for the constituent design of the software. May categories of Predicate Logic exists where we can take the formal specification into a higher resolution.

REFERENCES

[1] Toufik Taibi, "*Design Pattern Formalization Techniques*", United Arab Emirates University, UAE, (C) 2007, IGI Publishing Hamburg, New York.
[2] Henderson-Sellers, Brian. "*Object-Oriented Metrics Measures of Complexity*", Upper Saddle River, NJ: (C) 1996, Prentice Hall.
[3] Clark Archer, Michael Stinson, "*Object-Oriented Software Measures*", Technical Report submitted to Software Engineering Institute, Carnegie Melon University, CMU/SEI-95-TR-002, April 1995.
[4] Ian Somerville, "*Software Engineering*", 8th Edition, Chapter 27, Formal Specification, Prentice Hall Inc. (C) 2009.