A Prototype-based Approach to Meta-Modeling using SELF

[1]Syed Ahsan [2] Amjad Farooq, [3]Abad Shah

[1, 3] Al-Khawarizmi Institute of Computer Science
[2] Department of Computer Science and Engineering, University of Engineering and Technology, Lahore
ahsancs@hotmail.com

Abstract: The activities of system modeling and system implementation have traditionally been viewed as two distinct entities owning to the purported differences between modeling languages and programming languages. We however feel that with raised abstractions of programming languages and executable capabilities of modeling languages, these differences are no more distinct. Based on this argument, in our opinion, SELF, as a prototype based programming language is sufficiently rich to form the basis of prototype-based meta-modeling approach. Existing meta-modeling approaches do not provide adequate meta-design patterns in order to be able to alleviate inherited methodological deficiencies of class-based methodologies. We feel that our proposed approach may prove to be a suitable candidate for adoption by various agile practices to model today's complex and evolutionary systems. Also included is a comparison between class based and prototype based object modeling techniques to highlight the suitability of the later for modeling evolutionary domains. [Journal of American Science. 2010;6(10):52-59]. (ISSN: 1545-1003).

Keywords: Evolutionary System, Object modeling, Meta modeling, Prototype, SELF, Agile development, Software Engineering, Knowledge Sharing

1. Introduction

With increased reliance on computational tools, modern day systems are becoming increasingly complex as customers demand richer functionality delivered in ever shorter timescales. Conventional software engineering practices are not suitable for handling the complexity of modern system development because of their peculiar characteristics and ever evolving nature. More recently Meta modeling has been advocated to manage the design complexity of evolutionary domains by providing support to customized, flexible and agile methods that satisfy the requirements for stakeholder involvement and user participation, we feel that this approach has limitations because of the static structure of "*meta*-meta-modeling" architectures. Although these agile practices advocate an iterative, dynamic and agile approach to application development, we however feel that they have limited capability to model evolutionary domains because, i) Majority of them do not follow any formal software development model ii) Those which follow a formal model are based on one of the above mentioned conventional linear or static model, and iii) They suffer from the limitations of static analysis and design iv) Code migration and maintenance is difficult in case of "too agile" or "too casual" approach to software development.

In literature, for modeling the objects of evolutionary and explorative domains such as bioinformatics, the prototype-based object modeling technique has been identified as a more suitable candidate then the class-based object modeling technique (Chambers, 1992; Borning, 1986). SELF being a prototype-based language provides a rich environment for language design that supports the key requirements of a meta-modeling facility. Concreteness, uniformity, and flexibility make the physical world comprehensible. SELF attempts to apply these principles by using a model based on Prototypes that provides for a smooth transition from concrete to abstract and vice versa through one-to-one mapping between the representation and the object. The remainder of the chapter is organized as follows. In Section 2, we give the related work which includes limitations of Meta-modeling approach, the comparison and relative suitability of two object modeling techniques i.e., class based and prototype based technique to model evolutionary domains. The potential of SELF for meta modeling is discussed in Section 3. We conclude this research work by presenting conclusion and Future directions in Section 4.

1. Literature Survey

Software engineering has been enabling developers to cope with increasing complexity of software –intensive systems by better techniques of designing, implementing and testing the system. The focus has been on achieving the objectives of; a) more clearly defined methodologies b) raised abstraction of implementation tools and, c) better organization and automation of software verification and testing tools .

Model Driven Engineering (MDE) attempts to achieve the above mentioned objectives by proposing a framework in which any specification should be expressed by models, which are both human and machine understandable and can reside at any level of abstraction (Dedecker, 2001). Since these models are machine readable, the process of developing systems becomes iterative, refining abstract models to more concrete ones, and in the end, automatically generating and deploying the complete code (Walker, 1992).

In order to apply MDE in large software development projects, the Object Management Group (OMG) has launched the Model Driven Architecture (MDA) initiative to address the issues related to large development teams and a diversity of tools, such as model interchange, diagram interchange, model versioning and concurrent management etc.. In order to support MDA approach to software development, it is important to define precisely what language should be used to express models (Ungar and Smith, 1991) how to specify model transformations, how to exchange models, how to store and make models evolve, and more recently, how to generate code [54]. To achieve this MDE uses the techniques of meta-modeling and model transformation. Meta-modeling, through a meta-model, clearly defines a modeling language by specifying its abstract syntax along with its concrete syntaxes, in which a class of models can be precisely defined. Model transformation technique is used to clearly define relationships between models (Ahsan and Shah, 2008; Amber, 2002).

A meta-model must be part of a meta-model architecture which enables a meta-model to be viewed as a model, which itself is described by another meta-model. This allows all meta-models to be described by a single meta-model known as a meta-meta-model that enables all modeling languages to be described in a unified way. The traditional meta-model architecture, proposed by the original OMG MOF 1.X standards is based on 4 distinct meta-levels (Shah, 2001). These are as follows:

M0 contains the run time instances of application modeled at meta-level **M1. M2** is the architectural level which contains the meta-model that captures the language: for example, UML elements such as Class, Attribute, and Operation. **M3** is the meta-meta-model layer that describes the properties of all meta-models can exhibit. The meta-meta-model is the glue that binds the simplest set of concepts required to define any meta-model of a language.

2.1. Golden Braid Meta-model Architecture

An alterative and better representation of meta-model is provided through Golden Braid architecture (Shah and Mathkour, 1995) that relates meta-models, models and instances based on the fundamental property of instantiation, thus enabling description of an arbitrary number of meta-levels. Models at any level of abstraction are instances of this meta-architecture. This along with the ability to use meta-object protocol (MOP) that allows a program to keep its meta-model in sync with its runs time behavior, offers Golden Braid Architecture a great deal of flexibility (Chambers et al., 1989). We will show in Section 4 how our propose prototype based technique provides natural support for this behavior.

The quality of a meta-model can be measured by determining the quality of the

a) Abstract syntax definition, b) meta-operations definition, c) concrete syntax definition, d) semantics definition, and e) mappings to other languages. The Golden Braid Architecture proposes five levels to determine the quality of the above mentioned tasks (Chambers and Ungar 1991).

Table 1: Five levels of Golden Braid Architecture

Level	Abstract Syntax	Meta Operations	Concrete Syntax	Semantics	Mappings
Level 1	Defined	Partially defined	Not defined	Informal, incomplete	Not defined
Level 2	Relatively complete. Snapshots constructed and tested	Significantly defined	Not defined	Informally defined	Note defined
Level 3	Completely tried and tested	Completely defined	Defined but Partially formalized	Informally defined	In Initial stages
Level 4	Formalized and tested	Complete and Formalized	Completely formalized and tested	Initial models	Partially defined
Level 5	Complete and Formalized	Complete and Formalized	Complete and Formalized	Executable semantic model	Formalized and complete

At Level 5 the language definition will be complete and self-contained capable of generating semantically rich models capable of simulation, evaluation and execution. Even international standards such as UML do not exceed level 3 As we see in Section 4, SELF which is based on prototype technique of object modeling achieves level 5 owning to the uniformity, flexibility and concreteness provided by prototype based approach.

2.2. Limitations of Meta modeling approach

The aim of meta-modeling has been to raise design abstraction and to achieve full code generation to automate development, leading to improved productivity, quality and complexity hiding. Meta modeling approach has tried to address the challenge of managing complexity of evolutionary systems by providing a comparatively a unified and flexible design environments for languages (Ahsan and Shah, 2008; Agesen et al., 1993. However, it has the following non-exhaustive set of limitations:

i) Meta model focuses on high-level abstractions with other artifacts (of increasing concreteness) seen as of lesser value. We feel that low level abstractions are central to and useful in system development. and have a crucial role to play in the process;

ii) Existing approaches do not offers direct method implementation

iii) The basic metamodeling architectures are based on 'static' data models (mostly Entity Relationship Diagrams) and on their extensions. Hence the basic limitations that appear in the accurate expression of a method are semantic and syntactic weaknesses, which are inherent in the generic structure of the data model it SELF.

iv) Expensive change management in metamodeling associated to the static structure of "*meta*metamodeling" architectures.

v) Current meta-modeling and concrete syntax standards suffer from semantic ambiguity as they primarily define the syntax of a language and fail to give the semantic definition (Level 5 of Golden Braid Architecture). The semantic definition still requires a reference to software development artifacts, such as requirements specifications, models, and even program code. A lack of semantic ambiguity is particularly important in the context of a model-driven development as it is likely that several languages, or language variants will be used in any given development and this proliferation of languages introduces ambiguity in semantic representations concrete artifacts.

In literature, amongst various major features of existing MDA standards that contribute towards ambiguity in concrete artifacts, following are worth mentioning (Bornberg-Bauer and Paton, 2001; Amber, 2002):

- Concrete standards, in most cases, do not unambiguously refer to concrete artifacts and never to the actual language specification.
- Semantic of the language are not defined using a concrete model, but by using English

with the assistance of intermediate semantic models.

- To correctly and completely interpret a concrete artifact, its abstract syntax and semantics must be identified. Ideally, the concrete syntax standard should permit the artifact to be interpreted sufficiently so that related specifications of syntax and semantics can be identified if they are separate from the concrete syntax standard.

vi) Many modeling Languages such as UML represents programming concepts (classes, return values, etc.) with graphical symbols such as rectangles. This forces developers to visualize static structures (Chambers, 1992). Also for example, having a rectangle symbol to illustrate a class in a diagram and then equivalent textual presentation in a programming language, in most cases ends up with having the same information in two places without a significant improvement in automated code generation. Limited code generation possibilities force developers to start manual programming after design and a lot of effort goes into rewriting generated code and keeping all the other models up-to-date (Myers et al., 1992).

Meta-models should not be conceived to visualize code, but describe higher-level abstractions on top of programming languages. A few authors have suggested achieving this by narrowing down the design space through Domain-Specific Modeling (DSM) languages. However this can also be achieved by raising the abstraction level of programming languages as we will explain in Section 4.

vii) The above mentioned problems also influence and aggravate the model interoperability issues. Interoperability issues in the meta-model domain may occur in the definition, integration and representation of the syntax, semantics and notation of modeling languages.

Some important additional aspects to be considered in model interoperability are (Myers et al., 1992; Kniesel, 1999):

- The amount of available meta-attributes to define concrete attributes of a certain type may be limited.
- Existence of non-corresponding model fragments, i.e. their meta-models are partly not corresponding with the concrete models. This can result in information loss or in hidden information.
- Diversity of graphical representations and models cannot be understood after model exchange because of complete or partial loss of graphical information.
- History logs to record model changes which can be necessary in model synchronization in distributed model change scenarios or in evolutionary domains...

viii) One of the most widely known OMG standards for describing languages through meta-models, MOF (the Meta Object Facility), has a number of limitations (Seco and Caires, 2000; Flatt and Felleisen, 1998):

It is not rich enough to capture semantic concepts in a platform independent way.

MOF does not provide a means of expressing the concrete syntax of a language, whether it is a textual or diagrammatical syntax.

The limitations listed above are inherent in meta-modeling approach as this approach is an extension of class-based object modeling technique. Owning to this, the limitations of class –based object modeling technique as discussed below are reflected in Meta modeling.

The two object modeling techniques which are called class-based and prototype-based techniques form the basis of two types of software development methodologies; the *class-based methodologies* and *prototype-based methodologies* (Ungar and Smith, 1991; Elmasri and Navathe, 2002). These two types of methodologies mainly differ from each other as highlighted in Table 1 because they use the two different object-modeling techniques (Amber, 2002). 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 (Chambers et al., 1989).

3. THE POTENTIAL OF SELF FOR PROTOTYPE-BASED META-MODELING

Modeling languages have traditionally been viewed as distinct from programming languages owning to the purported differences between their abstract syntax, concreter syntax and semantics. The reason for this view point has been the differences between:

a) The level of abstraction that the languages are targeted at. For instance, UML tends to focus on specification whilst Java emphasizes implementation.
b) The representation choice of a language's concrete syntax. Modeling languages tend to provide diagrammatical syntaxes, whilst programming languages are textual (Garzotto et al., 1991).
c) Modeling languages have been traditionally viewed as having an informal and abstract semantics whereas programming languages are significantly more concrete due to their need to be executable (Smith et al., 1994).

However, with raised abstractions of programming languages and executable capabilities of modeling languages, these differences are no more distinct. Both have a concrete syntax, abstract syntax and semantics [52]. For example, Java has been widely extended with declarative features, such as assertions, whilst significant inroads have been made towards developing executable versions of UML. Similarly, there is already a human readable textual form of UML and tools that provide visual front ends to programming languages like Java are commonplace. With these developments, modeling languages and programming languages are increasingly viewed as being one and the same. Based on this argument, in our opinion, SELF, as a prototype based programming language is sufficiently rich to form the basis of prototype-based meta-modeling approach. SELF with its following features and capability fulfils the requirements of meta-modeling facility.

Concreteness
SELF uses prototypes which make objects to be easily comprehended, manipulated, directly inspected and even tested: new objects in SELF are created through copy and extending an existing object, a process called cloning (Borning, 1986). In class-based systems, an object is instantiated from a description, and thus is on a less concrete level. All display objects (circles, frames, buttons, pieces of text, and so on) in the SELF user interface inherit morph behavior which is the default behavior of the basic graphical object, called a "morph.", and are therefore kinds of morphs, acquiring concrete behavior by default (Chambers and Ungar, 1989; Lee, 1988).

Also, any SELF object can be viewed as a kind of morph called an "outliner," so the task of modifying or making new SELF objects takes place in this concrete world. The language, being based on prototypes lets the programmer work with real data structures rather than descriptions. Concreteness is in the user interface architecture in four ways: a physical look and feel, a single SELF level representation, the reification of layout constraints, and the use of embedding for composite structure (Smith et al., 1994).

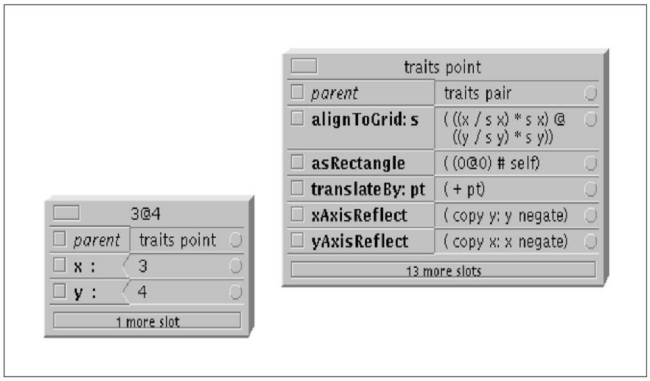

Figure 1: Two SELF Objects with 'morph' behavior

Uniformity
SELF enables a few concepts to be used to understand everything as everything is an object composed of named slots, and all computation is performed by message passing between objects. The user can directly take apart an application down to a very low level (Chambers and Ungar, 1989). Even the programming environment itself can be modified and deconstructed for use in applications. There is no system level distinction between using an application and changing or programming it (Lee, 1998). Enabling immediate and direct access to pieces of a running application can save time and enhance the sense of direct effect.

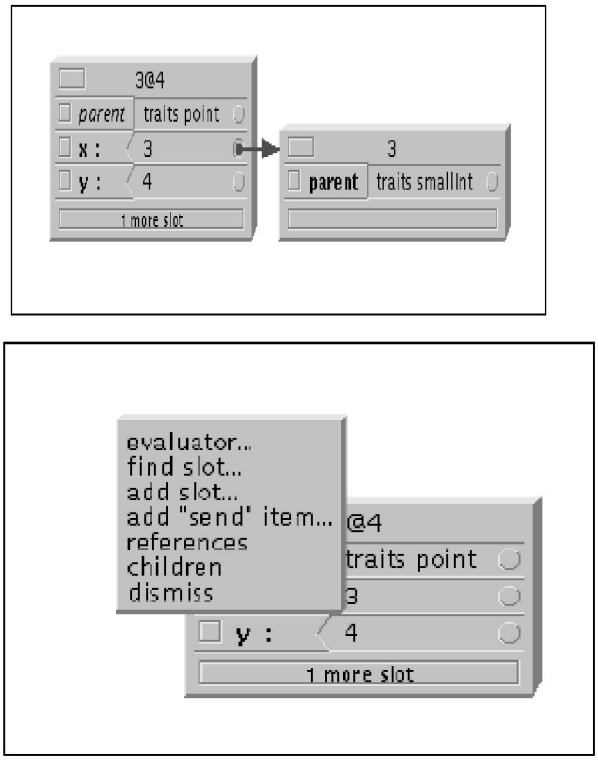

Figure 2: SELF slots are directly accessible and there is no run/edit distinction

Uniformity in the language semantics is achieved because everything is an object, all computation happens by message passing, and because slots are used to hold both state and behavior (Lee, 1998; Xin and Jian, 2008). Uniformity is in the user interface architecture because a single kind of display object, used to a very low level, can be directly accessed, and because there is no run/edit distinction. The environment itself is made of morphs, and is therefore available for reuse and modification.

Flexibility

In SELF-4.0, the morph serves down to a quite low level of graphics detail and everything the user perceives as a graphical entity is manipulable as a concrete object (Chambers and Ungar, 1990). The uniformity of SELF helps achieve flexibility by applying uniform change mechanism throughout the system. Use of message-activated slots for both state and behavior, and the lack of a class/ non-class distinction help achieve it further. SELF being a "prototype" based system, any object can have inheritance children, or can itself inherit from any other object. Also inheriting from an object gives you access to that object's slots. This simple object model enables the SELF environment to use a single object representation mechanism, the outliner, to present all the state and behavior available to an object through itself and its parents (Lee, 1998; Xin and Jian, 2008).

From the above discussion, it is evident that SELF provides a rich environment for language design that supports the key requirements of a meta-modeling facility. Concreteness, uniformity, and flexibility make the physical world comprehensible. SELF attempts to apply these principles by using a model based on Prototypes that provides for a smooth transition from concrete to abstract and vice versa through one-to-one mapping between the representation and the object .The object in the interface is intended to be concrete, immediate, and primary facilitating its conformance to the golden-braid metamodel architecture ensuring that the language, including its semantics is completely self described. One of the most important advantages of using SELF as a meta-modeling language is its adaptability to changes. When a change occurs, be it at the highest level of abstraction (e.g., a change in the requirements of the system) or at a lower level of abstraction its impact is well localized and the parts that are not touched by the change are immediately reusable.

SELF also supports the following features that are an essential part of the development process. These include:

• Execution: allows the model or program to be tested, run and deployed;
• Analysis: provides information of the properties of models and programs;
• Testing: support for both generating test cases and validating them must be provided;
• Visualization

After presenting briefly the potential of a prototype based language such as SELF we now clearly define a Prototype-based methodology so that the steps required to apply SELF as a meta-modeling language are outlined.

A Prototype based language such as SELF allows the methodologists to develop models at different levels of abstraction just like code. The advantage of having a Prototype-based methodology is that it clearly defines each step to be taken, forcing the developers to follow the defined methodology in this way. It specifies the sequence of models to be developed, and how to derive a model from another one at the abstraction

level immediately above it. Providing developers with such a methodology will ensure that they know at any moment during the development life cycle what is to be done next and how to achieve it.

4. CONCLUSION AND FUTURE DIRECTIONS

Contrary to the argument provided by Stein in his work *Delegation is Inheritance*, in this paper we argued that although delegation, just like inheritance, is a knowledge sharing mechanism, it is not constrained by strictly static inheritance like structure (class lattice) of class-based methodologies. In evolutionary system modeling, there are situations where prototype-based approach is not only more powerful but also entirely different than class-based methodology. We feel that by dealing with concrete entities instead of conceptual ones, our proposed approach has increased concreteness, uniformity and flexibility. The increased flexibility is also the result of its ability to dynamically change the Knowledge Sharing graph. These features helped us to realize the feature of *Dynamic Design.*

Another important contribution of our proposed methodology is the realization of the concept of *Dynamic Design.* This has been possible owning to the ability of a prototype to change its parent dynamically. Dynamic Design enables Prototype based software development methodology to accommodate and handle the specialized needs of modeling evolutionary domains such as Bioinformatics. This flexibility is also the result of concreteness and uniformity that is achieved by treating everything in a system as a concrete prototype.

With the availability of more formal methods and tools, Prototype based object modeling languages will get their well deserved importance and be accepted by the mainstream software development community. Class-based design functionality can be achieved through Class-less techniques.

We have taken first step towards formalizing the Agile software development using evolutionary and dynamic prototype based languages such as SELF. However, the object-focused model is not only applicable to prototype-based languages. There is no reason why class-based languages like Smalltalk and C++ cannot be supported by an object-focused programming environment. We feel that with the availability of more formal methods and tools, Prototype based object modeling techniques and languages will get their well deserved importance and be accepted by the mainstream software development community.

After presenting the potential of a prototype based language such as SELF we are working on formally defining a Prototype-based methodology so that the steps required to apply SELF as a meta-modeling language are clearly and unambiguously outlined. A Prototype based language such as SELF allows the methodologists to develop models at different levels of abstraction just like code. The advantage of having a Prototype-based methodology is that it clearly defines each step to be taken, forcing the developers to follow the defined methodology in this way. It specifies the sequence of models to be developed, and how to derive a model from another one at the abstraction level immediately above it. Providing developers with such a methodology will ensure that they know at any moment during the development life cycle what is to be done next and how to achieve it.

REFERENCES

2. [33]Clemens A. Szyperski, *Import is Not Inheritance, Why We Need Both: Modules and* Classes, ECOOP'92, Springer-Verlag, pp. 19- 32.
3. Abrahamssons, P., Salo, O., Ronkainen, J., Warsta., J., " Agile Software development methods" , Review and Analysis. VTT Publications, 2002.
4. Agesen, O., Palsberg, J., and Michael I. "Type Inference of SELF: analysis of Objects with Dynamic and Multiple Inheritance", in Proc. ECOOP '93, pp. 247-267. Kaiserslautem, Germany, July 1993.
5. Ahsan, S. and Shah A.(2008), A Framework for Agile Methodologies for Development of Bioinformatics, The Journal of American Science VOL(4), PP 15-21, Marsland Press Michigan, The United States.
6. Amber, S., "Agile Modeling: Effective Practices for Extreme Programming and the Unified Process" New York. John Wiley & Sons, Inc., 2002.
7. Bornberg-Bauer, E. and Paton, N.W., 'Conceptual Data Modeling for Bioinformatics", Briefings in Bioinformatics. 2001.
8. Borning, A. H. "Classes Versus Prototypes in Object-Oriented Languages", Proceedings of the ACM/IEEE Fall Joint Computer Conference (1986) 36- 40.

9. Chambers C., " The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages" Ph.D. Thesis, Computer Science Department, Stanford University, April, 1992.
10. Chambers, C. and Ungar, D. "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs", In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990.
11. Chambers, C. and Ungar., D. " Making Pure Object-Oriented Languages Practical." In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.
12. Chambers, C., Ungar, D. and Lee., E. " An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes",OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, 1989. Published as SIGPLAN Notices 24(10), October, 1989.
13. Dedecker J. Prototype-based languages and their programming idioms. Capita selecta, Vrije Universiteit Brussel, Ecole des Mines de Nantes, 2001.
14. Deutsch, L. P., and Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages (1984) 297-302.
15. Elmasri, R.A. and Navathe, S.B. 2000. Fundamentals of Database Systems 3rd Edition. Addision-Wesley Pubishing. ISBN: 0805317554
16. Flatt M., and Felleisen. M. Units: Cool modules for HOT languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
17. Garzotto, F., Paolini, P., & Schwabe D. (1991). Authoring-in-the-Large: Software Engineering Techniques for Hypermedia Application Design. *Proceedings of 6th IEEE International Workshop on Specification and Design,* (193–201).
18. Kniesel G., Type-safe delegation for run-time component adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, Lisbon, Portugal, 1999.
19. Lee, E. Object Storage and Inheritance for SELF, a Prototype-Based Object- Oriented Programming Language. Engineer's thesis, Stanford University (1988).
20. Myers B., Giuse D., Vander B. *Declarative Programming in a Prototype-Instance*
21. Pierce B. C. *Types and Programming Languages*. MIT Press, February 2002. ISBN 0-262-16209-1.
22. Randall B. Smith and David Ungar, *Programming as an Experience: The Inspiration for SELF*. ECOOP'95, Springer Verlag
23. Seco J. C. and Caires L.. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, 2000.
24. Shah A., and Mathkour, H., "Developing an Application Using SELF Programming Language," *ECOOP'96 Workshop WS14,* Linz, Austria, July 1995
25. Shah, A., (2001) .A Framework for Life-Cycle of the Prototype-Based Software Development Methodologies. *the Journal of King Saud University, Vol. 13*, Pp. 105-125,.
26. Smith, R.B., Lentczner, M., Smith, W.R., Taivalsaari, A., Ungar, D., "Prototype-based languages: object lessons from class-free programming (panel)", OOPSLA'94 Conference Proceedings (Portland, Oregon, October 23-27), ACM SIGPLAN Notices vol 29, nr 10 (Oct) 1994, pp.102-112
27. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227- 241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
28. Walker, J. (1992). Requirements of an object-oriented design method. *Software Engineering Journal*, 102–113.
29. Xin F., Jian C. "A Framework and Methodology for Development of Content-based Web Sites" Department of Computer and Information Science University of South Australia Downloaded on November 13, 2008 at 04:48 from IEEE Xplore.

5/6/2010