On the Cognitive Foundation and Abstract Means of Engineering Designs

Yingxu Wang, Prof., PhD, PEng, FWIF, SMIEEE
Theoretical and Empirical Software Engineering Research Centre
Dept. of Electrical and Computer Engineering
Faculty of Engineering, University of Calgary
2500 University Drive, NW, Calgary, Alberta, Canada T2N 1N4
Tel: (403) 220 6141, Fax: (403) 282 6855
Email: yingxu@ucalgary.ca

Abstract

Formal inference processes in all kinds of engineering designs are based on the cognitive process and means of abstraction. This paper presents a formal treatment of system design. It first discusses the differences of system designs in abstract and concrete systems. A hierarchical abstraction model of system descriptivity (HAMSMD) is developed that explains the abstract levels of entities and artifacts in both the concrete world and the abstract world. The HAMSMD model classifies the design of a system at five levels known as the levels of (1) analogue objects, (2) diagrams, (3) natural languages, (4) professional notations, and (5) mathematics from the bottom up. According to the HAMSMD model, an architectural, functional, or behavioral design of a system should adopt a higher-level abstract means to describe the system at a given abstract level, in order to obtain more powerful descriptivity and higher accuracy. Therefore, a mechanical system may be designed by using diagrams; while an information or software system cannot be precisely described by diagrams. The reason is that the former is at an abstract level lower than the descriptive means of diagrams, but not so is the latter.

Applications of the above findings in dealing with complicated problems in large-scale software system designs are discussed in the context of software engineering. Real-time process algebra (RTPA) is adopted as a highly abstract means in case studies on formal methodologies of software system design and modeling for software objects and design patterns.

Keywords: Design engineering, abstract means, cognitive foundation, RTPA, mathematical modeling, architectural design, behavioral design, design patterns, software engineering

1. INTRODUCTION

Engineering design is a cognitive process that represents the architectures and/or behaviors of a system using concrete or abstract means and notations. Design is the most centred activities and processes in software engineering than in the other engineering disciplines.

The following humour known as the ‘cat theory’ was presented at NASA’s web site [4] on methodology maturities of designs in various engineering disciplines:

- **Mechanical engineering** is like looking for a black cat in a lighted room.
- **Chemical engineering** is like looking for a black cat in a dark room.
- **Software engineering** is like looking for a black cat in a dark room in which there is no cat.
- **System engineering** is like looking for a black cat in dark room in which there is no cat and one yells, “I got it!”

Why the cat is only in the mechanical or chemical rooms? Because these disciplines have already developed rigorous theoretical frameworks and suitable mathematical means, and the other disciplines did not. The same situation of lacking theoretical foundations is still challenging software engineering, system engineering, and any other immature science and engineering disciplines.

It is found that the expressive power of icons and diagrams are inadequate in software engineering because they make software design and specifications vague. Why visual means may be used adequately in system designs of mechanical engineering and civil engineering, but not adequate in software engineering? The law of explicit descriptively [15] states that only a higher level of more abstract, precise, and rigor means is adequate and accurate to express an object at a given level of abstraction. It is recognized that not only the architectures of software are complex interrelated objects with functional variables and constraints, but also the behaviors of software are embedded relational processes. These type of abstract and complicated entities may only be expressed without
implication by professional notation systems, because only much more abstract and precise means is powerful enough to express an object at a lower level of abstraction.

This paper describes the cognitive difficulties of engineering designs and the abstract means for dealing with these problems. Section 2 analyzes the cognitive foundation of system descriptivity and models the hierarchy of abstraction and the descriptivity of abstract means at different levels. Section 3 discusses generic design methodologies and contrasts the differences of abstract and concrete system designs. Sections 4 and 5 demonstrate applications of abstraction and formal inferences in dealing with complicated design problems in software engineering, particularly for software design patterns.

2. THE HIERARCHICAL ABSTRACTION MODEL OF HUMAN DESCRIPTIVITY

It is recognized that software engineering seems using low-tech means to deal with high-tech problems [14, 15]. Consider that the accuracy of micrometer technologies is at the level of $10^{-6}\text{m}$, while that of nanometer technologies is at the level of $10^{-9}\text{m}$. If one asks whether a microtechnique may be used to denote, measure, or process a nanotechnical product, mechanical or electronic engineers will tell that is impossible! However, software engineers’ practice is still trying to do this using inadequate means and tools. Various graphical blocks and arrows are proposed to denote more intricate system architectures and dynamic behaviors of software systems. Further, academics and practitioners in software engineering seem used to the practice. Both the technique and the attitude form the profound problems of software engineering and are the fundamental reasons of almost all failures in industrial practice in software engineering.

This observation leads to the discussion that what kind of descriptive means of software engineering needs? The answer is abstraction. *Abstraction* is a powerful and fundamental mental function of human beings that most of the higher cognitive processes of the brain are relied on [12, 13]. Abstraction is a meta cognitive process of the brain at the meta cognitive layer that establishes an abstract model (or concept) for an entity of external world by eliciting the information of its common and qualitative/quantitative attributes or properties in order to mentally process it.

Abstract objects only exist in the brain as a thought or idea but not exist in the real world as a physical or concrete entity. However, the sources of abstract objects are reflections of real world entities, phenomena and their relations. Abstraction is a powerful key to reduce complexity in creative work such as software engineering. Abstraction is a software engineering principle for eliciting essential properties of a set of objects while omitting inessential details of them.

Set is a fundamental and powerful mathematical concept for abstracting and eliciting objects that share certain common properties. According to the founder of set theory, G. Cantor, abstraction is an elicitation of common properties of elements from a given set [3].

**Theorem 1.** The principle of abstraction states that, given any set $S$ and any property $p$, abstraction is to elicit a subset $E$ such that the elements of it, $e$, possess the property $p(e)$, i.e.:

$$\forall S, p \Rightarrow \exists E \subseteq S \land \forall e \in E, p(e)$$

(1)

According to the information-matter-energy (IME) model [12, 14], there are two categories of objects under studies in science and engineering known as the concrete entities in the real world and the abstract objects in the information world. In the latter, an important part of the abstract objects are human or system behaviors, which are planned or executed actions onto the real-world entities and abstract objects.

The abstract levels of cognitive information of both the objects and their behaviors can be divided into five levels of analogue objects, diagrams, natural languages, professional notations, and mathematics. A hierarchical abstraction model of system descriptivity (HAMSD) of human knowledge is shown in Fig. 1.

Since software is an abstraction of both real-world objects and their relations, and the expected and executed behaviors of a system that, in turn, are real or expected human behaviors [15], the means and nature of software obeys the same HAMSD model as shown in Fig. 1.

It is noteworthy that in software engineering the objects under study are system and human behaviors in the abstract world rather than concrete entities in the real world. This is a fundamental difference between software engineering and other engineering disciplines. This can be formally described in the following principle of software engineering.

**Theorem 2.** The abstract objects principle of software engineering states that fundamental constraints of software engineering stem from intangibility and intricate inner connections of software objects and systems, and the cognitive complicity to explicitly describe them.
Abstract levels of objects

<table>
<thead>
<tr>
<th>Levels</th>
<th>Objects under study</th>
<th>Abstract levels of objects</th>
<th>Means of description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Analogue objects</td>
<td></td>
<td>M1 Images, icons, pictures</td>
</tr>
<tr>
<td>L2</td>
<td>Diagrams</td>
<td></td>
<td>M2 Blocks, lines, arrows, blue prints, schematics</td>
</tr>
<tr>
<td>L3</td>
<td>Natural languages</td>
<td></td>
<td>M3 Languages syntaxes and semantics</td>
</tr>
<tr>
<td>L4</td>
<td>Professional notations</td>
<td></td>
<td>M4 Symbolic notations with syntax and semantics</td>
</tr>
<tr>
<td>L5</td>
<td>Mathematics</td>
<td></td>
<td>M5 Formal notations with rigor syntax and semantics</td>
</tr>
</tbody>
</table>

Fig. 1 The abstraction and descriptivity model (ADM) of software engineering

There are two approaches to system description as shown in Fig. 1 known as abstraction and explanation. The former enables the increase of the descriptive power in terms of expressiveness, precise, and rigor; while the latter enables the improvement of the intuitiveness of understanding and comprehension, using a means of description that is much closer to the real world images and analogue objects directly acquired by the sensations of the brain [14].

3. METHODOLOGIES OF DESIGN: ABSTRACT VS. CONCRETE SYSTEMS

In contrast to physical, mechanical, and geometrical designs in the concrete world, software design is carried out in the abstract world where only things that can be embodied are blocks, lines, or arrows, which show connections or relations between architectural and/or behavioral components at different levels. In other word, software behaviors, particularly the dynamic aspect, are inexpressible by conventional means of diagrams.

For instance, geometry is the ultimate form of graphical reasoning, and it is much rigor than any software block diagrams and class diagrams. However, even for that, algebraic geometry had to be developed to deal with more complicated structures and inferences in geometrical analyses.

Corollary 1. The expressive power of icons and diagrams are inadequate in software engineering because they make software design and specifications vague.

To indirectly prove Corollary 1, one may try to read a cartoon after erasing all text and explanations. It is obvious that one cannot obtain too much accurate information from the cartoon. Making things worse, different persons may obtain different information and perceive different meanings from the same picture. This is totally unacceptable in engineering design.

Why visual means may be used adequately in system designs of mechanical engineering and civil engineering, but not adequate in software engineering? The answer is because the objects of study of the former are physical and geographical located at Level 1 according to the HAMSD model; while those of the latter are abstract objects located at Level 3 or above in HAMSD.

It is recognized that architectures of software are complex interrelated objects with functional variables and constraints. Behaviors of software are embedded relational processes [15]. These types of abstract and complicated entities may only be expressed without implication by professional notation systems, because only more abstract and precise means is powerful enough to express an object at a given level of abstraction. This leads to the following theorem.
Theorem 3. The principle of explicit descriptively of software engineering states that only a higher level of more abstract, precise, and rigor means is required to express an object at a given level of abstraction.

According to Theorem 3, software architectures and behaviors cannot be explicitly expressed by diagrams, because the abstraction level of the latter is lower than the former. However, diagrams can be used to express and denote physical architectures and designs, because the abstraction level of the latter are higher than that of the former. This explains that, although visualization is a powerful means to form a mental image and visible representation of a design by means of diagrams and pictures in other engineering disciplines, it is not adequate in software engineering.

Corollary 2. Symbolic notations and mathematics are the key means for expressing and embodying software behaviors, because they are at higher level abstraction and therefore with more adequate descriptive power.

The descriptive power of formal notation systems in software engineering lies in its adequacy for describing abstract objects, their relations, behaviors, and for enabling rigorous inferences based on formally defined composing rules in terms of formal syntaxes and semantics. Although diagrams are widely used to represent or abstract physical designs, they are not suitable for expressing and embodying more abstract and intricately interconnected entities like software systems. Because using graphical means in system specification and refinement is just like using conventional microtechniques to denote and measure nanotechnologies.

Therefore, in software engineering, graphical means may be used to describe rough conceptual models of software systems for human beings, but not for the precise specifications of system architectures and behaviors in system modeling and execution. Lessons learnt from all the failures in the last four decades indicates, no matter how convenient it might appear to be, intuitive comprehension should not be an excuse to stack at any graphical and visual means for describing complicated software architectures and behaviors. This is what Theorem 3 and Corollary 2 reveals and suggests.

4. THE GENERIC MODEL OF DESIGN PATTERNS IN SOFTWARE ENGINEERING

On the basis of the HAMSD model and the generic principles of complicated system designs described in Sections 2 and 3, this section develops a formal model of software design patterns in software engineering.

Information hiding is a widely accepted principle for software engineering designs identified by Parnas [5, 6], which supposes that unnecessary details of information of software at a certain level should be masked in lower level implementations. Parnas viewed that the hierarchical approach to design is useful for representing and describing a system following the principle of information hiding. He enunciated that “a programmer is most effective if shielded from, rather than exposed to, the details of system parts other than his own.” Parnas explained that it was ill-structured information distribution that made software systems dirty by involving almost invisible connections (coupling) between supposedly independent modules. Parnas believed that limitation of the information coupling among modules was the key to improve design quality in software system design.

An equivalent expression of information hiding in software engineering is abstraction, which describes how the common properties and shared information of a set of objects may be elicited and explicitly represented. As a consequence of abstraction, the uncommon properties and unshared information of software components are then hidden at lower level structures.

Abstraction and information hiding are the major methodologies for implementing modularization, functional decomposition, and stepwise refinement in structured programming. It is also the foundation of a number of modern software design and modeling techniques such as abstract data types, encapsulation, object technology, software components, and design patterns.

A pattern is highly reusable and coherent set of complex classes that are encapsulated to provide certain functions [1, 2, 9]. Object-oriented design patterns can be classified into the creational, structural, and behavioral patterns [2]. Design patterns can be formally described by classes as shown in Table 1. Classes in object-oriented methodologies are classified into the categories of system classes (SC) and user derived classes (DC). The latter can be further divided into abstract classes (AC) and concrete classes (CC). These classes, as well as methods (M) modeled in a class, are treated as derived types of the System Architecture Type (ST) in RTPA [10, 11].

<table>
<thead>
<tr>
<th>No</th>
<th>Type</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System class</td>
<td>SC</td>
<td>A class provided by the system</td>
</tr>
<tr>
<td>2</td>
<td>Derived class</td>
<td>DC</td>
<td>An abstract class defined by a user based on SC</td>
</tr>
<tr>
<td>2.1</td>
<td>Abstract class</td>
<td>AC</td>
<td>A class serves as a generalization and conceptual model, which can be inherited but can not be instantiated</td>
</tr>
<tr>
<td>2.2</td>
<td>Concrete class</td>
<td>CC</td>
<td>An ordinary class derived from AC</td>
</tr>
</tbody>
</table>

Table 1. Taxonomy of Class Types
The concept of classes can be formally described by RTPA as shown in Fig. 2. The interfaces of a class are the means of access for users of the class, which models a set of attributes and methods. The implementations are hidden behind the interfaces of a class, which realize detailed functions.

\[
\text{Class}^{\text{ST}} \triangleq \{ \text{Architecture} : \text{ST} \\
|| \text{StaticBehaviors} : \text{ST} \\
|| \text{DynamicBehaviors} : \text{ST} \}
\]

(2)

\[
\text{Class}^{\text{ST}}.\text{Architecture}^{\text{ST}} \triangleq \{ <\text{Interfaces} : \text{ST}> \\
|| <\text{Implementations} : \text{ST}> \}
\]

(3)

\[
\text{Class}^{\text{ST}}.\text{Architecture}^{\text{ST}}.\text{Interfaces}^{\text{ST}} \triangleq \text{ClassID}^{\text{ST}} :: \{ <\text{Attributes} : \text{RT}> \\
|| <\text{Methods} : \text{ST}> \}
\]

(4)

Fig. 2 The abstract model of a generic class

The associations between classes can be classified into nine categories in object-oriented methodologies as summaries in Table 2. Table 2 also provides a set of formal descriptions of class associations in real-time process algebra (RTPA) [10, 11]. The associations of classes form a foundation to denote complicated relations between classes in software patterns.

As discussed earlier, pattern specification is based on class specifications. Using RTPA notations and methodology, a pattern is denoted by three parallel components known as the architecture, static and dynamic behaviors at the top level. Then, the architecture of the pattern is refined by a component logic model (CLM) [11]. The static and dynamic behaviors of the pattern are denoted by a set of collaborating processes.

The architecture of a generic pattern can be formally described by the four-level hierarchical model as shown in Fig. 3 using RTPA. The generic pattern model may be treated as a super meta pattern, which models any specific design pattern at four levels known as the interfaces, implementations, instantiations, and associations among the interfaces, implementations, and instantiations.

According to the four-level hierarchical model of patterns, the features of patterns lie in the hierarchical architectures as described by PatternST.ArchitectureST in Fig. 3. It is noteworthy that an ordinary class is modeled as a two-level structure with the class interfaces and implementations as shown in Fig. 2. However, the architectural model of a pattern is a four-level hierarchy featured with the extended refinement levels of instantiations and associations for deriving applications of the pattern.

| Table 2. Formal Semantics of OO Association Specification in RTPA |
|-------------------|---------------|------------------|
| **OO Associations** | **UML** | **RTPA** |
| Inheritance (Inh) | Inh $\triangleq Q : P$ | $Q \xrightarrow{\text{Inh}} P$ |
| Multiple inheritance (MInh) | $Q \xrightarrow{\text{MInh}} P_i$ |
| Delegation (Del) | Del $\triangleq Q \xrightarrow{\text{Del}} P$ |
| Aggregation (Agg) | Agg $\triangleq R_i P_i : Q_i$, $P_i \xrightarrow{\text{Agg}} Q_i$ |
| Composition (Com) | Com $\triangleq R_i P_i M_i : Q_i$, $P \xrightarrow{\text{Com}} Q_i$ |
| Generalization (Gen) | Gen $\triangleq R_i Q_i : P$, $Q_i \xrightarrow{\text{Gen}} P$ |
| Instantiation (Ins) | Ins $\triangleq Q : P$ |
| Dependency (Dep) | Dep $\triangleq R_i Q_i : P$ |
| Abstract/Concrete class (AC_CC) | $\text{Interface} \xrightarrow{\text{AC_CC}}$, $\text{Interface} \xrightarrow{\text{AC_CC}}$ |

The interface of a pattern, PatternST.ArchitectureST.InterfacesST, isolates users of the pattern from its internal implementation. Users may only access the pattern via its interface. This mechanism enables the implementation of the pattern independent of its users. Whenever the internal implementation needs to be changed, it is transparent to the users of the pattern as long as the interface keeps the
Since a design pattern is a highly reusable design of a software object, the implementation of a pattern, \( \text{Pattern}_{ST}.\text{Architecture}_{ST}.\text{Implementation}_{ST} \), will be kept at a generic concrete class. While the detailed and application specific functions related to users' requirements, which are captured via the interface, will be implemented at the lower-level concrete classes known as the instantiations \( \text{Pattern}_{ST}.\text{Architecture}_{ST}.\text{Instantiations}_{ST} \) at run-time.

The fourth component in the generic pattern hierarchy is the internal associations, \( \text{Pattern}_{ST}.\text{Architecture}_{ST}.\text{Association}_{ST} \), which is used to model the interrelationships among the rest three-level abstractions of classes and interfaces within the pattern.

The formal model of generic design patterns can be used as a formula to derive specific descriptions of any software design pattern. A case study will be provided in the following section, which shows that all typical and classical design patterns fit the generic pattern models of RTPA as developed in this section. It is noteworthy that a pattern is a generic model of reusable design notions. Specific behaviors in an executable instance are dependent on run-time information provided by users of the pattern.

5. A FORMAL MODEL OF THE BUILDER DESIGN PATTERN

The builder pattern is one of the important creational patterns proposed by Gamma and his colleagues [2]. The architecture of the builder pattern is shown in Fig. 4. This pattern is designed to separate the complex object construction process from its final divestible representations. The benefit of this treatment is that complicate construction processes may be reused to produce various object representations.

Fig. 4 The Structure of the Builder pattern

A formal description of the design model of the Builder pattern can be described on the basis of Figs. 3 and 4. As shown in Fig. 4, the \( \text{Director}_{AC} \) is an abstract class that serves as an interface of the builder pattern. The \( \text{Builder}_{AC} \) is an abstract class that represents the common and generic functions of the pattern, which may be implemented by a concrete class at lower levels. The \( \text{ConcreteBuilder}_{CC} \) is an concrete class that implements the conceptual model \( \text{Builder}_{AC} \). Note that the implementation of the \( \text{ConcreteBuilder}_{CC} \) is still a generic model, which will be completely implemented by the \( \text{Product}_{CC} \) at run-time.
when a user of the pattern provides further details of instances via the interface of the pattern.

Using Eqs. 5 through 10 as generic design formulae, the architecture of the BuilderPattern$^ST$ can be accurately and rigorously specified and step-wisely refined.

According to Eq. 5, the top-level architecture of the BuilderPattern$^ST$ can be specified as shown in Eq. 11:

$$\text{BuilderPattern}^ST \triangleq \{ \begin{array}{l} \text{Architecture} : ST \\ \| \text{StaticBehaviors} : ST \\ \| \text{DynamicBehaviors} : ST \end{array} \}$$ (11)

Using Eq. 6, the architecture of the pattern, BuilderPattern$^ST$.Architecture$^ST$, can be specified as follows:

$$\text{BuilderPattern}^ST.\text{Architecture}^ST \triangleq \{ \begin{array}{l} \text{<Interface>} \\ \| \text{<Implementation>} \\ \| \text{<Instantiation>} \\ \| \text{<Association>} \end{array} \}$$ (12)

Applying Eq. 7, the interfaces of the pattern, BuilderPattern$^ST$.Architecture$^ST$.Interface$^ST$, can be specified as follows:

$$\text{BuilderPattern}^ST.\text{Architecture}^ST.\text{Interface}^ST \triangleq \begin{array}{l} \text{Builder}^ST : \\ \{ \begin{array}{l} R \{ \text{Attributes}(iN) : RT \} \\ \| \text{<Builder} : \text{AC} > \\ \| \text{<Director} : \text{AC} > \end{array} \} \end{array}$$ (13)

Then, the implementations of the pattern, BuilderPattern$^ST$.Architecture$^ST$.Implementation$^ST$, can be specified as follows according to Eq. 9:

$$\text{BuilderPattern}^ST.\text{Architecture}^ST.\text{Implementation}^ST \triangleq \begin{array}{l} \{ \begin{array}{l} \text{<ConcreteBuilder} : \text{CC} > \\ \end{array} \end{array} \}$$ (14)

Applying Eq. 9, the instantiations of the pattern, BuilderPattern$^ST$.Architecture$^ST$.Instantiation$^ST$, can be specified as follows:

$$\text{BuilderPattern}^ST.\text{Architecture}^ST.\text{Instantiation}^ST \triangleq \begin{array}{l} \{ \begin{array}{l} \text{<Product} : \text{CC} > \\ \end{array} \end{array} \}$$ (15)

The relations between the components described above, the associations of the pattern, BuilderPattern$^ST$.Architecture$^ST$.Associations$^ST$, can be formally described on the basis of Eq. 9 as follows:

$$\text{BuilderPattern}^ST.\text{Architecture}^ST.\text{Associations}^ST \triangleq \begin{array}{l} \{ \begin{array}{l} \text{<Interface Class Associations>} \\ \| \text{<Builder} : \text{AC} > \\ \| \text{<Director} : \text{AC} > \\ \end{array} \} \end{array}$$ (16)

Based on the formal model of the architecture of the Builder pattern, its behaviors can be rigorously described. The specification of behaviors of a pattern at the top level is via the dynamic behavioral technique of RTPA [11]. Detailed static behaviors are described for each of the classes encompassed in the pattern separately. Note that the behaviors of a pattern are application specific. The conventional patterns as shown in Fig. 4 are inadequate to denote what the behaviors and functions are, because it only provides a rough conceptual model. Examples of formal description of static and dynamic behaviors of software patterns using RTPA may be referred to [9, 10, 11, 13].

6. CONCLUSIONS

Abstraction has been identified as a cognitive foundation for engineering designs and system modeling. This paper has presents a formal treatment of system design and its cognitive process. This paper has modeled the hierarchy of abstraction and the descriptivity of abstract means at five levels as described in the hierarchical abstraction model of system descriptivity (HAMSD) model. Based on HAMSD, generic design methodologies are sought and the differences of abstract and concrete system designs are contrasted. Applications of abstraction and formal inferences in dealing with complicated design problems in software engineering, such as object-oriented systems and design patterns, are demonstrated using real-time process algebra (RTPA).

ACKNOWLEDGEMENTS

The author would like to acknowledge the Natural Science and Engineering Council of Canada (NSERC) for its support to this work. The author would like to thank reviewers and colleagues for their valuable comments and suggestions to this work.
REFERENCES


