A SURVEY OF RECENT EMERGING CONSISTENT PROGRAMMING TECHNIQUES FOR DISTRIBUTED ENVIRONMENT

Palanivel .K[1], Amouda .V[2], Kuppuswami .S[3]
[1] Computer Centre, Pondicherry University, Puducherry – 605014, INDIA
[2]Centre for Bioinformatics, Pondicherry University, Puducherry – 605014, INDIA
[3]Dept. of Computer Science, Pondicherry University, Puducherry – 605014, INDIA
E-mail: [1]kpalani.cce, amouda.bic

Abstract

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs. OOP techniques include important features such as encapsulation, modularity, polymorphism, and inheritance. Nowadays, several new programming paradigms address some well-known limitations of object-oriented development without forcing developers to adopt new languages or abandon the object-oriented paradigm. Considering the increasing importance and applications in the development of complex systems, this paper intends to survey the literature relevant to the recent programming approaches and point out their motivations and contributions. Each programming approaches are described in such different contexts as features/goals, modeling, languages and designing with UML support. It is also identified the commonalities and differences with OOP and point out the benefits and drawbacks of recent programming approaches. The emerging approaches include agent, aspect, component, role, service and subject. These techniques provide a better solution in developing complex systems and meet some the challenges like interoperability, managing services metadata, appropriated levels of security, etc. The discussion includes current research issues in the community. The conclusion provides a summary of this paper and future research direction follow.

Keywords: Agent, Adaptive, Component, Object, Subject.

I.INTRODUCTION

The history of programming has been a slow and steady climb from the depths of direct manipulation of the underlying machines to linguistic structures for expressing higher-level abstractions. Progress in programming languages and design methods has always been driven by the invention of structures that provide additional modularity. Subroutines assembled the behavior of unstructured machine instructions, structured programming argued for semantic meaning for these subroutines, abstract data types recognized the unity of data and behavior, and object-orientation generalized this to multiplicity of related data and behaviors.

Procedural Programming was enabled by the development of closed subroutines. The ability to isolate name spaces and to explicitly indicate information transfers through argument lists and common block declarations permits a partial separation of concerns between implementers and users. Scoping and data hiding are used technically intricate development in compiler technology, supporting the technical innovation of compiled libraries and link editors.

Structured Programming, a statement-level code organizing discipline, and the associated stepwise refinement design method were brought to general attention at about the same time as was data abstraction. All of these ideas were introduced and implemented in the design (and marketing) of the programming language Pascal. The key components of procedural and structural programming are presented in figure 1.

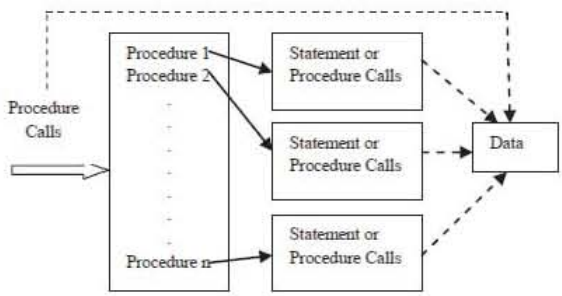

Fig. 1. Key Components of Procedural and Structural Programming

A. Object-Oriented Technology

Object-oriented programming is a code packaging convention that imposes structure on medium-to-large software systems using the notions of encapsulation and inheritance. This convention provides several useful software structuring innovations: (i) it provides a useful definition of module that is supported by syntactic structures in the programming language and checked by the compiler; (ii) it establishes an organizing principle for project decomposition (minimizing communication and dependencies among modules): (iii) it enforces a useful formal separation, supported by the programming language syntax, between architecture, implementation,

and realization. All three of these innovations became that enunciated the intellectual foundation for what became OOP. The key components of OOP are parameterized in figure 2.

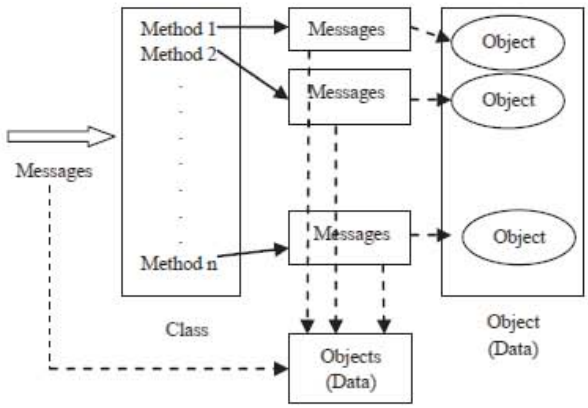

Fig. 2. Key Components of Object-Oriented Programming

B. Design Patterns and Frameworks

Most software reuse has focused on code reuse, such as reusing parts of existing applications, reusing library functions or reusing pre-built components. With the recent interest in design patterns [Gamma et al., 1995] and object-oriented frameworks, the focus is shifting away from just reusing code to reusing existing designs as well. Design patterns provide a reusable piece of a design which solves a recurring design problem in software construction. Design patterns form a cohesive language that can be used to describe classic solutions to common object oriented design problems. These patterns enable us to discuss systems of objects as quasi-encapsulated entities. By using design patterns to solve programming problems, the proper perspective on the design process can be maintained.

An object-oriented framework is the reusable design and implementation of a system or subsystem [Beck and Johnson, 1994]. It is typically implemented as a set of abstract classes which define the core functionality of the framework along with concrete classes for specific applications included for completeness. Users of the framework complete or extend the framework by adding custom application specific components or functions to produce and application.

C. Unified Modeling Language

Programming methodologies is a complex field, with many methodologies, and names, and many goals and means to reach them: structured programming, programming by refinement, program analysis and verification, refactoring, and many more. Methodologies are developed to enhance one or more programming variable: programming, program speed, reliability, conformance to user/customer needs, reusability, code reuse and sharing, information hiding, etc. Some methodologies are more formal than others, some are embodied in formal tools, programs, etc. Many methodologies involve object-oriented programming.

The Unified Modeling Language (UML) [40] is gaining wide acceptance for the representation of engineering artifacts in object-oriented software. The next step beyond objects leads us to explore extensions to UML and idioms within UML to accommodate the distinctive requirements of agents, agents, services, subjects, roles, etc.

D. Why New Programming Approaches?

Following are some the challenges for the researchers that have been introduced various new programming paradigms:

- Successful separation of concerns thus leads to ease of development, maintenance, and potential reuse. State-of-the-art software techniques already support separating concerns, for instance, by using method structuring, clean object-oriented programming, and design patterns. However, these techniques are insufficient for more complex modularization problems. A major cause for this limitation is the inherently forced focus of these techniques
- on one view of the problem; they lack the ability to approach the problem from different viewpoints simultaneously. The net result is that conventional modularization techniques are unable to fully separate crosscutting concerns.
- OOP is a fine way to create applications that perform specific tasks. However, it is a terrible architecture for creating robust network applications. Every OOP-based architecture for distributed applications (using CORBA, DCOM, or EJB) has failed to deliver on its promises and made network programming vastly more difficult than it needed to be.
- Both OO and CBD address software economies of scale that require reuse in software development. The achievement economies of scale in software production and deployment have been elusive. Object-oriented development is an enabler, component-based development mandatory. CBD provides an opportunity for greater reuse than what is possible with OO development.
- OO frameworks can be customized at open points determined by their designers. This allows designers to give careful thought to important kinds of extension and to support them especially well. However,

experience with software evolution shows that no designer can ever plan for all the kinds of extension and customization that users will eventually desire. The open points will, therefore, inevitably prove insufficient.

- Object-orientation is widely regarded as supporting extensibility, and indeed it does to some extent. For example, subclasses can always be created to extend existing classes, and frameworks are carefully-crafted collections of classes that rely on subclassing to customize them. Subclassing has the desirable property that it does not require modification or recompilation of existing source code.
- The problems of integrating existing applications are legendary. Even if the applications operate in the same domain and it looks like their functionality is complementary, the details are often sufficiently different that integration is difficult.
- In traditional class-based models, an object is linked to a single class (its generating abstract model), according to the mono-instantiation principle, and this class determines the object's structure and behavior during its lifespan.
- In an object-oriented design, objects are encapsulated entities, but are rarely self-sufficient. Although an object is fully responsible for maintaining the data it encapsulates, it needs to cooperate with other objects to complete a task. An interesting way to encode object interdependencies is through collaborations.

As a result, many programming approaches are deployed for software development whereas agent-oriented programming makes communication between autonomous components the central compositional notion; aspect-oriented programming provides more advanced modularization techniques, whereas adaptive programming is a special case of aspect-oriented programming; service-oriented programming, on the other hand, gives the developers and architects an easy way to integrate systems and they are far less complex, faster, more robust, more scalable, more easily maintained, and they are much easier to for designers to conceptualize; subject-oriented programming permits unanticipated extension: every object creation and every operation call is essentially an open point which can be extended at some later time and finally role-oriented programming .

Section 2 describes the concepts of OOP, and its benefits and drawbacks, modeling and design patterns. Section 3 introduces various programming paradigms introduced after OOP for distributed computing environment and finally section 3 concludes this paper.

II. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a programming language model organized around objects rather than actions and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. The primary goals of object oriented language development are abstraction, encapsulate, reusability and inheritance.

A. Object-Oriented Programming Languages

Simula was the first object-oriented programming language. Java, Python, C++, Visual Basic .NET and Ruby are the most popular OOP languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet. Ruby is used in many Web applications. Curl, Smalltalk, Delphi and Eiffel are also examples of object-oriented programming languages.

B. Object-Oriented Analysis and Design

Object-Oriented Analysis and Design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior. Object-Oriented Modeling (OOM) is a modeling paradigm mainly used in computer programming. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML) and the leading OO methodologies are expressed in [4].

C. Benefits and Weakness of OOP

The concepts and rules used in object-oriented programming provide these important benefits:

- The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics.
- Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data.
- The definition of a class is reusable not only by the program for which it is initially created but also by

other object-oriented programs (and, for this reason, can be more easily distributed for use in networks).

- The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

There are three main weaknesses to the current formulation of object-oriented design and programming[3]: (i) tends to encourage bottom-up design, (ii) tends to encourage toolkits of tiny classes, and (iii) provides no good answer to the problems of asynchronous I/O or concurrent processing, as occurs in user interfaces.

The way to fully realize the potential of OOP is to aim toward high-level classes of fairly large granularity that address substantial needs of identifiable clients. This development strategy has a crucial side effect that is in fact the most important potential benefit of OOP: when classes are written in this way, they are at a high enough level of abstraction that clients can contribute to the design, inspection, and testing of their software. By enlisting this whole new, well-motivated labor force, OOP might yet lead to a resolution of the continuing software crisis.

2.4 Pattern-Oriented Approach

When approaching the problem of conceiving a new system, expert designers tend to reuse solutions that have worked in the past. Design patterns are recurring organizations of classes and communicating objects that can be found in many object oriented systems, and were employed to make the design more flexible, elegant and reusable [41].

Design patterns are beneficial to the forward engineering process [10], but can also represent relevant information about the system in the reverse engineering phase. From a program understanding and maintenance perspective a pattern provides knowledge about the role of each class in the pattern organization, the reason for certain relationships among pattern constituents and with the remaining parts of the system. The discovery of patterns in a software artifact highlights the rationale of the adopted solution, represents a step in the program understanding process and improves documentation. Consequently, in maintenance, the identification of design pattern instances provides insight on software artifacts structure and reveals places where changes, reuse, or extensions can be expected.

Design patterns impose good design practices and show how the adequate use of object-orientation can help us in proper decomposition and through this enforce the reuse and maintainability. The drawbacks of design patterns are connected mainly with implementation problems and discussed below:

- Object schizophrenia – sometimes a specific design pattern rule requires splitting what intentionally was supposed to be a single object.
- Preplanning problem – applying a design pattern give us better decomposition of the problem. The reason for decomposition is usually a need for future adaptability of the component. Design patterns can help us but we need to anticipate this need in the design.
- Traceability problem (or indirection problem) – in concrete representation it is usually unclear which design pattern was used. The code implementing a pattern crosscuts the implementation and becomes scattered in the entire code.

2.5 Concurrent OOP

Concurrent programming in its many variants from multithreading to multiprocessing, distributed computing, Internet applications and Web services, has become a required component of ever more types of systems, including some that were traditionally thought of as essentially sequential. However, the industry is still looking for a good way to produce concurrent applications. The contrast with sequential

programming is stark: there, a widely accepted set of ideas (standard control and data structuring techniques, modularity and information hiding, object-oriented principles) have displaced the lower-level concepts that used to predominate, but the techniques commonly used to produce concurrent applications are elementary and often haphazard. Many developers, for example, want to add multithreading to their systems, but the proper use of multithreading remains a black art..

COOP takes object-oriented programming as given, in a form based on the concepts of Design by contract, and extends them in a minimal way to cover concurrency and distribution. To address such requirements of concurrent processing as mutual exclusion, synchronization and wait conditions, COOP gives new semantics to well known constructs (argument passing, preconditions) where the standard sequential semantics could not be applied anyway.

It takes advantage of the inherent concurrency implicit in object-oriented programming to shield programmers from low-level concepts such as semaphores, letting them instead produce concurrent applications along the same lines as sequential ones.

E. Real Computing Revolution

A real computing revolution is happening now, but the revolution involves distributed systems and bit-mapped graphics displays and the enabling technology of faster, smaller computer chips. OOP is riding the coattails of this real revolution. Bit-mapped graphics and distributed computing together allow an individual to work on many computers at once, in diverse locations, and to view text and graphics from all of those locations at once. Bit-mapped graphics made possible new user interfaces that have made computers accessible to many more people; this diffusion of computing technology is a societal revolution, not just a revolution in computing. Collaboration technology is just beginning to unlock the possibilities of the combination of bit-mapped graphics and distributed computing and is the software subfield riding the shock wave of this hardware-enabled revolution.

OOP has appropriately benefited from the real revolution. The software for controlling bit-mapped graphics and distributed computing is complex, but most of it maps nicely into classes. Class libraries of various kinds for controlling networks, distributed systems, and bit-mapped graphics have been developed experimentally and will eventually make the breakthrough technologies usable by non-expert programmers. New advances such as Java, a programming language based on C++ but that is built with an understanding of the World Wide Web, is opening a new frontier of distributed computation that is of truly revolutionary importance. The distributed computing concepts should be integrated into the field's basic software models, tremendous advantages in portability, access, and parallelis.

III. EMERGING PROGRAMMING APPROACHES

Structured programming deals with processes by procedures and controlling structures such as sequential statements, branch statements, and circulate statements. Whereas, object-oriented programming deals with processes by objects, classes, classification, inheritances and polymorphisms. A good programming approach is considered normally by its correctness, efficiency, readability, and user-friendly interface:

- It should run correctly to solve its intended problem;
- It should run efficiently to avoid wasting time and memory space;
- It should be readable to allow other programmers to understand, modify and improve it easily;
- It should have a user-friendly interface that is intuitively easy to learn and use.

This section introduces various programming approaches introduced for distributed environment after OOP. These approaches and its goals, programming languages, software engineering perspectives, modeling with UML supports, advantages and disadvantages are discussed below:

A. Component-Oriented Programming

Component-Oriented Programming (COP) has achieved wide acceptance in the domain of software engineering by improving productivity, reusability and composition. More recently, the COP languages ComponentJ [6] and ACOEL [7] extend a Java-like base language to explicitly support component composition. These languages can be used to express components and static architectures. However, neither language makes dynamic architectures explicit, nor neither enforces communication integrity

This success has also encouraged the emergence of a plethora of component models. These component models can now be applied at any software level, from operating systems to middleware (e.g., OpenCOM, Fractal, to applications (e.g., EJB, CCM, SCA).

Component-Based Software Engineering (CBSE), by contrast, makes no such assumptions, and instead states that software should be developed by gluing prefabricated components together much like in the field of electronic or mechanics. It accepts that the definitions of useful components, unlike objects, can be counter-intuitive.

The component diagram's or UML main purpose is to show the structural relationships between the components of a system. In UML a components represented implementation items, such as files and executables. Also, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces. In component-based development (CBD), component diagrams offer architects a natural format to begin modeling a solution. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

Usually, each of these component models defines their own abstract model and programming model. The abstract model defines the general concepts provided by the component model (e.g., component, port/interface, binding/ connection, composition/ assembly). The programming model applies these concepts to a particular programming language, while introducing some technical code specific to the component model. Thus, this technical code is tangled with the business code of the application.

Furthermore, if the abstract models of existing component models are quite similar; their programming models can differ a lot. This drawback limits the reuse and composition of components implemented with different programming models.

B. Agent-Oriented Programming

Agent-oriented programming relies on the assumption, that a complex distributed software system can be programmed as a set of communicating, interacting, knowledge base entities called (software) agents. The essential components of agent-oriented programming are shown in figure 3. AOP can be seen as an extension of object-oriented programming. The differences between agents and subjects are shown in Table 2.

Table 1. Difference between Agents and Objects

Features	*Agents*	*Objects*
Entity	Active	Passive
Behavior	Autonomous	Persistence
Control	Blackbox	Whitebox
Structure	Beliefs & Commitments	cannot

Agent Communication Languages: In addition, agents support high-level interaction (using agent-communication languages) between agents based on the "speech act" theory as opposed to ad-hoc messages frequently used between objects [8], examples of such languages are FIPAACL and KQML[9].

The agent-oriented (AO) approach promises the ability to construct flexible systems with complex and sophisticated behavior by combining highly modular components. The availability of agent-oriented development toolkits has allowed the technology to be assessed for industrial use. Agent-orientation is thus a paradigm for analysis, design and system organization. An agent- oriented modeling language must provide primitives for describing these higher-level structures, the inspiration for which derives from cognitive psychology and social modeling via artificial intelligence.

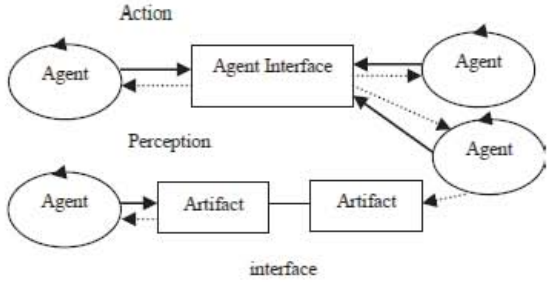

Fig. 3. Architecture of Agent-Oriented Programming

Some methodologies like MESSAGE, Gaia, Multiagent Systems Engineering (MaSE) and MAS-CommonKads [1,2] supports both the micro-level and macro-level of agent development. The motivation behind these methodologies is that existing methodologies fail to represent the autonomous and problem-solving nature of agents; they also fail to model agents' ways of performing interactions and creating organizations. The Agent-Object Relationship[10] modeling approach in the design of information systems.

Agent UML synthesizes a growing concern for agent-based software methodologies with the increasing acceptance of UML for object-oriented software development. The AUML [11] encourage CASE-tools vendors to support an agent-oriented notation in their products.

C. Aspect-Oriented Programming

Aspect-Oriented Programming centers on constructs called aspects, which treat concerns of a separate set of objects, classes, or functioning. Although commonly associated with OOP, aspect-oriented programming concepts need not confine themselves to the OOP world. Any program has principled points (join points) where programmers can identify and modify the program semantics. In AOP, programmers use a language-feature called a pointcut to specify join points, and advice (code like methods or functions) to specify the behavior to join at those points. Some variants of AOP also enable programmers to extend the types in the system. Together, these features enable aspects to implement behavior for concerns that crosscut the core concern of the application.

In OOP, an object is a unique concrete instance of an abstract data type, a class whose identity is separate from those other objects, although it can communicate with them via hierarchy. Aspect-oriented programming has the design goal of separating these concerns. The steps to successful aspect-oriented programming comprise: (i) the designer is a broad term from a person who designs any of a variety of things. This usually implies the task of creating or being creative in a particular area of expertise. Often times it is used to reference to someone who draws or in some ways defines and separate the concerns, core and crosscutting, (ii) the software developer, one who does computer programming or designs the system to match the requirements of a systems analyst.

Aspect-oriented programming includes AspectJ, Asepct C++ and JBOSS. Aspects emerged out of OOP and have functionality similar to using meta-object protocol. Aspects relate closely to programming concepts like subjects, mixins, and delegation.

Aspect-oriented software development (AOSD) is an emerging technology promoting advanced separation of concerns in software engineering. AOSD techniques allow one to modularize crosscutting concerns into separate "aspects" of a system and integrate those aspects with other kinds of modules throughout the software development lifecycle.

Aspect-oriented modeling (AOM) is a critical part of AOSD that focuses on techniques for identifying, analyzing, managing and representing crosscutting concerns in software design and architecture, while filling the gap between aspect-oriented requirements engineering and aspect-oriented programming. Theme/UML [37] is an extension of UML for aspect-oriented modeling..

The AOP helps to manage the cross-cutting functionality in the system by encapsulating the otherwise dispersed functionality into well defined modules. It adds new or to modify existing functionality in the base program. Also be used with software testing. The AOP is depending on the aspect language implementation, it can be used to modify method call parameters and return values; in some cases even classes' attribute values. While this opens new possibilities it may increase the complexity of the system and in worst case, lead to problems that are hard to track.

D. Adaptive Programming

Adaptive Programming (AP) is the special case of aspect-oriented programming where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies [42]. A traversal strategy may be viewed as a partial specification of a graph pointing out a few cornerstone nodes and edges. A traversal strategy crosscuts the graphs it is intended for, only mentioning a few isolated nodes and edges. Traversal strategies may be viewed as regular expressions specifying a traversal through a graph As an application of the above distinction, we can say that AP is a special case of AOP where some of the crosscutting is expressed adaptively using strategies to embed small graphs into large graphs. A key feature of this embedding is that it is specified by hiding the details of the large graphs which makes the graph embeddings 'adaptive' in that they work for a large family of large graphs

E. Service-Oriented Programming

The existing component models prescribed that programming problems can be seen as independently deployable black boxes that communicate through contracts. The traditional client- server model often lacks well-defined public contracts that are independent of the client or server implementation. This has made the client-server model brittle. In Service- Oriented Programming (SOP), components publish and use services in a peer-to-peer manner. In SOP a client is not tied to a particular server. Instead, service providers are interchangeable.

Service-Oriented Programming is a paradigm for distributed computing that supplements OOP. SOP focuses on what things can do. SOP builds on top of OOP, allowing services to be built using OO techniques. These services themselves provide increased reuse of the business logic, by allowing the service to be used in diverse applications. SOP approach focuses on the application's functionality, or in other words, what the application does. The key components of service-oriented programming is shown in figure 4.

SOP supports the basic programming constructs for sequencing, selection and iteration as well as built-in, advance behavior. Furthermore, SOP supports semantic constructs for automatic data mapping, translation, manipulation and flow across inner services of a composite service. In SOP, runtime properties stored on the service interface metadata serve as a contract with the service virtual machine.

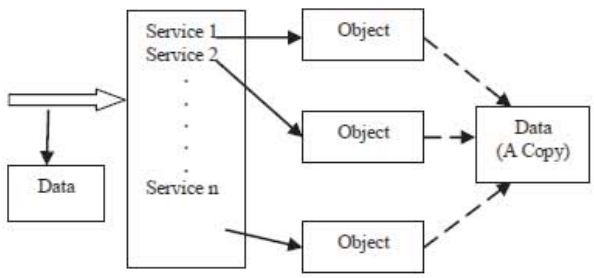

Fig. 4. Components of Service-Oriented Programming

Service-oriented architectural frameworks such as Microsoft .NET™, Cooltown, Sun's Jini™ Network Technology and Openwings™ are used for building systems spontaneously. These frameworks make it possible to build a system out of a network of services. Services can be added or removed from the network, and new clients can find existing services. The service-oriented modeling framework (SOMF) evocates that all software assets that we have been designing, architecting, constructing, and deploying to production environment are actually Services.

F. Subject-Oriented Programming

Subject-oriented programming refers to a method of programming that supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with glue or adapter subjects). The flexibility of subject

composition introduces novel opportunities for developing and modularizing object-oriented programs.

Subject-oriented programming-in-the-large involves dividing a system into subjects and writing rules to compose them correctly. It complements OOP, solving a number of problems that arise when OOP. SOP addresses some well-known limitations of object-oriented development without forcing developers to adopt new languages or abandon the object-oriented paradigm. These limitations include weaknesses in non-invasive system extension and evolution, large-scale reuse and integration, system decomposition and multi-team and decentralized development [39].

SOP attempts to solve problems surrounding the cognitive abilities of programmers by creating subjects, which by their nature are not that different, conceptually, from aspects. Whereas the early efforts in aspects used the concept of a code weaver assembling source prior to compile, the SOP paradigm relied on a compositor, so named due to the paradigm's heavy reliance on class composition. In subject-oriented programming, a subject is a collection of classes or class fragments whose class hierarchy models its domain in its own, subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. SOP allows object-oriented code to be decomposed into subjects in similar fashion. It reduces the monolithic nature of the design and allows for concurrent development, while subject-oriented composition provides a powerful mechanism for integration, evolution, customization adaptation and improved reuse.

Hyper/J supports multi-dimensional separation of concerns for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Adaptive programming has many of the same goals as subject-oriented programming, and is related in some interesting ways [38]. A subject-oriented programming approach provides mechanisms for arranging UML classes by concern, together with a profile for representing requirements in the UML, would facilitate the mapping from one domain to the other.

G. Role-Oriented Programming

For complex information systems, related to biology, geographical information, multimedia, etc. objects of these domains are evaluative, and related applications require more and more often taking into account this functionality, in the same way that reusability and extensibility currently are. The analysis and modeling, a way to express these objects should be persistent, and on the other hand be able to evolve through their life. This can be realized by allowing them to dynamically gain or lose some pieces of structure or behavior aspects. This capability is practically added by taking into account the role notion as a complement of the notions of concept or class.

Role-oriented programming is a form of computer programming aimed at expressing things in terms which are analogous to our conceptual understanding of the world. This should make programs easier to understand and maintain. The main idea of role-oriented programming is that humans think in terms of roles. This claim is often backed up by examples of social relations. In traditional class-based models, an object is linked to a single class (its generating abstract model), according to the mono-instantiation principle, and this class determines the object's structure and behavior during its lifespan.

Roles allow objects to evolve over time, they enable independent and concurrently existing views (interfaces) of the object, explicating the different contexts of the object, and separating concerns. Generally roles are a natural element of our daily concept forming. Roles in programming languages enable objects to have changing interfaces, as we see it in real life - things change over time, are used differently in different contexts, etc. In the older literature and in the field of databases, it seems that there has been little consideration for the context in which roles interplay with each other. Such a context is being established in newer role- and aspect-oriented programming languages such as Object Teams [12]

Role-Oriented Modeling to specify these contexts in the domain model, where each role specifies a certain context plus the associated relationships and attributes. In a way, such role models can be seen as independent conceptual models, and for each context we can have one role model, plus possibly one (usually quite simple) class-based model specifying characteristics common to all contexts. This additional dimension in conceptual modeling, which reflects navigational contexts already in the domain model, can lead to clearer and more focused models and to conceptual models which are better customized to different domain contexts.

The Object-oriented Role Analysis Method (OORAM), Object Oriented Role Analysis, Synthesis and Structuring (OORASS) and Object Role Modeling (ORM) [26] in the field of software engineering is a method for conceptual modeling, and can be used as a tool for information and rules analysis.

The advantages of roles in modeling and implementation are as follows: Roles allow objects to

evolve over time, they enable independent and concurrently existing views (interfaces) of the object, explicating the different contexts of the object, and separating concerns. Generally roles are a natural element of our daily concept forming. Roles in programming languages enable objects to have changing interfaces, as we see it in real life - things change over time, are used differently in different contexts, etc.

IV. CONCLUSION

This paper started with introduction to procedural programming and structural programming and drawbacks of these programming languages. This paper then introduced OOP approaches and its benefits and drawbacks. To overcome these problems several programming approaches have been conceived from OOP techniques. A brief survey was taken with respect to quality, consistent and relation with OOP under distributed computing/environment.

Components improve the reuse and consequently improved the productivity gains. The software agents provide a means to complex social interactions with their environment. Separation of concerns provides a way to encapsulate capabilities that the crosscut design such as many non-functional capabilities. Recently, services give the developers and architects an easy way to integrate systems with less complex, faster, more robust, more scalable, more easily maintained, and they are much easier to for designers to conceptualize. Subjects permit decentralization of class definitions that reduces the need for costly negotiation between, and centralized management of, object-oriented application developers and class providers. Each programming techniques concludes with their merits and demerits.

In future it would be very to extend this survey to include emerging software architectures and dynamically adaptable system architecting approaches. The environment in which software are executing today have increased considerable in complexity and software may need to adapt themselves at run-time to change the security level, memory usage, and so forth.

ACKNOWLEDGMENT

We would like to thank K. Sakthiavathy Department of Computer Science, Pondicherry University for constant support in this research work. We are grateful to our respected Marie Stanislas Ashok, Systems Manager & Head, Computer Centre for providing us with all the facility that were required in this research work.

REFERENCES

[1] Michael Wooldridge, Nicholas R. Jennings, David Kinny, The Gaia Methodology for Agent-Oriented Analysis and Design, Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, 2000.

[2] Scott A. DeLoach. Multiagent Systems Engineering of Organization-based Multiagent Systems. 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05). May 15-16, 2005, St. Louis, MO. Springer LNCS Vol 3914, Apr 2006, pp 109 - 125.

[3] James M. Coggins, George H. Jacoby and Jeannette Barnes, eds. Subject-Oriented Programming, Astronomical Data Analysis Software and Systems VASP Conference Series, Vol. 101, 1996

[4] Analía Amandi, Ramiro Iturregui, Alejandro Zunino, Object-Agent Oriented Programming, Universidad Nacional de Centro de la Provincia de Buenos Aires Facultad de Ciencias Exactas – Argentina.

[5] Federico Bergenti and Agostino Poggi, Agent-oriented Software Construction with UML, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma, , Parma, Italy.

[6] J. C. Seco and L. Caires. A basic model of typed components. In Proceedings of the 14th European Conference on Object-Oriented Programming, pages 108–128, 2000.

[7] C. Sreedhar. Programming software components using ACOEL. Unpublished manuscript, IBM T.J. Watson Research Center, 2002.

[8] Labrou Y., Finin T. and Peng Y. Agent Communication Languages: The Current Landscape. IEEE Intelligent Systems, 14(2), March/April 1999.

[9] Lind J. Issues in Agent-Oriented Software Engineering. The First International Workshop on Agent-oriented Software Engineering (AOSE-2000), 2000.

[10] Wagner G. Agent-Object-Relationship Modeling. In Proc. of Second International Symposium - from Agent Theory to Agent Implementation together with EMCRS 2000, April 2000.

[11] Federico Bergenti and Agostino Poggi, Agent-Oriented Software Construction with UML Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma Parco Area delle Scienze, Italy.

[12] Stephan Herrmann, Christine Hundt, Marco Mosconi, ObjectTeams/Java Language Definition, version 1.1, 2008, http://www.objectteams.org.

[13] C. Vecchiola, A. Gozzi, M. Coccoli, A. Boccalatte, An Agent Oriented Programming Language Targeting the Microsoft Common Language Runtime, University of Genova, DIST Via Opera Pia, 13 – 16145 Genova, Italy.

[14] Analía Amandi, Ramiro Iturregui, Alejandro Zunino, Object-Agent Oriented Programming, Universidad Nacional de Centro de la Provincia de Buenos Aires Facultad de Ciencias Exactas –Tandil - Buenos Aires – Argentina.

[15] Juval Löwy, Design and Build Maintainable Systems Using Component-Oriented Programming, Programming .NET Components,

[16] Nicolas Pessemier, Lionel Seinturier, A Safe Aspect-Oriented Programming Support for Component-Oriented Programming, Thierry Coupaye(2), Laurence Duchien, INRIA Futurs - LIFL, Project Jacquard/GOAL, 59655 Villeneuve d'Ascq, France.

[17] Schauerhuber W. Schwinger E. Kapsammer, W. Retschitzegger, M. Wimmer, Aspect-Oriented Modeling of Ubiquitous Web Applications: The aspectWebML Approach, Business Informatics Group, Vienna University of Technology, Austria.

[18] Romain ROUVOY,, Philippe MERLE, Leveraging Component-Oriented Programming with Attribute-Oriented Programming, JACQUARD Project – INRIA Futurs LIFL – University of Lille 1 59655 Villeneuve d'Ascq Cedex, France.

[19] Hedley Apperly, Introducing Component-Based Development, Kruchten P, The Rational Unified Process, An Introduction, Addison Wesley, 1999.

[20] Researchwhitepaperseries: Developer's Introduction to Resource-Oriented Computing, July, 2007.

[21] Heidrun Allert, Peter Dolog, Wolfgang Nejdl, Wolf Siberski Friedrich Steimann, Role Oriented Models for Hypermedia Construction Conceptual Modeling for the Semantic Web, Learning Lab Lower Saxony, University of Hannover, Expo Plaza 1, 30539 Hannover, Germany.

[22] Yannis Smaragdakis, Don Batory, Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, April 2002, Pages 215–255.

[23] Mayur R. Mehta, Sam Lee, Jaymeen R. Shah, Service-Oriented Architecture: Concepts and Implementation, Proc ISECON 2006.

[24] W.T. Tsai and Yinong Chen, Gary Bitter and Dorina Miron, Introduction to Service-Oriented Computing, Technology Based Learning and Research, College of Education, Arizona State University.

[25] William Harrison, Tim Walsh, Service-Oriented Programming – Supporting Emerging Variations of Object-Oriented Programming, Department of Computer Science, Trinity College Dublin 2, Ireland.

[26] Guy Bieber, Jeff Carpenter, Introduction to Service-Oriented Programming (Rev 2.1), Motorola ISD.

[27] Michel Deriaz and Giovanna Di Marzo Serugendo, Semantic Service Oriented Architecture, University of Geneva, Switzerland, November, 2004.

[28] Understanding Service-Oriented Architecture: An introductory overview, BT, J.P. Morgan Chase & Co., Meridian Health Care Management and Union Bank of California. http://www.versata.com .

[29] Harold Ossher and William Harrison, Frank Budinsky and Ian Simmonds, Subject-Oriented Programming: Supporting Decentralized Development of Objects IBM Thomas J. Watson Research Center Yorktown Heights, NY 10598.

[30] Harold Ossher and William Harrison, Developing Adaptable Software with Subject-Oriented Programming, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

[31] Juha Savolainen, Improving product line development with subject-oriented programming ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering Helsinki University of Technology, Finland.

[32] Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr, Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, Proceedings of: Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1999.

[33] Wai-Ming Ho , Jean-Marc Jézéquel, François Pennaneac'h and Noël Plouzeau Irisa, A Toolkit for Weaving Aspect Oriented UML Designs INRIA & University of Rennes, Campus de Beaulieu, AOSD 2002, Enschede, The Netherlands. Copyright 2002 ACM 1-58113-469-X/02/0004. [2].

[34] Terry Halpin, Object-Role Modeling (ORM/NIAM), Handbook on Architectures of Information Systems, Springer-Verlag, Berlin, 1998,

[35] Amund Tveit, A survey of Agent-Oriented Software Engineering, Norwegian University of Science and Technology, 2001.

[36] Fichman, Robert and Kemerer, Chris. Object-Oriented and Conventional Analysis and Design Methods - Comparison and Critique. IEEE-Comp, Oct, 1992, pp 22-39.

[37] Cormac Driver, Vinny Cahill, Siobh Clarke, "Separation of Distributed Real-Time Embedded Concerns with Theme/UML," pp.27-33, 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software, 2008.

[38] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns.In Proc. of ICSE, 1999.

[39] Harold Ossher, Peri Tarr, International Conference on Software Engineering Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States , Pages: 687 - 688 , 1999.

[40] Booch G, Rumbaugh J, Jaeobson I.The Unified Modeling Language User Guide Reading. MA: Addison-Wesley, 1998.

[41] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

[42] K. Lieberherr, From transience to persistence in object-oriented programming: architectures and patterns, ACM Computing Surveys (CSUR), Volume 28 , Issue 4es, 1996.

K. Palanivel, received his B.E in Computer Science & Engineering and M. Tech. in Computer Science & Engineering from Bhrathiar University, Coimbatore and Pondicherry University, Puducherry in 1994 and 1998 respectively. He cleared the National Level Eligibility test conducted by University Grants Commission, New Delhi in 1998. Presently he is pursuing his Ph.D. in Computer Science and Engineering in Pondicherry University since 2008. He joined as Technical Assistant in 1995. Presently he is working as Systems Analyst in Computer Centre of Pondicherry University. His field of interest is Software Engineering, computer networks and Design Patterns. He became the member of Indian Society of Technical Education in 2004. He is having more than 8 years of experience in the field of computer applications, teaching and research in Pondicherry University.