J. King Saud Univ., Vol. 13, *Comp. & Info. Sci.*, pp. 105-125 (A.H. 1421/2001)

A Framework for the Prototype-based Software Development Methodologies

Abad Shah
Department of Computer Science, College of Computer & Information Sciences
King Saud University, P.O.Box 51178, Riyadh 11543, Saudi Arabia

(Received 28 November 1999; accepted for publication 11 October 2000)

Abstract. In the object-oriented paradigm, two techniques, *class-based technique* and *prototype-based technique* are available for modeling the real-world objects. In this paper, we first study and analyze both object-modeling techniques, then using this study and analysis we identify a class of applications, and argue that the class-based methodologies that use the class-based technique as the object-modeling technique, are inappropriate to use for the development of this class of applications. Considering the requirements of the identified class of applications, we argue for need of a class of software development methodologies which are referred to as the *prototype-based methodologies*. In this paper, we also propose modifications in the classical Water-Fall life-cycle software development model, which make it consistent with the requirements of the prototype-based methodologies. The modified life-cycle model provides a framework and basic guidelines for proposing the prototype-based methodologies.

Keywords: object-oriented paradigm, class-based technique, prototype-based technique, object modeling, software development methodologies, framework

1. Introduction

In the literature, two techniques are available in the object-oriented paradigm for modeling the real world objects. The first technique is based on the mathematical concept of *set*, and the second technique is based on the *observation* of objects to acquire their knowledge and the acquired knowledge is used for modeling the objects [1, 2]. These two object-modeling techniques are referred to as the *class-based technique* and the *prototype-based technique*, respectively.

The class-based technique models an object by the two parameters, *structure* and *state* [3, 4]. The *structure* of an object provides the structural and behavioral capabilities to the object, and it is defined by a set of instance variables and methods

(or operations). The *state* of the object assigns data values to the instance variables, and the methods of the object that operate on the state. A set of objects that share the same structure is referred to as a *class* [4]. In this technique, classes of a system are organized as a directed acyclic graph (DAG) or simply by a graph, and this organization of the classes is referred to as a *class-lattice* [4]. The classes in a class-lattice hold *parent-child* relationships among the classes. A *child-class* (or *sub-class*) inherits the structure from its *parent-class (or super-class)*, and this knowledge sharing mechanism is referred to as the *inheritance mechanism* [5]. In the class-based technique, we refer to the structure of an object as the *knowledge* of the object.

The prototype-based technique defines an object by a single unit of knowledge which can be the structure, state, or a combination of both [1, 6]. An object is referred to as a *prototype* in this technique. Note that in this paper while referring the prototype-based we use the terms prototype and objects interchangeably. A prototype is defined by its *default knowledge*, and the knowledge is acquired by *observing* the prototype when it is created. The default knowledge can be the structure, state, or a combination of both parameters of the prototype [1, 2, 7]. A new object (or prototype) can be defined by sharing the knowledge of one or more existing objects and by defining additional knowledge in the newly defined object. The prototype-based technique is a class-less technique, which means that all objects of a system are placed at the same level without forming any structural organization (e.g., class-lattice) of the objects [1. 2].

The class-based technique and object-oriented paradigms have become synonymous among the computer professionals, and they consider every object-oriented language or system as a class-based language or system. The source of this misconception is due to an early programming language *Simula-67* [8] which used the class-based technique as an object-modeling technique in the programming language. Although, the prototype-based technique is considered more flexible, powerful, and simpler technique than the class-based technique but the prototype-based technique could not get the popularity that it deserved, and it had been ignored in the past [1, 2, 9]. Recently, the prototype-based technique is getting momentum, and languages and systems such as SELF programming language and temporal object system (TOS) [3, 6, 10, 11] have emerged, which are based on this technique.

Several object-oriented software development methodologies such as Object Modeling Technique (OMT), Booch methodology, Yourdon object-oriented methodology, Fusion Methodology and many more are reported in the literature [12 - 17]. These methodologies suggest different schemes for object-oriented analysis and design of a system. In other words, these methodologies suggest different processing steps in the analysis phase, design phase, and other phases of the classical Water-Fall life-cycle model (or simply the *classical life-cycle model*) during the development of a system [18]. A comprehensive comparison and study of some popular object-oriented

software development methodologies is available in [19, 20]. The common feature of these methodologies is that they use the class-based technique as the object-modeling technique, and we refer to these object-oriented methodologies as the *class-based methodologies.*

To the best of our knowledge, no software development methodology is reported in the literature, which uses the prototype-based technique as the object-modeling technique. Note that the methodologies which will use the prototype-based technique as the object-modeling technique, are referred to as the *prototype-based methodologies.* However, there exists a class of applications for which the prototype-based methodologies are more appropriate to use for their development than the class-based methodologies. The applications such as software development environment, Computer-Aided Construction (CAC), and the Web applications belong to this class of applications. One typical characteristic of the objects of this class of applications is that they frequently change their structure, state, or both. These frequent changes to the objects make a class-lattice complex and inefficient, and consequently make the class-based methodologies inappropriate to use for the development of this class of applications. Also, in some applications of this class such as Computer-Aided Construction (CAC), it is desirable to store, and later trace back the history of changes to a specific object [21]. The history of changes can be consulted in making maintenance decisions about the object. The class-based methodologies do not suggest any mechanism or guidelines during the system development for storing and later tracing back the history of changes to a specific object [3, 21, 22]. Therefore, we are convinced that there is a need of a new class of methodologies to properly develop this class of applications. We identify the characteristics of the class of applications later in Section 3.

Another common feature among the class-based methodologies is that they *follow* the classical life-cycle model [18]. Here the term *follow* means that the class-based methodologies develop a system using the general guidelines that are suggested by the classical life-cycle model. In our opinion, this follow of the prototype-based methodologies the classical life-cycle model without proper modifications will not permit the methodologies to exploit all features of the prototype-based technique, and the prototype-based methodologies will not be able to capture the crucial characteristics of the identified class of applications. For further support of our this claim, we give more reasons in the next paragraph.

The prototype-based technique does not differentiate between the two parameters (i.e., structure and state) of an object, and both parameters can be captured simultaneously as a prototype. The classical life-cycle model and the methodologies that follow this model, do not support the capturing the data parameter or the simultaneous capturing of both parameters of an object during the system development in any phase.

This deficiency of the classical life-cycle model influences the methodologies (such as the class-based methodologies) that follow it, and the model makes these methodologies unsuitable to use for the development of the identified class of applications. Therefore, it is necessary to modify the classical life-cycle model to overcome this deficiency and make it suitable for the prototype-based methodologies. The modified life-cycle model can provide a framework (or basic guidelines) for the proposing prototype-based methodologies.

The remainder of this paper is organized as follows. In Section 2, we study and analyze the class-based and prototype-based techniques, and give a comparison of both object-modeling techniques. Using this study and analysis, we identify the characteristics of a class of applications for which we argue that the prototype-based methodologies are appropriate to use for their development. In Section 3, we give the characteristics and identify a class of applications for which the prototype-based methodologies are appropriate to use for the development of this class of applications. The proposed modifications in the classical life-cycle model which can be used as a framework for the prototype-based methodologies, are given in Section 4. Finally, in Section 5, we give our concluding remarks and future directions.

2. Class-based and Prototype-based Techniques: Study and Analysis

In the previous section we briefly described the class-based and prototype-based techniques. In this section we give a comparative study of both object-modeling techniques. The comparative study is also summarized in Table 1. In the next section we use this study for identifying the characteristics of a class of application for which the prototype-based methodologies are appropriate to use for the development of this class of applications. This study also leads us in proposing the modifications to the classical life-cycle model.

2.1 Basis of the techniques

As mentioned earlier, the basis of the class-based technique uses the first knowledge representation technique that is based on the mathematical concept of *set* [2]. The set theory suggests that firstly a set be defined, then the set is instantiated. All instances (or members) of a set share a common definition (or properties). Similarly, in the class-based technique we first define a class (or the *structure*) by defining its instance variables and methods, then create instances of the class [1, 2]. All instances (or *states*) of a class share a common definition of the class, and instances are dependent on the definition of the class. The prototype-based technique uses the second knowledge representation technique, called the *observation*. An object is modeled by acquiring its default knowledge through observation [1, 2].

Table 1. A comparison of the class-based and prototype-based techniques

No	Features	Class-Based Technique	Prototype-Based Technique
1	Basis	Mathematical concept – set of knowledge representation	Knowledge representation through object observation
2	Object modeling parameters	(i.) defines an object by distinct parameters structure and state. (ii.) Object is not defined in an incremental fashion.	(i.) defines an object by a single parameter - prototype and does not differentiate between structure (or meta-data) and state (or data) of the object (ii.) Objects are defined in a pure incremental fashion.
3	Organization of objects of a system	Objects are organized into a hierarchical structure, called a class-lattice	Objects are not organized in any hierarchical structure - no class-lattice
4	Tracing of changes to a specific object	not possible	Possible, since each change to an object is stored in a separate prototype.
5	Knowledge sharing mechanism	Inheritance mechanism, and it is static mechanism.	Delegation mechanism, and it is dynamic mechanism.
6	Fixation of message-Passing pattern	Message-passing pattern is fixed at compile-time.	Message-passing pattern is fixed at run-time.
7	Retention control while message-passing	Control remains with the self class while the search goes to the next super-class.	Control is passed to the next prototype with the search delegation.
8	Flexibility and efficiency	(i.) In case of simple inheritance and single-parent delegation, both mechanisms are equally powerful (Stein, 1987) (ii.) Otherwise it is less flexible, and less powerful than the delegation mechanism. (iii.) Efficiency is predictable.	(i.) In case of simple inheritance and single parent delegation, both mechanisms are equally powerful. (ii.) More flexible and powerful than the Inheritance mechanism. (iii.) Efficiency is not predictable

2.2 Object parameters

The class-based technique models an object by two distinct parameters, and they are considered two different entities [2]. The data (or state) parameter of an object depends on the structure parameter of the object. It means that if the structure of a class suffers a change, then all states of the class are also affected by the change. During discussing the class-based technique, we refer to the structure of an object as the *knowledge* of the object.

An object is modeled as a single unit (or parameter) of knowledge in the prototype-based technique. The single unit can be the structure, state, or both. The property of not-differentiating between structure and state of an object of the technique empowers it to capture both parameters of an object simultaneously and in the same manner.

2.3 Organization of objects

In the class-based technique the object classes of a system are organized into a hierarchical structure - class-lattice. Level of a class in a class - lattice determines the abstraction level of the class. It means that a class at a higher level in a class - lattice is at a higher level of abstraction than a class at a lower level in the class - lattice. The parent -child relationship among classes results into the inheritance of knowledge (which is only the class's structure).

The prototype-based technique does not form any hierarchical structure among objects (or prototypes) of a system. The technique considers all objects of a system at the same level, and any object can dynamically share the knowledge of other objects of the system using the delegation mechanism which is explained later.

2.4 Tracing of changes to a specific object

After the creation of an object, the object can suffer different types of changes. The changes can occur to the structure, the state, or both parameters of the object. The class-based technique proposes *schema evolution management system* [4], and *version management system* [23] to capture and trace back the changes to the structure and state of an object, respectively. But as mentioned earlier, the technique does not suggest any solution to the situation when an object suffers a change to its both parameters simultaneously. In this situation the change to an object neither it can be captured nor it can be traced back. This drawback makes the class-based technique an unsuitable object-modeling technique for many applications.

Since the prototype-based technique defines a prototype by structure, state or combination of both, therefore this flexibility of the technique allows capture of all types of changes (including simultaneous change to both parameters) to an object as a prototype. As each change is captured as a separate prototype, it can later be traced back.

2.5 Knowledge sharing mechanisms

A knowledge sharing mechanism of a data modeling technique brings reusability of knowledge in a system. The inheritance mechanism is the knowledge (only structure) sharing mechanism of the class-based technique. It fixes at the *compile-time* knowledge sharing pattern (called class-lattice) among objects of a system, but the knowledge among the objects is shared at the *run-time* through message-passing [2]. The class-based technique prohibits any change to the pattern (or a class-lattice) of a system at the *run-time* [2]. Many types of inheritances such as the *single (or simple) inheritance*, *multiple inheritance*, *restricted (or selective) inheritance,* and *repeated inheritance* are available in the technique (details can be seen in the literature [5, 24, 25].

The prototype-based technique provides a dynamic knowledge sharing meachanism - called the delegation mechanism. Note that the delegation mechanism allows sharing of both parameters of an object (or a prototype) to another object. The mechanism fixes at the *run-time* the knowledge-sharing pattern among objects of a system [2, 7]. The dynamic nature of the mechanism facilitates the user to pick-up at the run-time a prototype (or prototypes) for the purpose of knowledge sharing. This facility makes the delegation mechanism more flexible and powerful than the inheritance mechanism. Two types of delegations are reported in the literature, and those are the *single-parent delegation* and the *multi-parent delegation* [2, 7]. The working of the delegation mechanism is described in Section2.6. Conceptually they are close to the simple inheritance and multiple inheritance, respectively.

Stein proposed two mathematical models for the inheritance and delegation mechanisms and formally proved that "Inheritance Is Delegation" [7]. But both these models have limited capability because they capture only the cases of *single-parent* inheritance (or simple inheritance) and *single-parent* delegation. Since the two models do not handle the cases of multiple inheritance and multiple-parent delegation, therefore, it is difficult to say in general that the inheritance mechanism is equal to the delegation mechanism [7].

2.6 Retention of control while message-passing

In the class-based technique, when part of knowledge (which can be an instance-variable or a method) is requested in a sub-class that is referred to as the *self* class, the search for the requested knowledge starts from the self class. The self class first tries to meet the request from its own local knowledge. Note that here the class's own local knowledge means the class structure. If the self class is unable to meet the request, then the search goes up in class-lattice of the system by sending a message to its immediate super-class (or parent class) of the self class. The immediate super-class is called the *client class*. If all the immediate classes of the self class are unable to meet the request, then the search goes to super-classes at the next upper level of the self class in the search of the requested knowledge. In this way, the search continues until either the requested knowledge is found and fetched and the result is brought back to the self class or the requested knowledge is not available in any super-class of the self class at any level in the class-lattice and in this case an error message is generated [2, 7]. In the class-based technique, when a search for a requested knowledge goes up the super-classes of a self class, the control remains with the self class. If the requested knowledge is found in one of the super-class of the self class, then the knowledge is processed there and the result is brought back to the self class. Thus the whole process covers two-ways in the class-lattice, one for the search of the requested knowledge starting from the self class, and the other to bring the result back to the self class [2, 7].

On the other hand, the delegation mechanism *forwards* the control from the *self prototype* to a *client prototype* with the request for the knowledge. The requested knowledge is either found or not found in the prototypes of the system, in both the cases the control or the result of the requested knowledge never comes back to a self prototype [2, 7]. Due to this feature of the delegation mechanism, the search for a requested knowledge goes only one way in prototypes of a system. This feature of the delegation mechanism makes it more powerful and efficient than the inheritance mechanism [2].

2.7 Creation of a new object

In the class-based technique, a new object can be created in two different situations: first situation is when an instance is created in an already existing class of a class-lattice, and second situation is when first a new class is defined in a class-lattice and then its instance is created. In the first situation, schema of a class-lattice is not affected, whereas in the second situation schema is affected and also overall performance of the system.

In the prototype-based technique, there are two approaches for creating a new prototype by sharing the knowledge of the existing prototypes. The first approach is called the *copying* (or *cloning*) of the existing prototypes [1]. After *copying* knowledge from an existing prototype (or prototypes), the new prototype is independent and has no connection with the original prototypes. Later, any changes to the original prototype do not reflect in the new prototype and vice versa [1, 2, 7, 26]. In the second approach, a new prototype is created as an *offspring* of one or more existing prototypes [1]. This new prototype is defined in an incremental fashion as an extension of an existing prototype (or prototypes) by defining only additional knowledge in the newly defined prototype. The additional knowledge is the difference between the knowledge of the existing prototype (or prototypes) and the new prototype [1, 2]. In the second approach, a group of prototypes that share a common knowledge, can be grouped together by placing the common knowledge at some common place as a prototype, and new prototypes are defined as the offsprings of the common knowledge [1-3, 7, 26]. This technique puts a group of prototypes which share a common knowledge into a single class, and the technique is considered a hybrid technique of the class-based and prototype-based techniques [2].

2.8 Flexibility and efficiency

From our discussion in the above sections, it is easy to conclude that the prototype-based technique is considered more flexible than the class-based technique [2, 6, 9]. Generally, there is a concern about the efficiency of the object-oriented technology. As mentioned earlier, the prototype-based technique supports a dynamic (at run-time) knowledge sharing mechanism while the class-based technique supports a static (at compile-time) knowledge sharing mechanism. Therefore, the languages and systems that use the class-based technique, are considered more efficient than the

languages and systems that use the prototype-based technique, respectively [1, 2]. We also experienced the same while developing an application [10, 11] using the prototype-based programming language SELF version 4.0 [6, 7, 26]. The cause of the efficiency problem is the dynamic (i.e., the run-time) knowledge sharing of the delegation mechanism. If we select many prototypes for the purpose of knowledge sharing, the efficiency problem becomes more serious. In other words, for applications that are developed using this technique, the efficiency of an application depends on the number of prototypes that are selected for the purpose of knowledge sharing. The efficiency of small size applications is acceptable. A complete performance comparison of C++ (class-based) and SELF version 4.0 (prototype-based) object-oriented languages can be seen in [10, 11].

3. The Characteristics of the Class of Applications Suitable for the Prototype-based Methodologies

In this section, we identify six (6) characteristics of a class of applications, and argue that the prototype-based methodologies are appropriate to use for the development of this class of applications. The identified class of applications is considered as a separate class and is referred to as the *identified class of applications*. The characteristics of the class are identified based on our study and analysis of the two data-modeling techniques in Section 2.

3.1 Applications without hierarchical structure

Objects of some applications have inherent property that their objects can easily be organized into a strong hierarchical structure. Objects of this type of applications interact with each other in a fixed and pre-defined pattern. For example, structural organization of a company's employees (e.g. president, executive presidents, vice-presidents, etc.) are related to each other in a strong hierarchical structure). Objects of this type of application interact with each other in a fix and pre-defined pattern that is defined by the hierarchy of the application. For this type of applications, the class-based methodologies are appropriate to use for their development. But there are another type of applications such as software development environments, objects of this type of application are loose or no hierarchical structure among its objects, and they do not interact with each other in any fix and pre-defined pattern because they are loosely related to each other. It may also possible that an object of such type of application is defined for only some specific purpose. We consider these applications as a class of applications, and advocate that the prototype-based methodologies are appropriate to use for their development because the methodologies use the object-modeling technique which is consistent with the characteristics of the class of application (details can be seen in the previous section).

3.2 Rapid prototype development

Objects of some applications such as software development and computer-aided design and computer-aided manufacturing (CAD/CAM) are of *dynamic* and *evolutionary* nature (for details see Section2). For a specific purpose a new version an object of this type application is created and effect of this new version of the object is seen on over all application. Sometimes different versions of the same or different objects of an application are created quickly (or rapidly) to see and achieve some desired objectives and results. This process of rapid versioning is referred to as *rapid prototyping*. Note that a new version of an object can be created due to structural or stature change to the object. We consider the applications that need this type of requirements as the class of applications. Since the prototype-based technique captures all types of changes to an object in a single uniform manner, therefore the prototype-based methodologies are suitable for the development of this class of applications.

3.3 Incremental growth of objects

In some applications such as Computer Aided Construction (CAC) and Virtual Reality, knowledge of the objects grows in an incremental fashion. In this type of application, an object is created with its default knowledge (i.e., the knowledge that is available at the time of its creation), and later its knowledge grows over its life-span. For example in CAC, an object building is defined when it is initially built, then each time the building needs maintenance and the maintenance is done, the knowledge of the object is updated. This update of knowledge shows maintenance that is done to the building. The incremental growth of the knowledge of objects can be in the structure, state or both parameters of the object. From our study of object-modeling techniques, it is obvious that the prototype-based methodologies are appropriate for the objects that grow their knowledge in incremental fashion. The class-based methodologies will design a complex and inefficient class-lattice for such type of applications.

3.4 Desirability to trace back changes to a specific object

In the engineering applications such as CAC, CAD/CAM and other complex and engineering related applications, it is desirable to trace back the history of changes to the structure, state, or both parameters of a specific object. The consultation of the history of a specific object is helpful in making the maintenance decision for the specific object. The history provides the maintenance engineer complete details of all changes that occur to the object since its creation. In [21], this type of CAC application is described and the importance and benefits of maintaining and tracing the history of changes to the objects are given. This characteristic of this type of applications is an important feature which makes these applications different from others.

3.5 Where the grouping of objects is not important

This characteristic is extension of the characteristic, which is already discussed in Section 3.1.The class-based technique puts the objects into a group (or class), which

share common properties, and different groups are then linked into a graph (class-lattice). But if we have some applications (such as the application on Web and virtual reality) in which each object is its own type, and the objects of this type of application do not much interact with each other, then it is inappropriate grouping of the objects and make their class-lattice. Therefore, for this type of applications in which objects have almost heterogeneous properties, it is not beneficial to organize the objects of these applications into a hierarchical structure like class-lattice. Rather it more useful to put all objects of such type of application at the same level as the prototype-based technique suggests.

3.6 Simultaneous capturing of changes to both parameters of objects

In our study (see Section 2), we have concluded that the class-based technique and methodologies are not capable to capture simultaneous changes to both parameters of an object. But in some applications such as the hypermedia systems, the web systems and the semi-structure applications [27], this type of change to objects occurs frequently and it is essential to capture the change. This characteristic of these applications makes the class-based methodologies unsuitable for the development of this type of applications.

The six (6) characteristics of the identified class of applications are briefly mentioned and argued that the prototype-based methodologies are appropriate to use for the development of these applications.

4. Proposed Framework for the Prototype-based Methodologies

In this section, first we look at the two types of software development methodologies i.e., the function-oriented and class-based methodologies and the object-modeling techniques used by them. The function-oriented methodologies are formally defined in the next paragraph. Some representative function-oriented methodologies are Structure Analysis/Structure Design (SA/SD), SADT, etc. [28, 29], and the class-based object-oriented methodologies are Object Modeling Technique (OMT), Fusion, Booch, etc. [12, 13, 17, 30]. After the study of the two object-modeling techniques, we point-out the difference between the two techniques, and describe how the classical life-cycle model is modified for the class-based methodologies to accommodate this difference. We also describe how the difference between the class-based methodologies and prototype-based methodologies makes the classical-life model an unsuitable framework for the prototype-based methodologies. In the last part of the section, we propose the modifications in the classical life-cycle model to adjust the difference. After incorporating the modifications the classical life-cycle model works as the framework for proposing the prototype-based methodologies.

The principle of *aggregation* in the *function-oriented methodologies* groups together functions of a system that are constituents of a higher level function implementation. In other words, the main emphasis of the function-oriented methodologies is on the system functionality, and the methodologies are referred to as the *function-oriented methodologies* [17]. In the class-based methodologies the principle of aggregation groups together functions (or methods) that operate on the same set of data. The main emphasis in the class-based methodologies is on designing object classes of a system and their organization [17]. Due to emphasis on different aspects of problem statement of a system in these two types of methodologies, the class-based methodologies spend more time and effort in the design phase than the analysis phase as compared to the function-oriented methodologies (for details see [17]). In other words, the difference in the object-modeling techniques of the function-oriented methodologies and the class-based methodologies has affected the processing of the two phases (the analysis phase and design phase) of the classical life-cycle model. The difference is accommodated through a modification which recommends that for the class-based methodologies the design phase of the classical life-cycle model spend more time and effort than its analysis phase. We conclude that the difference between two classes of methodologies may pursue modifications to the classical life-cycle model to accommodate the difference.

The prototype-based methodologies differ from the class-based methodologies since they use two different object-modeling techniques. We magnify the main differences between these two types of methodologies and they are based on our study of the two object-modeling techniques in Section 2. The differences are listed as follows:

(i.) The prototype-based methodologies and the class-based methodologies use different object-modeling techniques. The object-modeling technique that is used by the prototype-based methodologies models objects in entirely different and unique manner.

(ii.) The prototype-based methodologies are more implementation-oriented than the class-based, because their emphasis is more on design and run-time (or dynamic) aspects of objects of a system.

(iii.) The prototype-based methodologies do not organize objects of an application into a hierarchical structure like class-lattice, rather they place objects of a system at the same level, and the knowledge-sharing pattern among the objects of the system is fixed at the run-time.

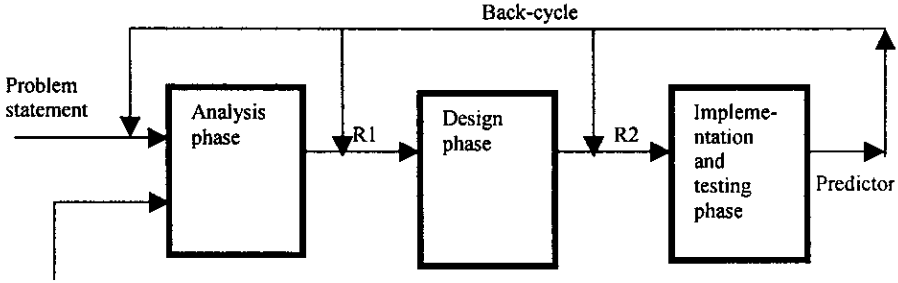
(iv.) Objects of a system which are developed using some prototype-based methodology, pass only once through the analysis process (or phase) in their life-span when the system is developed for the first time. After the system development, each update to the objects of the developed system is only processed by the design and implementation phases. It means that objects of a system are processed only once by the analysis phase and many times by the design and implementation phases of a prototype-based methodology. But this

is not the case for development of a new system and later updating the developed system using some class-based methodology, in both cases a fixed order sequence of the phases is followed. This difference between the two types of methodologies points out that the design phase and implementation phase of the prototype-based methodologies hold an *iterative* property since objects of a system may be processed by the two phases more than once in their life-spans. This difference is also further explained in Section 4.1.2.

4.1 The Proposed modifications and the framework

In this section we propose modifications to the classical Water-Fall life-cycle model, and the modifications are based on the above mentioned differences between the class-based and prototype methodologies, and the weakness and deficiencies mentioned in the next paragraph. The *modified life-cycle model* provides a framework for the prototype-based methodologies. Note that we use the terms the modified life-cycle model and the framework for the prototype-based methodologies in the same meaning.

The classical Water-Fall life-cycle model and its main phases are shown in Fig.1. Note that in the figure, the operation and maintenance phase is not shown. The function-oriented and class-based methodologies mainly follow the general guidelines of the classical life-cycle model but as it has been mentioned earlier, the class-based methodologies follow the model with minor modifications [17]. In the figure, an *additional knowledge* represents an update to objects of an already developed system.



Legend:
R1: Analysis report - output of analysis phase
R2: Design report - output of design phase
Product: A developed system

Fig.1. A common representation of classical water-fall life-cycle model.

Now we list the properties of the classical life-cycle model and point out that it holds, weaknesses and deficiencies. These weaknesses and deficiencies are also present in the class-based and function-oriented methodologies.

(i.) The phases of the model work sequentially for both the cases; the development of a new system and update to an already developed system with additional knowledge. We refer to this property of the classical life-cycle model as a *static order* of its phases.

(ii.) The *Back-Cycles* (see Fig. 1) represent the revision process of a system that is under-development.

(iii.) During the development of a system, the only meta-data knowledge (i.e., system specifications and requirements) of the system is considered and acquired. The data knowledge of a system is neither considered nor acquired due to absence of guide-lines from the classical life-cycle model.

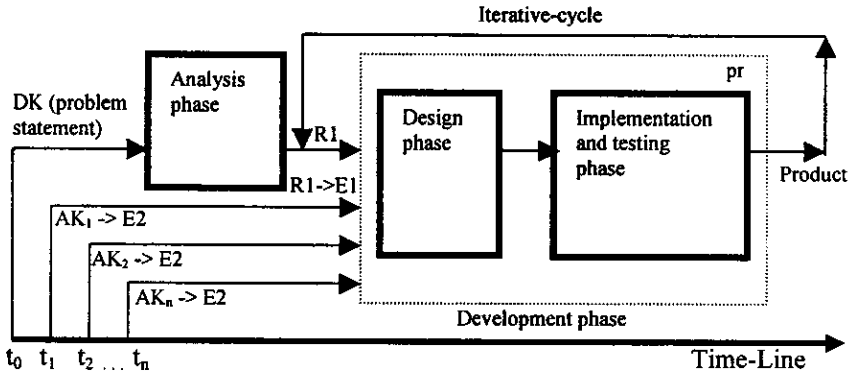
(iv.) The classical life-cycle model does not provide explicit guide-lines for the development of applications such as the web applications and hypermedia systems [27]. In these applications, the data or both data and meta-data are available before system development. We consider this as a deficiency in the classical life-cycle model.

The above listed weaknesses and deficiencies of the classical life-cycles model are the reasons that make it unsuitable for the prototype-based methodologies. Therefore, these weaknesses and deficiencies of the classical life-cycle model provide the justification and basis for its modification. Keeping these weaknesses and deficiencies and the characteristics of the prototype-based methodologies in mind, we suggest the following modifications in the classical life-cycle model.

Figure 2 shows overall structure of the modified version of the classical life-cycle model, and we refer it to as the *modified life-cycle model*. For the first case, when a system is developed from scratch both the modified life-cycle model and the classical life-cycle model function in the same fashion. But for the second case, when an additional knowledge is incorporated into an already developed system, then they differ (compare Fig.1. and Fig.2.). These two cases are shown as two types of events that occur at different time instances on the *Time-Line*. The time instance t_0 on the Time-Line represents the occurrence of the first type of event, when a system is developed from scratch and both the modified life-cycle model and the classical life-cycle model work in a similar manner. The time instances $t_1, t_2, \ldots, t_n$ represent the occurrences of the second type of event on the Time-Line, when the instances of additional knowledge $AK_1, AK_2, \ldots AK_n$, respectively updates an already developed system. At these time instances, the modified life-cycle model works differently from the classical life-cycle model. This is the second type of event which makes the modified life-cycle model different from the classical life-cycle model. The two types of events are formally defined in Section 4.1.2.

This is the occurrence of the second type of event that makes the modified life-cycle model suitable to be followed by the prototype-based methodologies.

It is already mentioned that both the design phase and implementation phase of the modified life-cycle model works iteratively and closely at occurrence of each instance of update to an already developed system. In other words, each update to an already developed system costs one iteration to the modified life-cycle model. An iteration means processing of the two phases. Due to this reason we consider both phases as a single phase, and refer it to as the *development phase* (see Fig. 2). In the figure, the development phase is shown by a rectangle with dotted sides, and one iteration is equal to one processing of the development phase.



Legend:

DK: Default knowledge (problem statement) of system at time t_0
AK_1: Additional knowledge of system available at time t_1
AK_2: Additional knowledge of system available at time t_2 . . .
AK_n: Additional knowledge of system available at time t_n
R1, R2 and Product: hold the same meaning as they hold in Fig. 1

Fig. 2. The modified classical life-cycle model.

The iterative property of the development phase is shown by the *Iterative-Cycles* in Fig. 2. The Back-Cycles (in Fig. 1) and the Iterative-Cycles (in Fig. 2) differ in their objectives and functions. An Iterative-Cycle represents the incorporating process of additional knowledge into an already developed system, whereas, a Back-Cycle represents the incorporating Process of a revision to an under-development system. The

first difference between them is that a Back-Cycle represents a revision to an under-development system. and an Iterative-Cycle represents an update to an already developed system. Another difference between the two cycle models is that a Back-Cycle can be initialed from any phase of the classical life-cycle model, whereas an Iterative-Cycle can only be initiated for the development phase (see Fig. 2). Note that a *product* (in Fig. 2) means an already developed system and also after completion of each iteration of the development phase.

The modified life-cycle model consists of two phases, namely: *analysis phase and development phase* as shown in Fig. 2. In the next two sections we describe the processing guidelines of the two phases which can also be followed by the prototype-based methodologies.

4.1.1 Analysis phase

General processing activities such as detail and comprehensive study of problem statement and identification and analysis of the system requirements of this analysis phases and the analysis phase of the classical life-cycle model are alike in the case of development of a system from scratch. Here we mainly describe those processing activities of the analysis phase which are different from the analysis phase of the classical life-cycle model. After completing the preliminarily processing activities such as detail study of a system, and identifying and analyzing the system requirements, this phase suggests to pick-up an initial list of prototypes of the system. The list is referred to as the *list of candidate prototypes*. This list can be prepared using similar criteria that are used by the class-based methodologies such as OMT and some others [17, 30]. The criterion that is commonly used by the class-based methodologies picks the *nouns* from a problem statement, and considers them as candidate prototypes. After preparing a list of candidate prototypes, the list is further processed, which includes discarding of the redundant and vague prototypes from the candidate prototypes, and merging identical and closely related prototypes, and a *final list of prototypes* is prepared. The output of the analysis phase is an analysis report which is denoted as R1 in Fig. 2, and it contains the final list of prototypes and other necessary information about the system such as relationships and links among the prototypes.

The main difference between this analysis phase and analysis phase of the classical life-cycle model is that a system is processed by this phase only once in life-span of the system, when the system is developed from scratch. Whereas, the analysis phase of the classical life-cycle model may process a system many times in its life-span (see Fig. 1). This phase also differs from the analysis phase of the classical life-cycle model in providing guidelines for study and analysis of both data and meta-data knowledge (if they are available) of a system during the analysis phase. This difference mainly makes the modified life-cycle models suitable for the prototype-based methodologies.

4.1.2 Development phase

As it has been mentioned earlier, the development phase consist of two phases, i.e., the design and implementation phases that work closely due to the characteristics of the prototype-based technique. This object-modeling technique does not differentiate much between the processes of design and implementation of an object (for details see Section 2). The input to the development phase is the output of the analysis phase, which is denoted *R1*. In this phase, the data types of instance-variables of every prototype are defined and default data-values to the instance-variables are assigned. Also, instance-variables are identified, defined, and assigned data values in the case of an update to an already developed system. In Fig. 2, the time instances $t_1, t_2, \ldots, t_n$ show those cases. Methods of every prototype are designed and coded in these phases.

A system schema is developed using the final list of prototypes and establishing the delegation links/relationships among the prototypes of the system. The links are defined using the information that are collected in the analysis phase in the case of a new system development, or from additional knowledge in the case of an update to an already developed system. The run-time scenario and interface of the system are also designed and developed in this phase. These things can be done using the report *R1*.

The development phase is triggered on the occurrences of the two events that are referred to as ***E1*** and ***E2***. ***E1*** triggers the developed phase only once in the life-span of a system, when the analysis phase completes its processing. This trigger of the development phase is denoted by *R1* -> ***E1*** in Fig. 2. ***E2*** triggers the development phase to update an already developed system using additional knowledge. In Fig. 2, *n* number of triggers of ***E2*** are shown, and the triggers are denoted by *AK1* -> ***E2***, *AK2* -> ***E2***, . . ., *Akn* -> ***E2*** in the figure.

When ***E2*** is triggered, the development phase takes both a *product* (an already developed system) and additional knowledge as input, and process them to incorporate the additional knowledge to the *product* following the guidelines which are mentioned earlier. The output of the development phase is an updated version of the *product* in the case of occurrence of ***E2***, or a newly developed system in the case of occurrence of ***E1***. Now we summarize the main functions of the development phase as follows:

(i) It defines the data types and assigns data values (if available) to instance-variables. Also, it designs and codes methods of the prototypes. This function corresponds to occurrence of ***E1***.

(ii) It incorporates additional knowledge to an already developed system by using function described in (i). This function operates on the occurrence of ***E2***.

(iii) It establishes relationship among prototypes by using the information that is collected for this purpose in the analysis phase and is available in *R1*, or from additional knowledge in the case of an update of an already developed system.

This function of the phase also develops a system schema by defining the delegation links among the prototypes of the system.

(iv) It designs and implements interfaces and the run-time knowledge sharing patterns of a system.

5. Conclusions and Future Work

We have presented a study and analysis of the two object-modeling techniques of the object-oriented paradigm. Based on this study and analysis, we have identified the characteristics of a class of applications, and argued that the class-based methodologies are inappropriate to use for the development of the class of applications. We have also studied the classical Water-Fall life-cycle model which is used as a framework (or guidelines) by the class-based and function-oriented methodologies. We have pointed out its weaknesses and deficiencies in the model, and argued that the classical Water-Fall life-cycle model in its classical form is unsuitable to use for proposing the prototype-based methodologies. We further argued that it is necessary to modify the classical Water-Fall life-cycle model to make it a suitable framework for the prototype-based methodologies. For this purpose, we have proposed the modifications to the classical life-cycle model.

The modified life-cycle model consists of two main phases, namely: the analysis phase and the development phase. The main feature of the analysis phase is that it works once and processes information of problem statement of a system when initially the system is being developed. Whereas, the development phase may work and process information of a system more than one time in the life-span of the system. The development phase allows to capture both parameters (meta-data and data) of objects of the system. These salient features of the modified life-cycle model make it a suitable framework for the prototype-based methodologies.

In our opinion the proposed framework can be helpful in proposing a class of methodologies, i.e., the prototype-based methodologies. These methodologies can be used for developing the identified class of applications. We are actively working for proposing a prototype-based methodology using the proposed framework.

References

[1] Borning, A. H. "Classes Versus Prototypes in Object-oriented Languages." *ACM/IEEE Fall Joint Conference*, (1986), 36-40.

[2] Lieberman, H. "Prototypical Objects to Implementation Shared Behavior in Object-oriented Systems." *Proceedings of the ACM International Conference on Object-oriented Programming Languages, Systems and Applications (OOPSLA'86)*, 1986, 214-223.

[3] Fotouhi, F., Shah, A., Ahmed, I. and Grosky, W. "TOS: A Temporal Object-oriented System." *Journal of Database Management*, 5, No. 4, (1994), 3-14.

[4] Nguyen, G. T., and Rieu, D. "Schema Evolution in Object-oriented Database Systems." *Data & Knowledge Engineering Journal, North-Holland*, 4, No. 1 (1989), 43-67.

[5] Kim, W. "Introduction to Object-oriented Databases." The MIT Press, Cambridge, Massachusetts, USA, 1990.

[6] Chambers, C., Unger, D. and Lee, E. "An Efficient Implementation of SELF Dynamically-typed Object-oriented Language Based on Prototypes." *Proceedings of ACM International Conference on Object-oriented Programming Languages, Systems and Applications (OOPSLA'86)*, 1986, 49-70.

[7] Ungar, U. and Smith, R. B. "SELF: The Power of Simplicity." *Proceedings of the ACM International Conference on Object-oriented Programming Languages, Systems and Applications (OOPSLA'87)*, 1987, 227-242.

[8] Dahl, O.J. and Nygaard, K. "SIMULA - An ALGOL-Based Simulation Language." *Combinations of the ACM, 9(9)*, (September 1966), 671-678.

[9] Aksit, M., Dijkstra, W. J. and Tripathi, A. "Atomic Delegation: Object-oriented Databases." The *IEEE Software Journal*, (March 1991), 84-92.

[10] Shah A. and Mathkour, H. "SELF Programming Language as an Implementation Tool." T*he 16th National Computer Conference*, Saudi Arabia, (February 4-7, 2000), 347-358.

[11] Shah A. and Mathkour, H. "Developing an Application Using SELF Programming Language." *ECOOP'96 Workshop WS14*, Linz, Austria, July 1995 (accepted)

[12] Booch, G. "*Object-oriented Design With Applications.*" Redwood City, California: Benjamin Cumming, 1991.

[13] Coleman, D. *et al.* "Object-oriented Development - The Fusion Methods." Prentice Hall, International, Inc.. 1990.

[14] Jackson, M. A. "System Development." Englewood Cliffs, New Jersey: Prentice Hall , International, 1993.

[15] Seidewitz, E. V. and Stark, M. "Towards a General Object-oriented Software Development Methodology." *The ACM Ada Letters*, 7, No. 4 (1987), 54-67.

[16] Shumate, K. "Structured Analysis and Object-oriented Design are Compatible." *The ACM Ada Letters*, 11, No. 4 (1991), 78-90.

[17] Wirfs-Brock, R., Wilkerson, B. and Wiener, L. "Designing Object-oriented Software." Englewood Cliffs: New Jersey: Prentice Hall, 1991.

[18] Pressman, R. "*Software Engineering-A Practitioner's Approach.*" McGraw-Hill, 1992.

[19] Bernard Software Engineering Inc. "*A Comparison of Object-oriented Development Methodologies.*" 902 Wind River Lane, Gaithersburg, Maryland, USA. 1992.

[20] Embley, D., Jackson, R. and Woodfield, S. "Object-oriented Systems Analysis: Is It or Isn't It?." *IEEE Software*, (July 1995), 19-33.

[21] Shah, A., Fotouhi, F., Grosky, W., Al-Dehlan, A. and Vashishta, A. "A Temporal Object System for a Construction Environment." *Proceedings of the XIII Conference of the Brazilian Computer Society (SEMISH-90)*, September, 1993, 211-225.

[22] Shah, A., Fotouhi, F., and Grosky, W. "*Share-kno*: Knowledge Sharing Mechanism of the Temporal Object System." *The Journal of King Saud University, Computer and Information Sciences*, Vol. 11, (1999), 25-43.

[23] Katz, R. H., Chang, E., and Bhatega, R. "Version Modeling Concepts for Computer-Aided Design Databases." *Proceedings of the ACM SIGMOD Conference*, 1986, 379-386.

[24] Kim, W., *et al.* "Composite Objects Support in an Object-oriented Databases." *Proceedings of Second International Conference on Object-oriented Programming Languages, Systems and applications (OOPSLA)* 1989, 118-125.
[25] Mayer, B. "Object-oriented Software Construction." Prentice-Hall, 1988.
[26] Hölzle, U. and Ungar, D. "A Third-generation Self Implementation." *Proceedings of the ACM International Conference on Object-oriented Programming Languages, Systems and Applications* (OOPSLA) 1994, 229-243.
[27] Nestorov, S., Abiteroul, S. and Motwan, R. "Extracting Schema from Semistructures." *ACM-SIGMOD Record*, 27, No. 2 , June 1998.
[28] DeMarco, T. "Structured Analysis and System Specification." New York: Yourdon Press, 1978.
[29] Yourdon, E. and Constantine, L. "Structured Design." Englewood Cliffs, New Jersey: Prentice Hall, 1979.
[30] Rumbaugh *et al.* "Object-Oriented Modeling and Design." Englewood Cliffs, New Jersey: Prentice Hall, 1991.

إطار عمل لطرائق تطوير البرمجيات المؤسسة على استخدام نماذج برمجية ابتدائية

أباد شاه

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات،

جامعة الملك سعود، ص.ب: ٥١١٧٨ ، الرياض ١١٥٤٣ ، المملكة العربية السعودية

(قدّم للنشر في ١٩٩٩/١١/٢٨م؛ وقبل للنشر في ٢٠٠٠/١٠/١١م)

ملخص البحث. في النمذجة الشيئية، هناك أسلوبان فنيان متاحان للحصول على نماذج لمواضيع من عالم الواقع ، أحدهما مؤسس على النوع، والآخر مؤسس على النموذج الابتدائي.

وفي هذا البحث، فإننا سندرس ونحلل كلا الأسلوبين المستخدمين للنمذجة الشيئية، وبالتالي سنستخدم النتائج في تحديد أنواع التطبيقات الممكنة، ومن ثم نطرح للمناقشة ونساند الرأي القائل بأن الطرائق المبنية على النموذج المؤسس على النوع كأسلوب للنمذجة الشيئية، لا تصلح كأساس لتطوير هذا النوع من التطبيقات.

وبالنظر إلى متطلبات هذا النوع المحدد من التطبيقات، فإننا نرى شدة الحاجة إلى هذا النوع من طرائق تطوير البرمجيات، والذي يشار إليه بالطرائق المؤسسة على النموذج الابتدائي.

وفي هذا البحث، نقترح أيضاً التعديلات اللازمة على نموذج تطوير البرمجيات التقليدي المعروف بدورة حياة الشلال، وهذه التعديلات تؤدي إلى جعل هذا النموذج متوافقاً مع متطلبات طرائق التطوير المؤسسة على النموذج الابتدائي. وتؤدي هذه التعديلات ونموذج دورة الحياة المنبثق عنها، إلى إيجاد إطار عمل وإرشادات أساسية لاقتراح طرائق مؤسسة على النموذج الابتدائي.