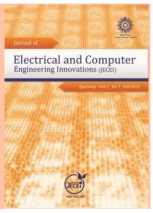
SRTTU

Journal of Electrical and Computer Engineering Innovations

JECEI, Vol. 4, No. 1, 2016

Regular Paper



Declarative Semantics in Object-Oriented Software Development - A Taxonomy and Survey

Hassan Rashidi[1,*]

[1] Department of Statistics, Mathematics, and Computer Science, Allameh Tabataba'i University, Tehran, Iran.

[*]Corresponding Author's Information: hrashi@atu.ac.ir

ARTICLE INFO

ARTICLE HISTORY:
Received 19 September 2016
Revised 29 October 2016
Accepted 30 October 2016

KEYWORDS:
Taxonomy
Object
Semantics
Object-Oriented
Software Engineering

ABSTRACT

One of the modern paradigms to develop an application is object oriented analysis and design. In this paradigm, there are several objects and each object plays some specific roles in applications. In an application, we must distinguish between procedural semantics and declarative semantics for their implementation in a specific programming language. For the procedural semantics, we can write a set of instructions that must be executed sequentially. The declarative semantics declare a set of facts and rules. They do not specify the sequence of steps for doing the processing. In this paper, we present four taxonomies for the rules in object-oriented paradigm and discuss how the paradigm can be extended to support declarative semantic of applications. Then, the rules in the taxonomies are evaluated in four case studies. After that, an approach is recommended for finding and implementation of declarative semantics, based on some practical experience obtained from the evaluation.

1. Introduction

There are two kinds of semantics in developing an object-oriented application [1]. The first one, known as procedural semantic, is related to the statements that must be executed sequentially. The second one, known as declarative/nonprocedural semantic, refers to the statements that must be executed in several parts of the application, possibly in parallel.

Some applications have many natural requirements that are given in a declarative manner (see [2], [3], [4], [5] and [6]). For example, suppose: (a) applications that monitor a physical systems in a factory or refinery, (b) applications that apply business policies or engineering guidelines in a enterprise resource planning, and (c) software development tools to handle many exceptional conditions. When these requirements occur, handling of the declarative semantics (rules and facts) is left to the analysts/developers. One of the most difficult tasks for developers is transforming declarative semantic into the procedural one. It is very natural for developers to incorporate these declarative semantics across the methods of various classes. However, when a declarative statement affects multiple classes, it must be programmed in several places of the methods across the classes. This is not a good practice because there is a transformation of declarative semantics into procedural semantic and it creates hidden coupling between the methods in the classes. The first reason makes the model less readily understandable and violates the goal of modeling reality the way the domain experts see it. The second reason makes maintaining and changing the model very difficult.

During their maintenance period, applications tend to grow and take more budgets ([5] and [7]). As they grow, developers encounter more situations in which declarative statements have been distributed across several methods. These situations make the maintainability of the model too difficult. For example, maintaining an invariant involving two objects may require that similar but not identical tests be inserted into a variety of places within the code. This leads to errors of omission and logic by analysts, designers, and programmers as the application is extended.

Because the invariant is not in one place, it is never explicitly stated. Unstated assumptions make modifications of code difficult and error-prone. We need both a method and a mechanism to handle declarative statements.

One of the main sources of complexity in software programs is the constant need to check whether data passed to a processing element (method in a class) satisfy the requirements for correct processing or not. Sometimes, it is necessary to perform these checks in the method itself and sometimes in its clients. Unless class designers formally agree on a precise distribution of responsibilities, the checks end up not being done at all, a very unsafe situation or, out of concern for safety, being done several times. Redundant checking may seem harmless, but it is not. It hampers efficiency, of course; but even more important is the conceptual pollution that it brings to software systems. Complexity is probably the single most important hostile of software quality. The distribution of redundant checks all over a software system destroys the conceptual simplicity of the system, increases the risk of error, and hampers such qualities as extensibility, understandability, and maintainability.

The main motivation of this paper is to survey on declarative semantics and their taxonomies. The structure of remaining sections is as follows. In Section 2, the literature review and taxonomies on rules are presented. In Section 3, the experiences of applying the approaches to four case studies are presented. In Section 4, several approaches related to rules/facts in the object-oriented paradigm are recommended. Finally, Section 5 is considered to the summary and conclusion.

2. Literature Review and Taxonomies

One of the major challenges in the object-oriented development and transforming the legacy systems into object-oriented one is how identify declarative semantics (rules and facts) (to identify objects see [2], [8], [9], [10], [11] and [12]). To do this, many developers use their experiences. The rule-based model in an artificial intelligence system is based on a control view of reality. In this model, the software has an inference engine that executes a set of rules (if-then statements). In theory, the sequence in which the rules were executed was not material. However, in practice, most of developers were not able to find rules that were truly decoupled. Because there was no structure to organize the rules, the software has also very poor cohesion. Furthermore, rule-based systems did not help developers to manage the data and did not support procedural concepts.

There is a collection of mechanisms integrated into a consistent model that helps manage the complexity of the procedural aspects in applications. The concepts of abstraction, encapsulation, inheritance, relationship, and polymorphism in object-oriented methods that support the design and implementation of sophisticated procedural applications are identified between the classes/objects (see [2], [13], [14], [15], [16] and [17]). Moreover, there are many popular design modeling processes and guidelines such as GRASP [18] and ICONIX [19] for assigning responsibility to classes and objects in object-oriented design.

Most object-oriented developments are presently based on the assumption that all aspects of the application/system will be modeled within the procedural paradigm (see [14], [20], [21], [22], [23], [24] and [25]). However, there are aspects of an application/system that are nonprocedural (declarative) and are better modeled using other mechanisms. Declarative semantics are employed for a variety of purposes, such as enforcing invariants, in a domain model, auditing complex data structures, monitoring the state of a state machine, or checking constraints while a user inputs data [26]. Some declarative semantics are best that captured directly in the classical object-oriented paradigm, while others are better captured via other mechanisms [27].

One kind of rule that is better captured via another mechanism is the data-driven rule [5]. This is due to the property that it requires a mechanism to act as a "monitor" of the model while it observes changes to an object's attributes and reacts when a condition is satisfied. The data-driven mechanism is well-suited to handling rules that monitor things. It supports the situation-action' directive without complicating an application's procedural logic. It also satisfies two very important goals of the object-oriented paradigm: (a) the model should be built to reflect the way the domain experts see reality, and (b) whenever possible, the code for the application should be generated from a model that is easy for the domain experts and the end users to understand. To satisfy these two goals, declarative statements (including rules) need to be rigorous. They must be understandable to the end user so that they can verify that the rules correctly represent business policies and desired application/ system behavior. Thus, declarative statements, including rules, should be written in structured English.

In 1990, the language R++ was developed. It is a rule-based programming language based on C++. The R++ extension permits rules to be defined as members of C++ classes. The programming system of the invention takes the classes with rules defined using R++ and generates C++ code from them in which the code required for the rules is implemented completely as C++ data members and functions of the classes involved in the rules.

A limited number of works have focused on the taxonomy of rules. James Martin and James Odel (1992) have constructed the following classification scheme for rules [28]. Their scheme describes three types of rules. The first one is *Integrity rules* that state something must always be true (e.g., a value for an attribute must he in the integer range from 1 to 5). The second one is *Derivation rules* that state how a value or set of values is calculated (e.g., Tax Withheld=Federal Income Tax + State Income Tax). The third one is *Behavior rules* that describe the dynamic aspects of behavior, such as what conditions must be true for performing an action (e.g., when the door is open, the light in the oven is turned on).

Lee and Tepfenhart (2005) identified seven categories for the rules [29]: (a) Data integrity rules; (b) Relationship integrity rules; (c) Derivation rules; (d) Service precondition rules ; (e) Service post condition rules; (f) Action trigger rules; (g) Data trigger rules; (h) Control condition rules.

Rashidi (2015) reviewed the relationships among objects in object-oriented software development and made five taxonomies for their properties [30]. Mainly, the relationships are three basic types. This paper presents five taxonomies for properties of the generalization/specialization, association and aggregation relationships. The first taxonomy is based on temporal view and the second one is based on structure. The third taxonomy relies on behavioral view and the fourth one is specified on mathematical view. Finally, the fifth taxonomy is related to the interfaces between objects. Moreover, in this paper the relationships are evaluated in a case study and then several recommendations are proposed. The main conclusion is that the relationships must capture some concepts that applies to the problem domain or some sub-domains.

One the major gaps and research needs is to have an overview and taxonomy on rules in object-oriented software development. According to Merriam-Webster [31], taxonomy is the study of the general principles of scientific classification, and is especially the orderly classification of items according to their presumed natural relationships. The major differences between rules in the software, in general, depend on the integrity, service and triggers views, and in particular derivation view. There are, therefore, four taxonomies to categorize the rules in object-oriented development. These taxonomies are described in the following sub-sections.

A. The First Taxonomy: Integrity-view

The first taxonomy for rules is based on Integrity of software model. The Integrity, by definition in the context of computer systems, refers to methods of ensuring that data and their relationships are real, accurate and safeguarded from unacceptable modification. Hence, we have two types of rules in this taxonomy: Data Integrity Rule and Relationship Integrity Rule. These are described as follows:

- **Data Integrity Rules (DIR)**: They state that something must be true about an attribute(s).
- **Relationship Integrity Rules (RIR):** They state that something must be true about a relationship.

B. The Second Taxonomy: Computation view

The second taxonomy for rules is based on computation view. In this taxonomy, we have two types of rule: Calculation rules and Heuristic rules. These rules are described as follows:

- **Calculation Rules (CR):** They state how a value or a set of values is computed. For example, in the education systems of each university, semester/session grade-point average and cumulative grade-point average are calculated repeatedly to represent numerically a student's quality based on his/her courses marks.
- **Heuristic Rules (HR):** They are usually related to search capabilities or they are common-sense rules that help to find a good enough solution for an optimization problem. For example, the throughput of system must be maximized.

C. The Third Taxonomy: Service-view

There are two different rules in this view: Service Precondition rules and Service Post Condition rules. These rules are described as follows:

- **Service Precondition Rules (SPRER):** They state that something must be true before a service be performed.
- **Service Post Condition Rules (SPOSR):** They state that something must be true after a service is performed.

D. The Forth Taxonomy: Trigger-view

The modern approach for implementation of software is to use the Data Driven software. In this approach, the software constantly monitors attributes and reacts automatically in response to changes in monitored objects when appropriate. The action portion of the code sits apart from the routine procedural code and is automatically triggered by relevant changes in the objects that the rule monitors. This relieves the analyst/developer from designing and programming explicit control for the data-driven rule. The trigger rules in the modern software can be classified as follows:

- **Action Trigger Rules (ATR):** They define the causal relationship between events and actions.
- **Data Trigger Rules (DTR):** They define the causal relationship between an attribute's condition and an action.

- **Control Condition Rules (CCR):** They handle situations in which multiple triggers are involved in the rule.

Several examples for each rule in this taxonomy are given in the next section.

3. Practical Experience and Guidelines

In order to evaluate the declarative semantic (rules and facts) mentioned in Section 2, we did four case studies, including a couple of general systems and a couple of particular systems. These case studies are described in the following:

- **ATM System:** This system was a simple ATM in which we expected to see use cases covering the principal functions such as withdraw cash, display balance, print statement, change PIN and deposit cash. The use case description had to be described the actors involved, the inputs and outputs, normal operation and exceptions. More details on this application are given in [5] and [17]. The class diagram for ATM system is depicted in Figure 1.
- **Control Command Police System (CCPS):** A mini-requirement for CCPS is briefly described in [32] and then the system is expanded in [15]. This police service system must respond as quickly as possible to many reported incidents. Its main objectives are to ensure that incidents are logged and routed to the most appropriate police vehicle. The full specification of the system and its implementation are given in [15]. Due to its fertility for reusability in both application and system software, we selected CCPS in our study whose its class diagram is depicted in Figure 2. In this class diagram, there are many classes. The main classes, here, are 'Incident', 'Police Staff', 'Police Vehicle', 'Police Officer', 'Director', 'Route Manager', 'Incident Waiting List', 'Response' and 'GPS Receiver'. We show the attributes and methods for only a couple of classes, namely, the 'incident' and 'response' class. These limitations made a more clear and informative picture. If we had shown the attributes and methods for all classes in this diagram, we would have a messy picture.
- **Voicemail System:** This system was a voice mail system consists of a speaker, a keypad, and a microphone. We model the operation of an embedded software system for a voicemail system included in a landline phone. This had to display the number of recorded messages on an LED display and should allow the user to dial-in and listen to the recorded messages. More details on this application are given in [5]. The class diagram for this system is depicted in Figure 3.
- **Firm Planning System:** In this system, a time series data including balance sheet, profit and loss account, financial ratios, production lines information and others variables relating to personnel, etc. of a firm (company) are available and must be stored in a database. An economic expert helps to estimate several equations to make a model among the time series data. The system must be able to accept several exogenous variables that are imposed from outside the system. The system uses the model to predict the endogenous variables in the coming years according to the equations subject to the exogenous variables. More detail on this system are given in [33]. The class diagram for this system is depicted in Figure 4.

Note that the mechanism on which artificial intelligence systems were built was primarily an inference engine. An inference engine processes a collection of facts and rules to make deductions using logical inference. The rules of an inference engine processes are called production rules. Most of developers understand declarative semantics from this perspective. However, the rules identified here were not production rules. They were rules linked with the object-oriented model to provide a meaningful and useful model for implementation.

Although declarative semantic are different from procedural semantic, identifying them in a requirements document is relatively simple. Its simplicity refer to while procedural semantic are always part of a specified sequence (e.g., a procedure, an activity, or a task), a declarative semantic are stands alone. A declarative semantic is independent of any sequence of other statements. It declares a factor for a rule. Several samples for each rule in the taxonomies of Section 2 are given in Table 1.

The numbers of rules identified in the four systems are put in Table 2. The number shown in the parentheses in front of the name of each system is according to the number of classes in the class diagrams (see Figures 1 to 4).

At the first glance from Table 3, we can get the following observations:

- **Observation-1:** The numbers of rules identified for the control command police and voicemail systems have the largest and smallest values, respective, among systems. It shows that these systems are the biggest and smallest ones, respectively, in our case studies (See Figure 3 and Figure 2).
- **Observation-2:** The numbers of rules in kind of Heuristic Rules (HR) are the smallest one in the systems. This observation shows there are few rules related to optimization problems during the object-oriented software development. In the Control Command Police system, the method 'Shortest Distance' of the class 'Route Planner' must choose the closest vehicle to address of the incident for its handling (see Figure 2). In the Firm Planning

system, we have a class for optimization problems (see the class 'Optimization' in Figure 4).

- **Observation-3:** The number of rules in kind of Relationship Integrity Rules (RIR) in two systems, the control command police and firm planning systems, is almost the same. It shows we have considered the multiple associations among objects during the object-oriented software development.

The number of rules identified in the systems is not very convenient to make any judgment because they are absolute values. Hence, we decided to calculate the percentage of the number of rules identified. The result of the calculations in the four systems is depicted in Figure 5.

TABLE 1
SOME SAMPLES OF THE RULES IN THE SYSTEMS

Taxonomy	Type of Rules	Sample for the systems
1st Taxonomy	Data Integrity Rules (DIR)	In the Voice Mail System, the capacity of mailbox must be in the integral range from 100 to 500MB.
	Relationship Integrity Rules (RIR)	In the Control Command Police System, the dispatcher may not supervise more than ten police officers.
2nd Taxonomy	Calculation Rules (CR)	In the Firm Planning System, we have the equation TA = CA + FA where TA, CA and FA are Total Assets, Current Assets and Fixed Assets, respectively.
	Heuristic Rules (HR)	In the Control Command Police System, the best strategy is to send the closest vehicle to address of an incident for its handling.
3rd Taxonomy	Service Precondition Rules (SPRER)	In the ATM System, amount of withdraw must be less than or equal the balance for the account.
	Service Post Condition Rules (SPOSR)	In the ATM System, the balance for the account will be decreased by the amount of withdraw
4th Taxonomy	Action Trigger Rules (ATR)	In the Control Command Police System, when an event is reported, then prepare the requirements immediately
	Data Trigger Rules (DTR)	In the Firm Planning System, when the CASH is below the minimum required then make a loan from a bank
	Control Condition Rules (CCR)	In the Control Command Police System, If an event has been handled and the report prepared then the event is closed

TABLE 2
THE NUMBER OF RULES IDENTIFIED IN THE SYSTEMS

Taxonomy	Rules	ATM (16)	Voicemail (11)	Control Command Police (15)	Firm Planning (14)
1st Taxonomy	DIR	5	7	11	3
	RIR	5	6	10	10
2nd Taxonomy	CR	4	8	5	15
	HR	**1**	**1**	**2**	**5**
3rd Taxonomy	SPRER	5	8	12	6
	SPOSR	7	4	7	5
4th Taxonomy	ATR	6	4	6	3
	DTR	6	6	6	4
	CCR	6	4	6	3
SUM		45	48	65	54

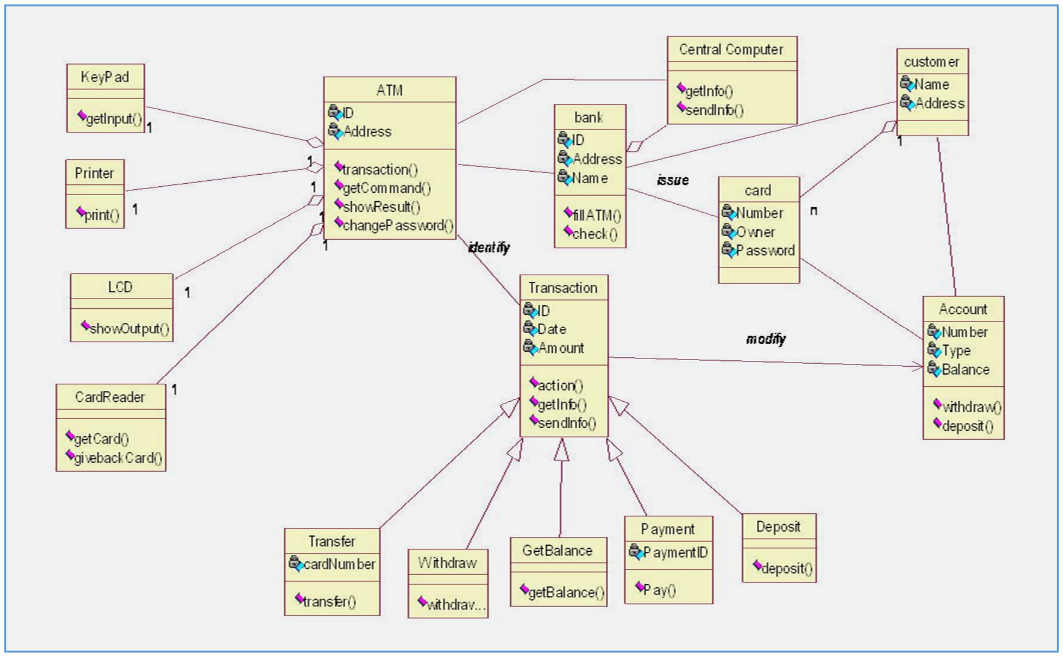


Figure 1: The Class Diagram of the ATM System.

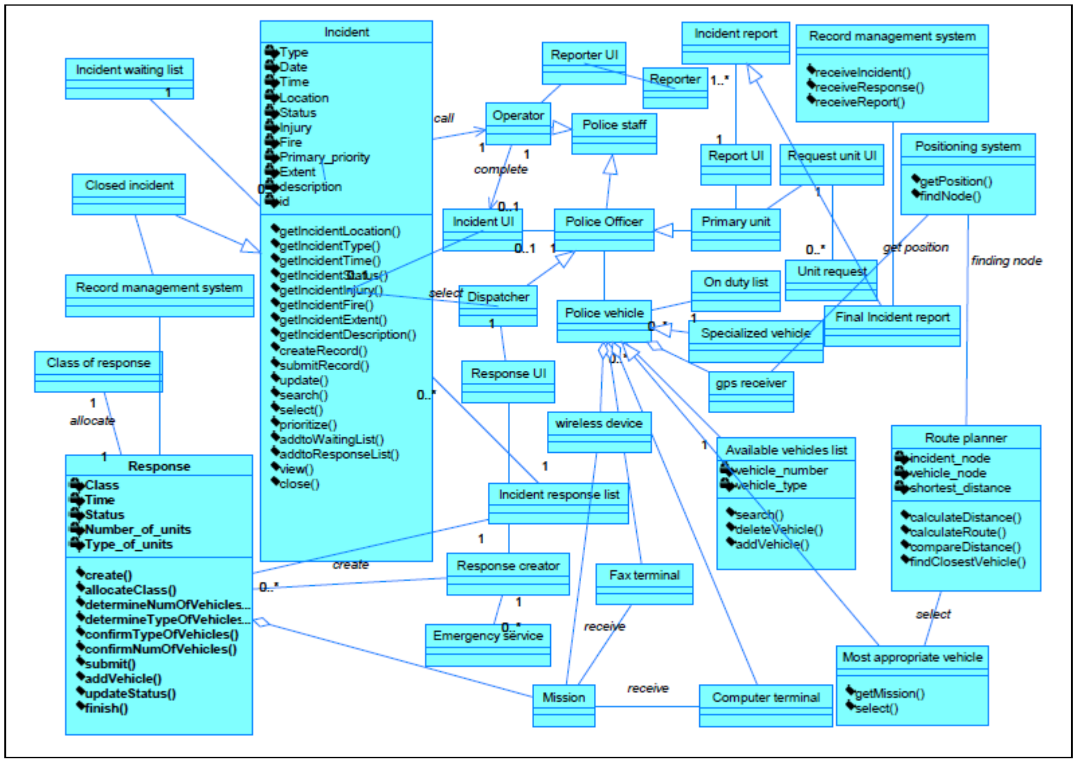


Figure 2: The Class Diagram of the Control Command Police System [34].

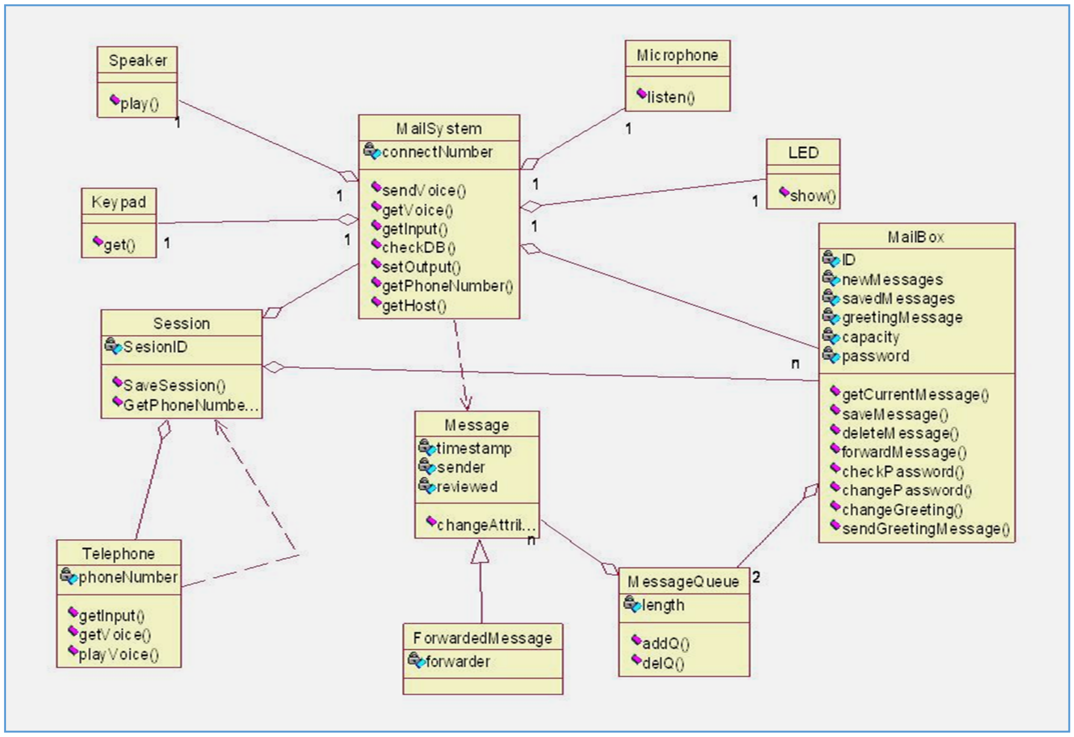


Figure 3: The Class Diagram of the Voice Mail System.

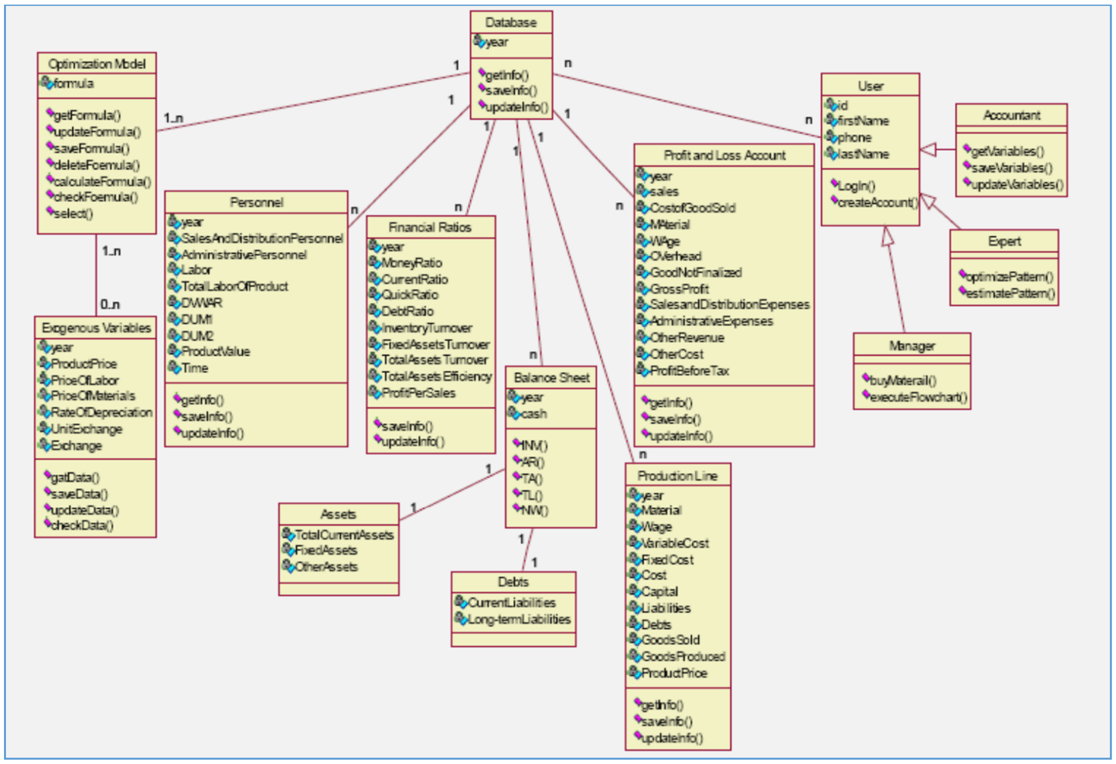


Figure 4: The Class Diagram of the Firm Planning system.

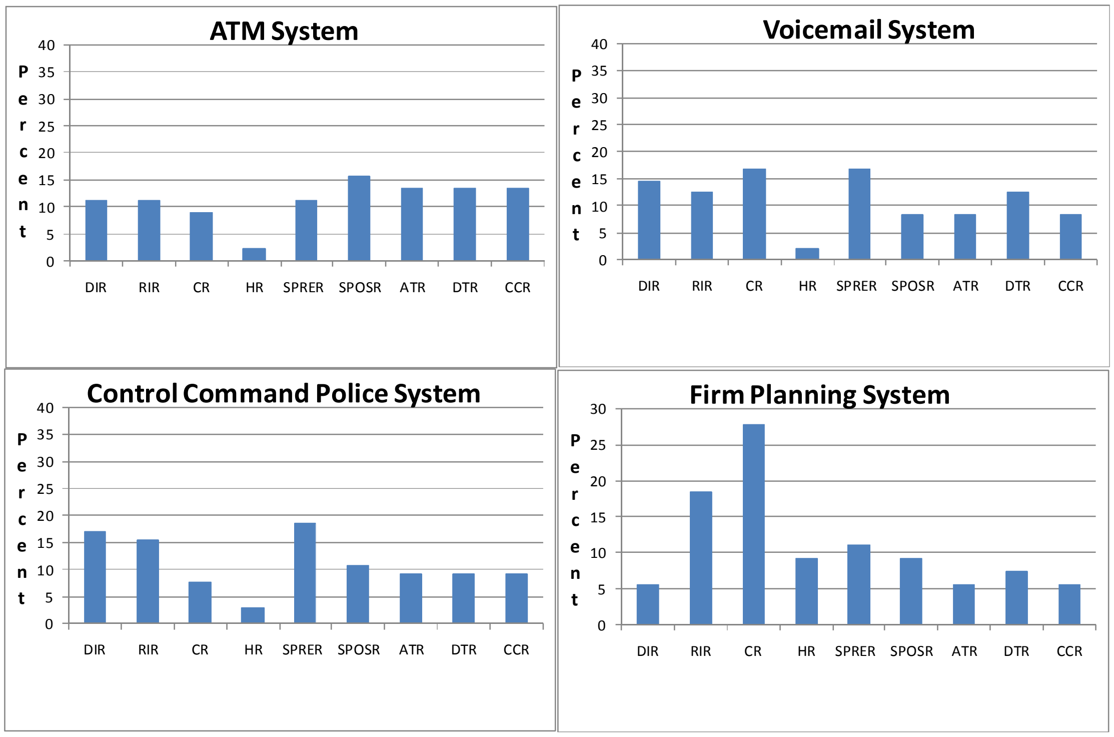


Figure 5: Percentages of the rules identified in the systems.

From Figure 5, we can get the following observations:

- **Observation-4:** The number of Calculation Rules (CR) identified in the firm planning system is the maximum. Because we have many mathematical equations in the firm planning system. These equations are around the item in the balance sheet, profit and loss accounts, the production lines and the relations around the personnel information.
- **Observation-5:** The number of Heuristic Rules (HR) identified in the systems is the minimum value. There exist five heuristic rules in the firm planning system that are derived for the optimization problems in this system.
- **Observation-6:** The number of rules in Service Post Condition Rule (SPOSR) in the control command police and ATM systems is greater than other rules.

The average percentages of the number of rules identified in the four systems are depicted in Figure 6. From this figure, we can get the following observations:

- **Observation-7:** The average percentage of the number of different kinds of rules identified in ATM system has the minimum variance. It varies between 2 and 16 and shows that ATM is a general system.
- **Observation-8:** The average percentage of the number of different kinds of rules identified in firm planning system has the maximum variance because this is a particular system. It varies between 5 and 27. It seems some other types of rules such as control condition and post service condition rules are considered in the computation rules.
- **Observation-9:** The average percentage of the number of Service Precondition (SPERR) and Service Post Condition (SPOSR) rules identified in the four systems are almost the same. It shows the Service Precondition (SPERR) rules are as important as Service Post Condition (SPOSR) rules in our case studies.

A. Guidelines to Identify and Specify declarative semantic

In our experience, when a requirement is written as a declarative statement, the best practice is to specify it as a rule. A technique that captures rules explicitly and makes them easy to read is structured English. For example, in the control command police system, the rules can be used to: (a) enforce something that should always be true (invariants) like "there is only one dispatcher", (b) detect things that should never be true (constraint violations) like "each event must not be handled more than once", (c) maintain the integrity of the domain model like "each police vehicle has a number", (d) monitor for and react to important events like "traffic incident for which it is necessary to send ambulance and vehicles with specific equipment", (e) express domain knowledge such as business policies, engineering rules, and situation-action heuristics like "the best strategy is to send the closest vehicle to address of an incident for its handling", (f) specify an operation (function) that would have to be used in many methods like "we must ensure that each incident is logged once it is reported", and (g) exploit the data-driven or event-driven nature of rules like "the system must respond as quickly as possible to reported incidents".

To sum up our guidelines, it is noted that rules usually capture information about how the business should operate. Rules encapsulate business knowledge. The most common statements in declarative semantics that indicate a rule are in Table 3. In these constructs, a condition is a Boolean expression, an event is a condition that can be detected by an object, constraints are a set of restrictions, and an action is an invocation of a procedural statement.

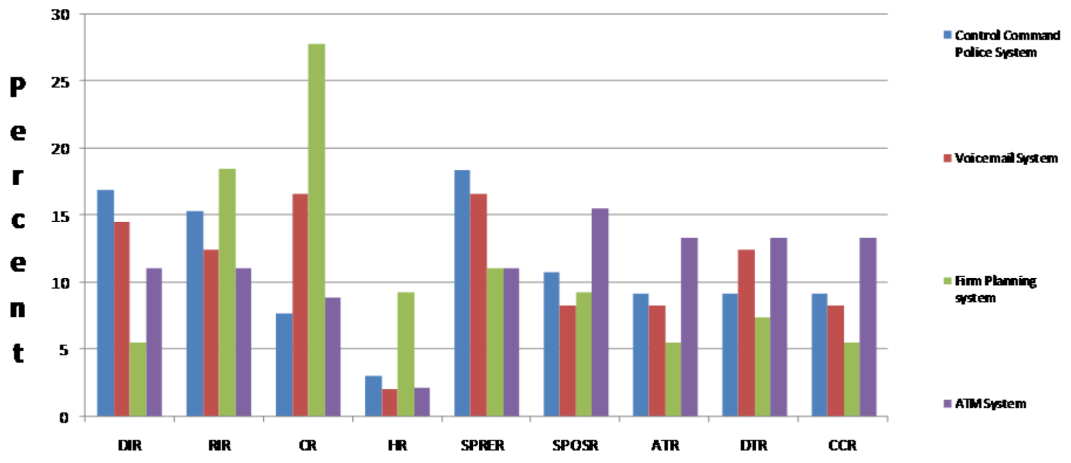


Figure 6: Average Percentages of the Rules identified in the four systems.

TABLE 3
VARIOUS KINDS OF RULES IN ANALYSIS DOCUMENTS

Kind of Rule	Common statements
Data Integrity Rule (DIR)	• IT MUST ALWAYS BE THAT statement of fact • IT MUST ALWAYS BE THAT IF condition THEN action
Relationship Integrity Rule (RIR)	
Calculation Rule (CR)	• Fact usually presented by an equation • WHEN condition or event THEN action • IF condition or not, THEN action
Heuristic Rule (HR)	• IT MUST BE THAT statement of policy • THE BEST Solution is to MAXIMIZE/MINIMIZE the function subject to constraints
Service Precondition Rule (SPRER)	BEFORE service that is to be performed IT MUST BE THAT fact
Service Post Condition Rule (SPOSR)	AFTER service that has been performed IT MUST BE THAT fact
Action Trigger Rule (ATR)	• IF condition THEN action • WHEN event IF condition THEN action
Data Trigger Rule (DTR)	
Control Condition Rule (CCR)	

B. Guidelines for documenting declarative semantics Using UML

During the design phase, developers identify and adapt design patterns and frameworks to realize specific subsystems. They must refine and specify precisely the interfaces of classes using constraint languages such as UML's Object Constraint Language. Finally, they map the detailed object design model to source code and database schema. The guidelines for documenting and mapping the rules into the object-oriented concepts are as follows:

a) **Data integrity and data trigger rules**: The rules of data integrity and data triggers are mapped onto an attribute. Normally, they are checked every time the attribute changes value. This is best documented by creating a new stereotype, called "data trigger," which is used to capture the actions associated with the rule(s). Then, artificial associations are drawn between the classes that need data triggers and the data trigger class.
b) **Relationship integrity rule**: A relationship integrity rule is mapped onto a relationship. It normally affects the instantiation, deletion of, and addition to a relationship. This is documented as a constraint in UML.
c) **Calculation rule**: A calculation rule is documented as part of the method. However, there are situations in which it is implemented as a trigger. Care must be taken when we have a calculation rule.
d) **Heuristic Rule**: An optimization rule is mapped to a class/object when it is related to an optimization problem. Obviously, there are several methods along data to perform the optimize operations. At least, one of the methods must be public to make in the class.
e) **Service precondition rule**: A service precondition rule is mapped onto a service. As suggested by developers, the precondition is a requirement that should be guaranteed by the calling object. This needs to be captured as part of the entrance criteria. If there is an operation specification for the service, use the precondition section-of the operational specification to document this.
f) **Service post-condition rule**: A service post-condition rule is also mapped onto a service. If an operations specification is written for the service, use the post-conditions section to document this. The post-conditions must also be included in the method description. It is a rule that must be checked by the developer of this service. The service must guarantee that the post-condition is satisfied.
g) **Action trigger rule**: An action trigger rule is mapped onto a finite state machine. It is usually an event in a state transition diagram. This is documented as an event in UML.
h) **Control condition rule**: A control condition rule is mapped onto a finite state machine. It is usually a condition needed for a change of state. This is documented as a guard condition in UML.

C. Guidelines for implementing declarative semantic

Our experiences show that declarative statements are usually written at a higher level of abstraction than procedural statements. Implementing declarative statements using this mechanism frees the analyst, designer, and programmer from having to manage flow of control for these statements. In our experience, the best mechanism for implementation is a data-driven mechanism. This mechanism simplifies the task of maintaining model integrity in two important ways. Firstly, it enables invariants and constraints to be stated explicitly in a single place, rather than having them scattered in multiple places in methods. This makes the model and thus code easier to understand and modify. Secondly, because it is data-driven, invariants and constraints are reevaluated automatically whenever relevant changes are made to an object's attribute. This relieves the analyst/programmer of the burden of explicitly incorporating data integrity rules into their procedural logic. The application's procedural logic is no longer cluttered with code for maintaining model integrity.

The mapping guidelines given in Section 3.2 show that service precondition rules, service post-condition rules, control condition rules, action triggers, and calculation rules map very nicely into the classical object-oriented model. However, relationship integrity rules, data integrity rules, and data triggers are not well supported in the model. To handle these rules, a data-driven mechanism is necessary. There are two ways to supply a data-driven mechanism:

- Use the triggers in the database system: Using triggers in the database system is the classical way of handling data integrity and data trigger rules. Every time the database recognizes a change in data value, it triggers a routine written by the User. The appropriate rules are implemented in that routine: This is reasonably straightforward for simple data-driven rules, but it is a little more tricky for complex rules (such as relationship constraints).The data-driven mechanism is important because many of our declarative requirements are given to us in a manner. This gives us a way to capture reality as domain experts see it-without the need to transform a declarative requirement (or solution) into a purely procedural model.
- Use the language R++ that extends C++ to include rules. This language bridges the gap between

object-oriented procedural semantics and data-driven rules. C++ classes contain two kinds of members: data members and member functions. R++ extends the C++ class construct with a new kind of member, a 'rule'. This enables object-oriented applications to employ data-driven computation. As an extension to C++, R++ fits comfortably with C++ concepts and practices. R++ rules are relatively easy to learn; the syntax is similar and the behavior is much like a "reactive" member function. An R++ rule is syntactically defined as follows:

Rule *class-name: rule-name {condition => action}.*

The condition-action pair behaves like an if-then statement; **if** *condition* then ***actions***.

The action is automatically executed when the condition evaluates to true. The system monitors the data members appearing in the rule's condition, and when a data member changes its data values, it creates a trigger event. The trigger event causes the rule to reevaluate the condition, taking into account new /changed data. If the condition is satisfied, the rule is fires. When a rule fires, the action is executed. Within the condition, part of the rule existential quantifiers (all and exists) and logical operators *(and* and *or)* are supported. Existential qualifiers and logical operators are used -to form compound conditions. In addition, the language supports accessing related objects (thus their associated services) via a concept called *binding.* Service calls (function calls) are also supported in the condition.

4. Recommended Approach

In this section, the approach is recommended to identify and specify the declarative semantics in applications. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced. Facts can be expressed as rules very easily. They will be derivation rules. Ideally, we want a technique that captures rules explicitly in a manner that is easy to read and will generate correct codes. An inference engine may be used to implement a method in a class; however, we do not recommend this technique because the significant parts of software are dedicated to objects in which performance must be maximized.

The recommended approach is to systematically use preconditions, and then allow the service author to assume, when writing the method, that the corresponding precondition is satisfied. The aim is to permit a simple style of programming, favoring readability, maintainability, and other associated qualities. This notion applies to libraries and classes within an application more so than servers in a distributed computing system. A server that handles multiple clients cannot afford to assume that all clients are well behaved. It must be coded defensively so that a rogue client cannot crash it or corrupt any data stored within it. When declarative statements appear in the requirements document, the following steps are recommended:

- **Step 1**: Separate the declarative statements from the procedural statements.
- **Step 2**: Restate the declarative statements using structured English as rules, taking care that the rules are rigorous and implementable.
- **Step 3**: Map the rules onto the appropriate object oriented mechanism.
- **Step 4**: If data-driven rules are used, employ a data-driven mechanism to model these rules. We recommend R++ over using database triggers.

5. Summary and Conclusion

In this paper, we distinguished between procedural and declarative statements in programming languages. Most of developers have worked with procedural programming languages. These languages provide several constructs so that we can write a set of instructions that must be executed sequentially. The sequences vary depending on conditions test, and a set of instructions is executed repetitively. However, declarative languages declare a set of facts and rules. They do not specify the sequence of steps for doing the processing.

Declarative statements, or rules, are another natural form in which domain experts and end users state their requirements. Analysts and developers should accept declarative statements as a natural part of textual requirements. It follows that declarative statements should be captured within a model. To do this, developers must translate the textual declarative requirements into structured English to assure that they have rigorous and implementable requirements. After stating all of the declarative requirements in structured English, developers should map each declarative statement into a kind of rule.

In this paper, we maked four taxonomies for the rules. The first taxonomy is based on integrity view and the second one is based on computation operations. The third taxonomy relies on service view and the fourth one is specified on trigger view. The rule taxonomies allow it to be properly assigned to the appropriate object-oriented mechanism in the model Moreover, the rules in the taxonomies were evaluated in four case studies

For implementation, we suggested the data-driven mechanism. This mechanism supports rules triggered by a change in the value of an attribute. Historically, triggers in a database system were used to implement this mechanism. However, documenting database trigger functions and getting people to read the documentation were not easily accomplished. An alternative approach is to use R++ language, which provides the data-driven mechanism as an integral part of the language. This approach is highly desirable because we can see all the code in one place. As with any tool, data-driven rules are good for some tasks and not as effective for other tasks. We recommend them for: (a) Enforcing invariants; (b) Maintaining data integrity; (c) Maintaining relationship integrity; (d) Detecting constraint violations; (e) Stating business policies and engineering guidelines.

REFERENCES

[1] M. Langer, "Analysis and Design of Information Systems," 3rd ed., Springer-Verlag London Limited, 2008.

[2] P. Coad, E. Yourdon, Object-Oriented Analysis, Yourdon Press, 1991.

[3] S. H. Pfleeger, J. M. Atlee, "Software Engineering: Theory and Practice," 4th ed., Pearson, 2010.

[4] R. S. Pressman, "Software Engineering: A Practitioner's Approach," 8th ed., McGraw-Hill, 2015.

[5] Y. Sommerville, "Software Engineering," 10th ed., Pearson Education, 2016.

[6] L. A. Stein, H. Lieberman, and D. Ungar, "A shared view of sharing: The Treaty of Orlando, Object-Oriented Concepts, Databases, and Applications", W. Kim and F. H. Lechosky, Eds. New York: ACM Press, 1989.

[7] M. Asadi, H. Rashidi, "A Model for Object-Oriented Software Maintainability Measurement," *International Journal of Intelligent Systems and Applications*, pp. 60-66, 2016.

[8] G. Bavota, A. De. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, pp. 1616-1664, 2014.

[9] K. Beck, W. Cunningham, "A laboratory for teaching object oriented thinking," OOPSLA '89 Conference proceedings on Object-oriented programming systems, languages and applications, ACM SIGPLAN Notices, 1989.

[10] A. Cockburn, Writing Effective Use Cases (Draft 3), Addison Wesley Longman, 2000.

[11] M. Fokaefs, N. Tsantalis, E. Strouliaa, and A. Chatzigeorgioub, "Identification and Application Of Extract Class Refactoring In Object-Oriented Systems," *Journal of Systems and Software*, vol. 85, pp. 2241–2260, 2012.

[12] H. Rashidi, "Objects Identification in Object-Oriented Software Development - A Taxonomy and Survey on Techniques", Journal of Electrical and Computer Engineering Innovations, vol. 3(2), pp. 27-43, 2015.

[13] B. Bruegge, A. H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns, and Java, Pearson Prentice Hall, 2010.

[14] I. Jacobson, M. P. Christerson, and F. Overgaard, Object-Oriented Software Engineering- A Use Case Approach, Addison-Wesley, Wokingham, England, 1992.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, E. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1992.

[16] R. King, **My Cat Is Object-Oriented,** Object-Oriented Concepts, Databases and Applications, Addison Wesley, 1989.

[17] R. Wirfs-Brock, Designing Object-Oriented Software, Prentice-Hall, 1990.

[18] C. Larman, "Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development," 3rd ed., Prentice Hall, 2005.

[19] D. Rosenberg, M. Stephens, Use Case Driven Object Modeling with UML: Theory and Practice, Apress, 2007.

[20] G. Canforaa, A. Cimitilea, A. D. Luciaa, and G. A. D. Lucca, "Decomposing Legacy Systems into Objects: An Eclectic Approach," *Information and Software Technology*, vol. 43, pp. 401-412, 2001.

[21] M. Fowler, K. Scott, "UML Distilled A Brief Guide to The Standard Object Modeling Guide," 2nd ed., Addison Wesley Longman, Inc, 1999.

[22] N. Goldsein, J. Alger, Developing Object-Oriented Software for the Macintosh Analysis, Design, and Programming, Addison-Wesley, 1992.

[23] J. V. Gurp, J. Bosch, "Design, Implementation and Evolution of Object-Oriented Frameworks: Concepts and Guidelines," *Software—Practice and Experience*, vol. 31, pp. 277-300, 2001.

[24] I. Jacobson, G. Booch, The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.

[25] J. Rumbaugh, "Getting Started: Using Use Cases To Capture Requirements," *Object-Oriented Programming*, vol. 7(5), pp. 8-12, 1994.

[26] S. Schlaer, S. Melior, Object Lifecycles: Modeling the World in States, Yourdon Press, 1992.

[27] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1998.

[28] J. Martin, J. Odell, Object-Oriented Analysis and Design, Prentice-Hall, 1992.

[29] R. C. Lee, W. M. Tepfenhart, "UML and C++: A Practical Guide to Object-Oriented Development," 2nd ed., Pearson Prentice Hall, 2005.

[30] Z. Rashidi, "Properties of Relationships among objects in Object-Oriented Software Design," *International Journal of Programming Languages and Applications*, vol. 5(4), pp. 1-13, 2015.

[31] Merriam-Webster Online (2011), Dictionary and Thesaurus, from http:// www.merriam-webster.com

[32] K. S. Subhash, M. Navi, and B. Bhojane, "NLP based Object-Oriented Analysis and Design from Requirement Specification," *International Journal of Computer Applications*, vol. 47(21), 2012.

[33] H. Rashidi, Firm Planning, Using Computing Models, Eghtesad Farda Press (in Persian), 2014.

[34] H. Rashidi, "Software Engineering-A programming approach," 2nd ed., Allameh Tabataba'i University Press (in Persian), Iran, 2014.

BIOGRAPHIES

Hassan Rashidi is an Associate Professor in Department of Mathematics and Computer Science of Allameh Tabataba'i University. He received the B.Sc. degree in Computer Engineering and M.Sc. degree in Systems Engineering and Planning, both from the Isfahan University of Technology, Iran. He obtained Ph.D. from Computer Science and Electronic System Engineering department of University of Essex, UK. His research interests include software engineering, software testing, and scheduling algorithms. He has published many research papers in International conferences and Journals.