International Journal of Business, Human and Social Sciences
ISSN: 2517-9411
Vol:2, No:6, 2008

Compiler-Based Architecture for Context Aware Frameworks

Hossein Nejati, Seyed H. Mirisaee, and Gholam H. Dastghaibifard

***Abstract*—** Computers are being integrated in the various aspects of human every day life in different shapes and abilities. This fact has intensified a requirement for the software development technologies which is ability to be: 1) portable, 2) adaptable, and 3) simple to develop. This problem is also known as the Pervasive Computing Problem (PCP) which can be implemented in different ways, each has its own pros and cons and Context Oriented Programming (COP) is one of the methods to address the PCP.

In this paper a design for a COP framework, a context aware framework, is presented which has eliminated weak points of a previous design based on interpreter languages, while introducing the compiler languages power in implementing these frameworks.

The key point of this improvement is combining COP and Dependency Injection (DI) techniques. Both old and new frameworks are analyzed to show advantages and disadvantages. Finally a simulation of both designs is proposed to indicating that the practical results agree with the theoretical analysis while the new design runs almost 8 times faster.

***Keywords*—** Dependency Injection, Compiler-based architecture, Context-Oriented Programming, COP, Pervasive Computing Problem

I. INTRODUCTION

PCs, PDAs, cellular phones, and hundreds of portable electronic devices containing microchips are our familiar partners in every day life which play an important role in every moment of modern lifestyle so that in many cases we are not even aware of these calculations among us. This lifestyle has led to a problem which in spite of developments in hardware technologies, software developers are still challenging with.

This problem is the ability to develop applications which have 1) Maximum adaptability with the running context, 2) Portability to different platforms, and also have 3) an acceptable simple process of development. This problem is also known as *Pervasive Computing Problem* (PCP).

The new embedded computers technology in mobile devices have led the PCP to a new stage at which requirement of a framework capable of developing applications having adaptability, portability, and simplicity is vital. This is because the new mobile applications should work with any type of hardware and software on which the may be run and adapt their behavior in order to reach desired goals. These type of applications are Context Aware applications.

Context Oriented Programming (COP) is one of the most important methods, introduced for implementing context aware architectures. COP aims to reach a context aware framework which changes the application behavior with respects to the running context in order to tune its steps towards reaching goals. In this framework, the final application can have different behavior, adapted to the environment. This property can address the adaptability and portability problems. Framework development tools enable developing COP application using simple and human-understandable logic which indeed address the simplicity problem. As it is clear, COP aims to address the Pervasive Computing Problem.

Although many researches have been done on the Context Aware frameworks, the implementation of this framework is yet a problem. Almost all of the context-aware applications are being used in the research laboratories [1]. This means the current implementations are still not suitable and powerful enough to answer the PCP in real environments. In this article, a new feasible design for implementing COP frameworks are presented based on an interpreter-based method by R. Keays (2003) [2]. Our new compiler-based architecture uses Dependency Injection (DI) technique to involve compiler languages (beside interpreter languages) as a development basis and therefore inheriting their power to the COP. After describing related works in the next section, advantages and disadvantages of the interpreter-based design is discussed in section three which is followed by describing the new design in section four. Finally, Implementation and conclusion are presented in the fifth and sixth sections.

II. RELATED WORKS

Different works have been introduced in related to COP, using approaches such as multiple inheritance, layered software architecture, and interpreter-based architecture, which each one has its strengths and weaknesses. Rather than implementing COP or semi-COP frameworks, several researches on related issues such as Typing, Matching and Binding have been also presented. In this section some of the related literature is described.

H. Nejati is PhD candidate, School of Computing, National University of Singapore (e-mail: nejati@nus.edu.sg)

S. H. Mirisaee is Master Candidate, Faculty of Information Science and Technology, University KEBANGSAAN Malaysia (e-mail: h.mirisaee@ftsm.ukm.my)

G. H. Dastghaibifard Assistant Professor, Computer Sciences and Engineering, Shiraz University, Shiraz, Iran (e-mail: dstghaib@shirazu.ac.ir)

Dey and Abwors (2000) have defined the context as follows: “Any information that can be used to characterize the situation of an entity. An entity is a person, a place, or a physical or computational object that is considered relevant to the interaction between a user and application including the user and application themselves [3].” By this definition, they also categorize meaning of the data into context class.

Dey and Abwors’ proposed architecture is consisted of four main components: application core, adoption system, context management system, and user interface. The whole design is based on web services to ensure maximum adoption to the running context [3]. In the adoption component, they have used a model [4] which Berhe and his research team have developed, for comparing all possible transformations and finding optimum path to achieve an adopted format to content situation.

ContextL [5] is an extension to Common Lisp Object systems which allows Context Oriented Programming. It provides means to associate partial class and method definitions with layers and to activate and deactivate such layers in the control flow of a running program. When a layer is activated, the partial definitions become part of the program until it becomes deactivated. This enables modifications in the program behavior according to the context of its use. In ContextL, the application layers can be designed only before releasing the application.

The Java 2 Micro Edition (J2ME) extends the functionality of Java by grouping device capabilities into specific categories [6]. J2ME architecture consists of three layers of abstraction to ensure that applications will execute across varying devices. The first layer, the Java Virtual Machine (JVM) implements a customized virtual machine for that device. The second layer (the Configuration layer) defines available features of the JVM and core Java libraries. Portable applications must meet the third layer (the Profile layer) which defines application programming interfaces (APIs), made available by the underlying layers. That specific application will run on all devices conforming to that profile. J2ME have actually chosen the other way in which context is virtually changed for host application.

Gassanenko [7] has added context and first class environments definitions to Forth programming language [8] by adding Object Oriented Programming (OOP) concepts [9]. These definitions can result in different behavior in different execution environments. These contexts do not extend further than function definitions.

Keays and Rakotonirainy [2] use the term COP for an approach that separates code structure (referred to as skeleton) from program parts which need to be changed according to different environments (referred to as stubs). When the environment changes, a procedure named context-filling substitutes old stubs with new stubs and therefore controls the program behavior. Finding the most suitable stub is done by “Match-Box” process which searches for a stub using parameters such as stub goal and context. This design is described in more details in the next section because it is used as a basis for the new compiler based design.

Another important portion of a COP framework design is to define a format for goals and contexts of program parts (i.e. typing) and to match and bind the closest available behavior to current context (i.e. binding) . Knowledge representation (KR) [2] is one of the most famous works to address typing.

Several approaches have been also proposed to address the matching and binding problems. Instances are the matching the nearest available service in the Blue-tooth Service Discovery Protocol [10] and XML tree to tree transformations in Zhang and Shasha algorithm [11], [12] for matching; and scoping by using local, global, and built-in Python dictionaries [9] Semantic Web [13] and RFD [14] for binding. However in this paper, this section of COP is not described in detail and instead, the focus is on details of injecting compiler language abilities into COP design presented in [2].

III. AN INTERPRETER-BASED ARCHITECTURE FOR COP FRAMEWORKS

In this section an interpreter-based approach presented by Keays (2003) in [2] is described and discussed in more details.

This is because the new compiler-based design is built on the basis of this architecture. Bold properties of this architecture is that 1) it is based on interpreter language specific properties 2) it has theoretically addressed almost all of the COP framework requirements. 3) The design is flexible and derived from an intuitive idea.

It is noticeable that Keays’ design also violated some minor COP requirement rules such as first class goal constructs (first class definition will be discussed later in this section) and at the end a demonstration implementation is presented in Python language using the Python dictionaries and XML-based knowledge representation.

Keays’ design is divided into three parts [2]:

1) Skeleton: is the user program’s main structure which contains basic logic of the program. Skeletons are actually traditional user programs which are converted to

COP style. In this style the code is implicitly divided into parts which do not have to change in context changes, and parts which are required to be changed in when the context changes to a specific status.

2) Stub which is a code, developed for a particular goal in a particular context. These codes are collected in a stub database and are indexed by their goal and working context. Stubs are pure codes which have COP style and can be attached to the skeleton after modifications (Binding process will be described in advance).

3) Match-Box which is a separate process which listens to user program requests for code updates, and changes the required code sections in the user program with respect to the request, current context, and the nearest stub matched to the request specifications.

Before detailed review, three more definitions should be covered which are used in Keays’ design:

Context refers to any particular information, demonstrating an entity or an entity status. Examples of context can be intensity of the light which a device receives, hardware specifications of the running device, or network bandwidth accessible for the application.

Goal is referred to a specific goal of using a variable, defining a function, writing a sub program, or developing an entire application. For example, a variable goal can be performing as an accumulator and a function goal can be releasing calculation results.

Gap is a program sections required to be changed in context-change events with respect to changed parameters of environment. Gaps have a specific syntax for declaration in which the goal of the section and the context parameter that should be watched for changing is specified. For example consider a mobile robot navigation program in which a gap is defined with its goal set to "sensing obstacles" and its context parameter to "available sensors". This part of code (stub) should be changed if the available sensors are changed (context-change event) and substitute with another stub which is developed to use current sensor(s) for sensing an obstacle in the way.

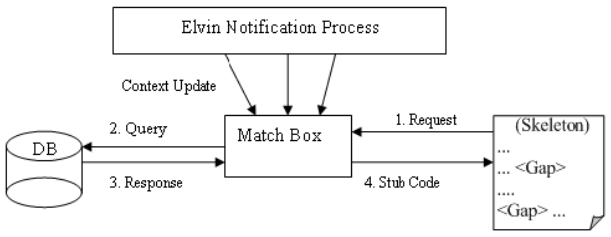

Fig. 1 Interpreter-based architecture for COP framework by R. Keays: 1) Skeleton request for a gap filling operation, sent to the Match-Box; 2) Match- Box query, using request parameters and Elvin Bus context information, sent to stub database; 3) The stub, selected according to the Match-Box query, is sent back to the Match-Box; 4) Modified stub, adapted to the skeleton, substitutes the gap.

During run-time when interpreter reaches a gap in the skeleton code (user program code) it sends a request to the match-box process including the gap parameters (goal and context). Match-box then fills in the gap with a sub which best matches the requested parameters. This procedure is called Gap Filling Procedure. As shown in Fig.1 Gap filling in the interpreter-based design can be divided into 4 phases: 1. Request, in which the skeleton reaches a gap and sends a request including goal and context (and extra other parameters if required), to the match-box for a gap filling; 2. Query, in which the match-box queries the stub database using the request parameters and current context status to find the best matching stub; 3. Response, which contains selected stub from the database; and 4. Modify and return the stub code, capable of being bound to the skeleton.

This design combines codes in run-time without need for code rewriting and therefore, updates user program whenever required.

A. Interpreter-Based Design Evaluation

Keays' architecture [2] has pros and cons which are almost direct results from using interpreter languages as a basis. Its three main advantages are: 1) effective design for addressing Pervasive Computing Problem, 2) relatively simple way of changing running process toward applying COP features, and 3) ability to change the skeleton codes without recompilation.

On the other hand, using this type of programming languages has the problem of time which can be divided into development time and running time sub-problems.

The simplicity of changing an interpreter in comparison with changing a compiler can be regarded as one of the most important motivations toward using an interpreter basis for implementing this framework. Interpreter language running processes are likely to be simpler than compilers where changes are needed for adding COP features and structures.

In addition, interpreters run program statements one bye one, leaving a chance for the gap filling operation to change forwarding codes whenever needed. Due to this particular feature of interpreters, pausing the interpreter running process (e.g. via a proper usage of semaphores), filling the gap, and then resuming the interpreter is possible when a gap filling request is fired. In this way the interpreter would not be aware of code changes and would start running the next statement which is then the first statement of bound version of an adapted stub.

Although it seems that the only weakness of using interpreter language basis is time problem, this problem has undermined the architecture. This problem can be divided into two timing sections: Running time, which is required time for released applications to run; and developing time, which is amount of time consumed on developing a COP application to be released.

Interpreters are normally slower than compilers when running the resulting application, moreover, intensive amount of input/output (I/O) in the former design slows down the application even more. Most of these I/O operations are required for searching in the skeleton and stub codes. Firstly, none of these codes are well-formed which results in a word-by-word linear search for keywords; Secondly, for binding the stub to the skeleton additional I/O operations are also required (e.g. for changing names to avoid collisions); And at last, when substituting the modified code in the skeleton body, the whole stub code should be rewritten in the user program. The time problem extends even to application development. Developers have to consume a significant amount of time for development because:

1) Developer is doomed to define goal and context for every variable and function which appears in the gap scope using a specific syntax.

2) Lack of modern interpreter based development environments make developing process slow.

The interpreter-based design presented in [2] has introduced a general method for solving COP problem. Implementing this method on the interpreter languages basis has added simplicity but also caused the architecture to perform relatively slowly.

Next section describes a compiler-based architecture in details which has inherited strong points of the later design and used DI technique to solve its weaknesses.

IV. COMPILER-BASED ARCHITECTURE FOR COP FRAMEWORKS

This section deals with the compiler-based architecture based on the aforementioned interpreter-based design. Two considerable issues for designing this architecture is keeping structure of the later design, and reducing compiler complexity and overload. Applying both together, results in solving the interpreter-based architecture weaknesses and remaining its advantageous points unchanged.

The most complex and most important component in Keays' architecture is the Match-Box. It is responsible for almost all of COP-related behavior of the framework - receiving gap-filling request, processing information from the real world, matching the best stub from database, binding, and placing final code in the skeleton gap. Therefore, for keeping the structure of the architecture, the structure of match-box should be kept. To mitigate the complexity, and remove the recompilation requirement, DI method has been utilized, changing some parts of the older design, but keeping the concepts unchanged. Before describing the new compiler-based architecture and using DI, a preliminary section of DI is presented here.

A. Dependency Injection

Dependency Injection (DI) is a programming pattern which generates a general interface to inject component dependencies and in this way creates a level of abstraction [15]. DI removes responsibility of object instantiation, initialization and configuration from the requester to an Object Factory also known as Container. Therefore, the requester component does not need to be aware of how the object is created, initialized, or configured and in this way dependencies between classes will be removed. DI has been also named as Inversion Of Control (IoC) in some contexts [15], but it is technically a type of IoC family as it removes control of object creation, initialization, and configuration from user class, to a general object factory class.

Interface Injection, Setter Injection, and Constructor Injection [16], [17], [15] are three different implementations of DI which our recommended method for COP framework implementation is Setter Injection. Reasons are described in "Design Details", (see IV-B).

B. Design Details

The compiler-based framework can employ DI technique for removing recompiling requirement while having code changes. It should be noted that combining these aspects should be done in such a way that it has minimum overhead, maximum speed, and benefits of both interpreter and compiler languages. This idea is described here first, following with relative designs for new database and DI object factory.

1) Combining DI and COP: Combining COP and DI is an idea, motivated from a similarity between these two aspects. DI wants to remove control of instantiating dependent objects from the user program and therefore relaxing these dependencies. COP similarly, wants to remove control of behavior changing with respect to the running context from the user program. Therefore COP can be considered as another type of IoC.

COP main goals are adaptability, portability, and simplicity and to reach these goals in a compiler-based platform, changing the program behavior without recompilation is one of the most important issues. Simultaneously, DI technique can create required objects dynamically, on the fly, and without any recompilation. However, for utilizing DI in our COP framework implementation there exist other points which should be considered. These issues are: (1) To keep the advantages of previous design, and (2) to reduce overhead of input/output (I/O) operations for configuration codes. These codes are used to build object factories and are usually in XML format. The overhead of configuration code should be then compare with the I/O operation of interpreter-based design to ensure that it does not exceed the previous design I/O time. For the first issue, it should be noted that the interpreter-based architecture owes its important properties to the matchbox.

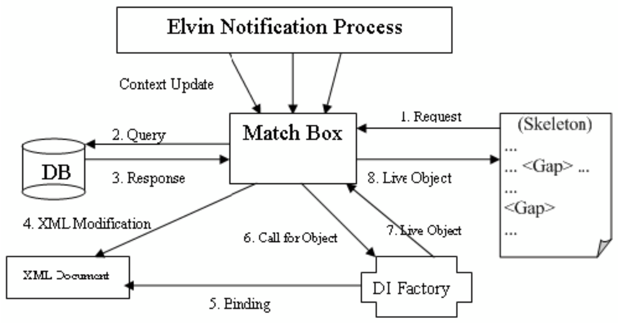

Fig. 2 Compiler-based architecture for COP frameworks: 1) Skeleton request for a gap filling operation, sent to the Match-Box; 2) Match-Box query using request parameters and Elvin Bus context information, sent to stub database; 3) Object creation parameters for the closest match, returned from the object database; 4) Modifying the XML configuration document to satisfy the object creation parameters, by Match-Box; 5) Binding the factory to the modified XML configuration file ; 6) Match-Box call for object creation, sent to the bounded DI factory; 7) Resulting object from the factory; 8) Return of factory produced, adapted live object to the skeleton, ready to be used.

This is the match-box which is responsible for receiving user program request, finding the best match for it, and binding it back to the skeleton. Therefore, match-box main structure should not be changed to maintain interpreter-based benefits as much as possible. In the new design match-box has an additional role of creating an adapted object for being used in skeleton. This object creation and adaptation is done via DI factory. The match-box can be divided to two main parts: first part is front end which receives the user program request and values from environment by Elvin Bus process and the second part is back end which includes searching for suitable stub in data base based on environmental values and filling the gap. DI factory relates to the back end and does not change the front end; therefore, match-box main structure will remain unchanged. After object creation by DI factory, matchbox consequently delivers the object to the user program. The resulting object should also implement an interface, using a unique signature in order to let the user program use it as a turn-on switch. This interface should activate the object to perform a particular task it is created for.

Introducing DI to the design also leads to database structure changes. I/O problem should be considered in both database and DI factory design. This particular design is discussed in the next two parts, “Database Design” and “DI Object Factory Design”.

Having these changes in the previous design, application developers are not restricted to use interpreter languages and both of interpreter and compiler are usable if they support object oriented programming. Fig. 2 shows the proposed design.

2) data base design: Database should store two types of data in compiler-based design: 1. Data required for creating each individual object 2. Data required for finding the best match, for match-box query. In this design database includes definitions for all of producible objects. There are different database schema which can be employed here. Selecting the schema depends on object definition method used in XML files because different types of defining objects require different information to be stored. Here three methods for XML file structures and therefore for database schemata are briefly discussed.

First method is storing entire set of objects for different conditions in a single XML file. The database is therefore merely one XML file and searching process is parsing this file and finding the suitable object. Advantage of this approach is that a separate database management system is no longer required and therefore system cost and load would decrease in simple cases having a few simple objects. In the other hand, XML file size drastically increases when number of objects increases (common in real pervasive environments) and this results in unacceptably slow search operations. There are two ways for searching (parsing) an XML document, SAX [18] and DOM [19], in both of which, a large-sized document decreases parsing speed significantly.

One alternative approach is having possible objects set, categorized based on their goal or context in several XML files, in a manner that no changes would be required. This means having a complete definition for each object before the running time. This causes XML document modification overhead to be omitted. A simple database is also needed to indicate XML file which contains an individual object definition. This method mitigates the effect of increasing objects number in comparison with the latter method and can be used having a larger set of objects. However, as in a real pervasive environment the number of possible cases (and therefore objects) is inevitably high, the XML files will again grow very fast and the parsing pace will tumble.

The third method has relatively small XML documents; each one contains only one object definition for a set of possible goals. The goals are categorized into several XML file based on their types. For example one XML file for I/O based goals; one for OS based goals, and one for network based. These object definitions do not contain actual class library or property values and will be set to real values when a request is received. The real values should be kept in a separate database. This causes the XML file to grow slower, but needs a more complicated database which can increase system cost for implementation and maintenance. In this solution the database should contain XML file URL, object name, class library, and actual values for object properties which are needed in initialization. Despite having the overhead of a separate database and more complicated implementation this method seems to be more reliable in real pervasive environments than the two others. This is again because it is closer to real pervasive environments which require defining a large set of objects.

As a conclusion for this discussion, it appears that the first method can be useful when the focus is on other framework components rather than database; the second suggested way is similar to the first one but can be used for more complicated contexts and can be realized for simple real applications; And the third is the most reliable one among presented approaches while it can manage a noticeably larger number of objects in each goal category but forcing overhead of a separate complicated database management system.

3) DI Object Factory Design: The two main DI frameworks are Spring-Java [20] from Sun family and Spring.NET from Microsoft family which is almost same, therefore only Spring.NET is described here. Spring.NET is chosen here to describe the schema and the most important parameters in creating an object factory in Spring.NET are: 1) Injection method type, 2) object recreation configuration, and 3) Object factory usage method [17]. The first parameter is a choice between using Setter, Constructor, or interface injection methods which is again related to the problem of large number of objects. Applying the constructor or interface injection method may lead to create hundreds of constructors and interfaces for each object category which must be handled by the object definers (i.e. the database element developers). In the other hand, using the Setter injection, the developers are required to indicate names of properties needed to be changed and their values. These properties can vary from the class to a simple text message.

After database finds the suitable class with its properties, the match-box changes the properties instantly and calls the factory to instantiate the suitable object. Setter injection seems to be relatively simpler than creating constructors and interfaces because the match-box takes responsibility of changing XML document when needed, but it still need a time consuming labor when the number of objects rapidly grow.

The second attribute has two possible values, being singleton or non-singleton (prototype), which configures type of object re-creation in the factory. The singleton type objects is only instantiated at the first time the factory is called to return it. The non-singleton type in the other hand is created on every incoming request for the object. It is clear that the DI factory must be a prototype object creator because using a singleton object, it is highly probable that the dated version of object does not satisfy requirements of a changed context.

Difference between static and instance factory methods, the third parameter, lie in ways of using (calling) the factory itself. In the static factory method there is a static method, responsible for object instantiation. So the factory can be called directly from factory class. But in the instance factories, a factory object should be first instantiated from factory class and then called for starting an object creation process. Here again, there is a trade-off between simplicity and over head. Although the static factory method is simpler, when the XML resource file needs to be updated, it must be re-declared and re-bound to the factory after each change. In COP framework case, the XML resource should be changed during the running time and therefore it is reasonable to use instance factory method (which does not need re-declaration) to eliminate this overhead.

At the end of this section, regarding to our discussion a brief overall plane is that, database should contain properties (and the values) needed to be changed and related XML URL; XML files should contain dummy object definitions for each goal and each XML file should be related to a particular context; and at last, object factory is a setter object creator which creates prototype (non-singleton) objects and is an instance object factory method.

V. Implementation Terms and Results

This section describes implementation terms for both interpreter and compiler-based architectures using the design details describes in previous sections to compare the features and determine how the proposed architecture is better than the interpreter-based version.

Selected scenario for this test was a context aware application which displayed a welcome message customized to language spoken in country of running context. For example application displays “welcome” if the context is Australia and “bienvenue” if it is France.

Python and C#.Net were programming languages which are used for interpreter simulated and compiler simulated implementations respectively, and Spring.Net was employed for dependency injection service. For the sake of simplicity, Elvin notification process and database search were simulated with fake functions which returned a controlled set of information. The former because of remaining unchanged in the new architecture, and the later due to its remarkable similarity in both designs.

For measuring the executing speed, gap-filling operation is done 1000 times for each implementation. This operation includes selecting one country randomly and substitute suitable stub in user program or injecting adapted object in it. These experiences are done using Microsoft Windows XP running on a Pentium 4, 2.8 GHz processor and 256MB of main memory.

Average run-time of 20 times running (each includes 1000 gap-fillings) is then presented. The test results show that compiler-based application was 7.8 times faster than the interpreter-based application like the prediction in the design section. Fig. 3 illustrates the implementation results.

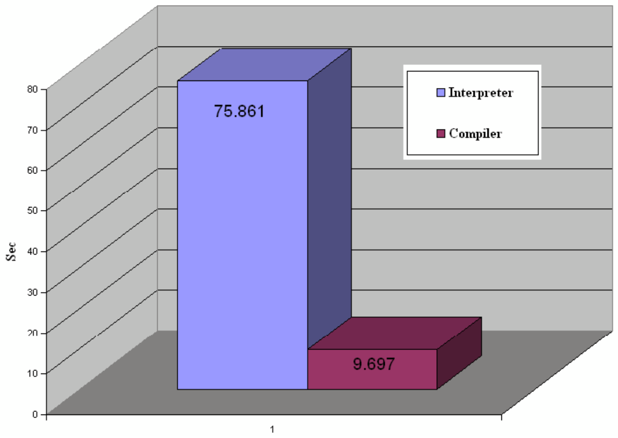

Fig. 3 it is Comparison between Interpreter-based design and Compiler-based design in running time. The new design finished the task in 9.697 seconds while the other one finished the same task in 75.861 seconds.

VI. Conclusion

Although automatic instrument usage in human life is growing with a significant pace, a powerful framework capable of addressing problems of real environment pervasive computing problem is not proposed yet. This type of programs must change their behaviors according to environment conditions.

In this article a new architecture based on an existing interpreter-based architecture was introduced and simulated. The new design has kept advantages of the previous and removed its major weaknesses by injecting possibility of using compiler programming languages via using dependency injection technique. Details of both previous interpreter-based and new compiler-based architectures were discussed and finally results from simulated applications of both were demonstrated. The test showed agreement with theoretical analysis while the new architecture was 7.8 times faster. Providing the ability for using the compiler languages in new design also improves programming and developing quality in produced applications because of existing compiler language features such as modern interactive development environments (IDEs).

References

[1] P.J. Brown, J.D. Bovey, X. Chen, “Context-aware applications: from the laboratory to the marketplace,” *IEEE Personal Communications*, vol. 4, no. 5, Oct. 1997, pp. 58-64.

[2] R. Keays, A. Rekotonirainy, “Context-Oriented Programming,” *International workshop on Data Engineering for Wireless and Mobile Access*, San Diego, USA, ACM Press, pp. 9-16, 2003.

[3] A.K. Dey, G.D. Abowd, “Towards a better understanding of context and context-awareness,” *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, The Hague, The Netherlands 2000.

[4] G. Berhe, L. Brunie, J.M. Pierson, “Content adaption in distributed multimedia systems,” Journal of *Digital Information Management* 3 (2), 95- 100 (special issue on distributed data management)

[5] D. Gelernter, S. Jagannathan, T. London, *Environments as First Class Objects*. POPL ’87, Proceedings.

[6] S. Helal, "Pervasive Java," IEEE *Pervasive Computing*, vol. 1, no. 1, pp. 82-85, Jan.-Mar. 2002.

[7] M.L. Gassanenko, “Context-Oriented Programing: Evolution of Vocabularies,” Proc. of the *euroFORTH'93 conference*, Marianske Lazne (Marienbad), Czech Republic, pp. 14, 1993.

[8] S. Pelc, “Programming Forth”, MicroProcessor Engineering Limited, - Copyright © 2005. available: www.mpeltd.demon.co.uk/arena/ProgramForth.pdf

[9] M. Lutz, *Programming Python*. O’Reilly and Associates. USA 1996.

[10] A. Sasikanth, “Enhancing the Bluetooth Service Discovery Protocol,” Honours thesis, University of Maryland, 2001.

[11] J. Wang, B.A. Shapiro, D. Shasha, K. Zhang, K. M. Curre, “An Algorithm for Finding the Largest Approximately Common Substructures of Two Trees,” IEEE *Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 889-895, 1998.

[12] D. Shasha, J.T.-L. Wang, K. Zhang, F.Y Shih, “Exact and Approximate Algorithms for Unordered Tree Matching,” IEEE *Trans. Systems*. Man, and Cybernetics, vol. 24, pp. 668-678, 1994.

[13] T. Berners-Lee, *Semantic Web Road Map*. W3C draft September 1998. Available: http://www.w3.org/DesignIssues/Semantic.html

[14] F. Monala, E. Miller, “RDF Primer,” W3C Working Draft 19 March 2002. Available: http://www.w3.org/TR/2002/WD-rdf-primer-20020319/

[15] *Apache Software Foundation*. Available: http://avalon.apache.org

[16] M. Talevi, “container overview,” 2006. available: http://docs.codehaus.org/display/PICO/3.+PicoContainer

[17] M. Pollack, *et al*, “The Spring.NET Framework Reference Documentation,” Version 1.1.2, June 12, 2008. available: http://www.springframework.net/doc-latest/reference/html/index.html

[18] D. Megginson's, *SAX*. 27-April 2004, available: http://www.saxproject.org

[19] *Java Script Kit – DOM (Document Object Model) Reference.* available: http://www.javascriptkit.com/domref/

[20] R. Johnson, *et al*, “Spring - Java/J2EE Application Framework Reference Documentation,” Version 1.2.9, 2008. Available: http://static.springframework.org/spring/docs/1.2.x/reference/index.html