# Agent Oriented Software Engineering: A So-Lution in Developing Multi-Agent Systems.

**Elizabeth Ndunge Benson**

Phd Information Technology Student, Jomo Kenyatta Univerity, Kenya

***Abstract-*** Agent Oriented Software Engineering (AOSE) is an exciting and promising approach for solving complex and real world problems. It is crucial for industrial and commercial application as these systems are required to operate in increasingly complex, open, dynamic, unpredictable and inherently high interactive environments.

In this paper, I will outline the concept of agent oriented software engineering and explain the need of AOSE as a programming paradigm. The paper also presents a discussion on challenges of developing multi-agent systems and how agent oriented software engineering can be applied to solve these challenges.

***Index Terms***- Agent-Oriented Software Engineering (AOSE); Information Communication Technology (ICT); Object Oriented Programming; Multi-Agent Systems (MASs); Multiagent, Generic Architecture for Information Availability (Gaia), Multiagent Systems Engineering Methodology (MaSE)

## I. INTRODUCTION

An agent is a computer system that is situated in its environment and is capable of autonomous action in order to meet its design objectives (Leon florin, 2010). Intelligent agents retain the properties of autonomous agents, and in addition show a flexible behaviour, characterised by:

• Reactivity: the ability to perceive their environment, and respond in a timely manner to changes that occur in it;

• Pro-activeness: the ability to exhibit goal-directed behaviour by taking the initiative;

• Social ability to interact with other agents and possibly human users.

(V. Dignum and F. Dignum ,2010), The most important difference between traditional object-oriented programming and agent-based programming is the freedom of an agent to respond to a request. When an object receives a message, i.e. one of its methods is called, the control flow automatically moves to that method. When an agent receives a message, it can decide whether it takes a corresponding course of action or not.

Because of the distributed, autonomous and cooperative features, the design and implementation of algorithms in a multi-agent framework raise a different class of problems from the design and implementation in an object- oriented environment.

This paper is structured in three sections. Section one presents the need for agent oriented software engineering (AOSE), section two outlines the challenges of developing multi-agent systems and section three explains how agent oriented software engineering addresses the challenges of developing multi-agent systems.

## II. LITERATURE REVIEW

**SECTION ONE**

**THE NEED FOR AGENT ORIENTED SOFTWARE ENGINEERING.**

An agent is a software entity exhibiting the following characteristics in pursuit of its design objectives (Huib Aldewereld and Virginia Dignum, 2011)

i. Autonomy. An agent is not passively subject to a global, external flow of control in its actions. That is, an agent has its own internal thread of execution, typically oriented to the achievement of a specific task, and it decides for itself what actions it should perform at what time.
ii. Situatedness. Agents perform their actions while situated in a particular environment. The environment may be a computational one (e.g., a Website) or a physical one (e.g., a manufacturing pipeline), and an agent can sense and effect some portions it.
iii. Proactivity. In order to accomplish its design objectives in a dynamic and unpredictable environment the agent may need to act to ensure that its set goals are achieved and that new goals are opportunistically pursued when-ever appropriate.

(Bass et al. 2009), traditional object-based computing promotes a perspective of software components as “functional” or “service-oriented” entities that directly influences the way that software systems are architected. Usually, the global design relies on a rather static architecture that derives from the decomposition (and modularization) of the functionalities and data required by the system to achieve its global goals and on the definition of their interdependencies. (Schwabe et al, 2010) in their research argue that:

i. Objects are usually considered as service providers, responsible for specific portions of data and in charge of providing services to other objects
ii. Interactions between objects are usually an expression of inter-dependencies; two objects interact to access services and data that are not available locally;
iii. Everything in a system tends to be modeled in terms of objects, and any distinction between active actors and passive resources is typically neglected.

( Zambonelli and Parunak, 2003) Object-oriented development, while promoting encapsulation of data and functionality and a functional-oriented concept of interactions, tends to neglect modeling and encapsulation of execution control. Some sort of "global control" over the activity of the system is usually assumed (e.g., the presence of a single execution flow or of a limited set of controllable and globally synchronized execution flows). However, assuming and/or enforcing such control may not be feasible in complex systems. Thus, rather than being at risk of losing control, a better solution would be to explicitly *delegate* control over the execution to the system components

(Parunak, 2009) Delegating control to autonomous components can be considered as an additional dimension of modularity and encapsulation. When entities can encapsulate control in addition to data and algorithms, they can better handle the dynamics of a complex environment (local contingencies can be handled locally by components) and can reduce their interdependencies (limiting the explicit transfer of execution activities). This leads to a sharper separation between the component-level (i.e., intra-agent) and system-level (i.e., inter-agent) design dimensions, in that also the control component is no longer global.

(V. Dignum, 2009)The dynamics and openness of application scenarios can make it impossible to know a priori all potential interdependencies between components (e.g.,what services are needed at a given point of the execution and with what other components to interact), as a functional-oriented perspective typically requires. Autonomous components delegated of their own control can be enriched with sophisticated social abilities, that is, the capability to make decisions about the scope and nature of their interactions at run-time and of initiating interactions in a flexible manner (e.g., by looking for and negotiating for service and data provision).

(Koen V. Hindriks, 2009)For complex systems, a clear distinction between the active actors of the systems (autonomous and in charge of their own control) and the passive resources (passive objects without autonomous control) may provide a simplified modeling of the problem. In fact, the software components of an application often have a real-world counterpart that can be either active or passive and that, consequently, is better suited to being modeled in terms of
both active entities (agents) and passive ones (environmental resources).

(Nick Tinnemeijer., 2011)Traditional object abstractions have been enriched by incorporating novel features such as internal threads of execution, event-handling, exception handling, and context dependencies and are being substituted, in architectural styles, by the higher level abstraction of self-contained (possibly active) coarse-grained entities (i.e.,components).

The researcher argues that these changes fundamentally alter the way software architectures are built, in that active self-contained components intrinsically introduce multiple loci of control are more naturally considered as repositories of tasks, rather than simply of services. Also, the need to cope with openness and dynamics requires application components to interact in more flexible ways (e.g.,by making use of external directory, lookup, and security services).

(Birna van Riemsdijk et al ,2011)Objects and components are too low a level of abstraction for dealing with the complexity of today's software systems, and miss important concepts such as autonomy, task-orientation, situatedness and flexible interactions. For instance, object- and component-based approaches have nothing to say on the subject of designing negotiation algorithms to govern interactions, and do not offer insights into how to maintain a balance between reactive and proactive behaviour in a complex and dynamic situations.

This forces applications to be built by adopting a functionally oriented perspective and, in turn, this leads to either rather static software architectures or to the need for complex middleware support to handle the dynamics and flexible reconfiguration and to support negotiation for resources and tasks.

(Ghassan Beydoun, 2011)An agent-oriented approach is beneficial in the below types of situations

i. Where complex/diverse types of communication are required.
ii. When the system must perform well in situations where it is not practical/ possible to specify its behavior on a case-by-case basis.
iii. Situations involving negotiation, co-operation and competition among different entities.
iv. When the system must act autonomously
v. When it is anticipated that the system will be expanded, modified or when the system purpose is expected to change.

In summary, agent-based computing promotes an abstraction level that is suitable for modern scenarios and that is appropriate for building flexible, highly modular, and robust systems.

## III. CHALLENGES OF DEVELOPING MULTI –AGENT SYSTEMS

There are two main classes of multi-agent systems.

i. *distributed problem solving systems* in which the component agents are explicitly designed to cooperatively achieve a given goal;
ii. *open systems* in which agents are not co-designed to share a common goal, and have been possibly developed by different people to achieve different objectives. Moreover, the composition of the system can dynamically vary as agents enter and leave the system.

(Carles Sierra et al, 2009) One of the major problems in the field of multi-agent systems is the need for methods and tools that facilitate the development of systems of this kind. If the agents are considered to have the potential to be used as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically applicable to this paradigm.

Their research argued that the acceptance of multi-agent system development methods in industry and/or enterprise depends on the existence of the necessary tools to support the analysis, design and implementation of agent-based software.

(Hongyuan Sun et al, 2009) The major challenges of developing multi-agent systems are summarized by the below questions

i. How to formulate, describe, decompose and allocate problems and synthesize results among a group of intelligent agents?
ii. How to enable agents to communicate and interact? What communication language and protocols do we use? How can heterogeneous agents interoperate? What and when can they communicate? How can we find useful agents in an open environment?
iii. How to ensure that agents act coherently in making decisions or taking action, accommodating the nonlocal effects of local decisions and avoiding harmful interactions? How do we ensure the MAS do not become resource bounded? How do we avoid unstable system behavior?
iv. How to enable individual agents to represent and reason about the actions, plans, and knowledge of other agents to coordinate with them; how do we reason about the state of their coordinated process (for example, initiation and completion)?
v. How to recognize and reconcile disparate viewpoints and conflicting intentions among a collection of agents trying to coordinate their actions?
i. How to design technology platforms and development methodologies for MASs?

**(**Hongyuan Sun, 2010**)** the challenges in developing multi-agent systems include the following.

i. There is no agreement on how to identify and characterize roles in the analysis phase and agent types in the design phase.
ii. The concepts used in the methodologies, like responsibility, permission, goals and tasks do not have a formal semantics or explicit formal properties. This becomes an important issue when these concepts are implemented; implementation constructs do have exact semantics.
iii. There is a gap between the design models of the methodologies and the existing implementation languages. It is unreasonable to expect a programmer to implement the proposed complex design models. To bridge the gap, a methodology should either intro- duce refined design models that can be directly implemented in an available programming language, or use a dedicated agent-oriented programming language which provides constructs to implement the high-level design concepts.
iv. The methodologies that include an implementation phase, such as Tropos, propose an implementation language in which it is not explained how to implement reasoning about beliefs, reasoning about goals and plans, reasoning about planning goals, or reasoning about communication.
v. It is widely recognized that an agent may enact several roles. None of the methodologies addresses the implementation of agents that need to represent and reason about playing different roles.
vi. The methodologies, with the exception of the organizational rules ignore organizational norms and do not explain how to specify and design them or even how to do implementation.
vii. Open systems are not really supported. The methodologies implicitly suppose that agents are purposely designed to enact roles in a system. But as soon as agents from the outside may enter the analysis, design and implementation needs to treat agents as given entities.
viii. In the analysis, methodologies do not consider the environmental embedding of a system. The structure of the organization in which a system will be embedded, has a large influence on the type of organizational structure of the system, at least when it interacts with more than one person.

(K. S. Decker et al 2010), the implementation is developed completely manually from the design. This creates the possibility for the design and implementation to diverge, which tends to make the design less useful for further work in maintenance and comprehension of the system.

Their research argues that although present AOPLs provide powerful features for specifying the internals of a single agent, they mostly only provide messages as the mechanism for agent interaction. Messages are really just the least common denominator for interaction, and, especially if flexible and robust agent interactions are desired, it is important to design and implement agent interactions in terms of higher-level concepts such as social commitments, delegation of goal/task, responsibility, or interaction goals . Additionally, AOPLs are weak in allowing the developer to model the environment within which the agents will execute.

(P. Yolum, et al 2011), In most of the practical approaches for verification of multi-agent systems, verification is done on code. While this has the advantage of proving properties of the system that will be actually deployed, it is also often useful to check properties during the system design, so more work is required in verification of agent design artifacts. In fact, all the work on model checking for multi-agent systems is still in early stages so not really suitable for use on large and realistic systems. **(**Juan, T., Pearce, A., et al, 2002 **)** believes that the major challenges of designing a multi-agent system include:

1. How to decompose problems and allocate tasks to individual agents.
2. How to coordinate agent control and communications.
3. How to make multiple agents act in a coherent manner.
4. How to make individual agents reason about other agents and the state of coordination.
5. How to reconcile conflicting goals between coordinating agents.
6. How to engineer practical multiagent systems

## IV. HOW AGENT ORIENTED SOFTWARE ENGINEERING ADDRESSES THE CHALLENGES OF DEVELOPING MULTI-AGENT SYSTEMS

(Wooldridge, M., Jennings, 2000) the focus of multi-agent programming languages can be on individual agents, multi-agent organizations, multi-agent environments, or their combinations. Programming languages focusing on individual agents are concerned with issues such as autonomy of agents, reactive behaviors, social awareness, reasoning about norms and organizations, communication and interaction with other agents, and capabilities to sense and act in a shared environment.

(Zambonelli, F., Jennings, 2000), Multi-agent organizations can be implemented either endogenously or exogenously, i.e., either individual agents are implemented in terms of social and organizational concepts, or organizations are implemented as computational entities outside agents controlling their behaviors. Their paper argued that programming languages that support the implementation of multi-agent environments need to provide programming constructs to implement sense and act abilities of agents, tools, artifacts, services, and resources that can be used by agents.

(Bresciani, P., Giorgini, P et al, 2004 ), Some multi-agent programming languages come with formal and computational semantics, an implemented interpreter, or both. The existence of formal semantics for multi-agent programming languages is essential for a better understanding of the programming constructs and the verification of multi-agent programs. Without a formal semantics one cannot guarantee the correctness of programs.

Multi-agent programming languages can be analyzed by means of general programming principles they respect and support. Examples of such principles are modularity, encapsulation, reuse, separation of concerns, recursion, abstraction, exception handling facilities, and support for legacy codes. Of course, the very concept of agent itself supports some of these principles such as encapsulation and reuse.

Dam, K. H., & Winikoff, M. (2003). the idea of implementing environments and organizations separately support the separation of concerns principle. Multi-agent programming languages can be used in a more efficient and effective manner when they support such principle at different levels. For example, at the individual agent level, modularity can be used to support the implementation of different functionalities and roles, recursion can be used to implement complex plans, and exception handling can be used to implement plan failure operations.

Danny Weynes (2008), Multi-agent programming languages can be evaluated in terms of the functionalities provided by their corresponding integrated development environments. An integrated development environment supports the development of multi-agent programs by means of functionalities such as editing tools allowing easy browsing of codes, debugging tools that help to localize errors and anomalies, and automatic testing tools allowing the automatic generation of test cases for specific part of the programs. The main difficulty for such an integrated development environment is the distributed nature of multi-agent programs, e.g., how to browse through a program that is distributed by means of agents, modules, environment, and organization programs. Debugging is even harder as it is not clear how to debug one single agent when the execution of the agent depends on the execution of other agent programs, the environment program, and the organization program

## V. AGENT ORIENTED METHODS AND HOWTHEYADDRESS THE CHALLENGES OF DEVELOPING MULTI-AGENT SYSTEMS.

### 1. GAIA

Gaia comprises an analysis and design phase and explicitly refrains from including an implementation phase. Jurgen Lind (2010),

Analysis is driven by a set of requirements and aims at understanding the system and its structure. It provides two models: a role model and an interaction model. The role model specifies the key roles in the system and characterizes them in terms of permissions (the right to exploit a resource) and responsibilities (functionalities). The interaction model captures the dependencies and relations between roles by means of protocol definitions. Gaia is only concerned with the society level; it does not capture the internal aspects of agent design.

Virginia Dignum, Hulb Gideweld and Frank Dignum (2012), The design phase provides three models: the agent model, the service model, and the acquaintance model. The agent model identifies so called agent types, which are sets of roles. The service model identifies the services (or functions) associated with a role. Finally, the acquaintance model identifies the communication links between agent types. This model can be used to detect potential communication bottlenecks. The method has been extended with a model of organizational rules and organizational structure. This allows the developer to specify global rules that the organization should respect or enforce.
Like norms, such rules are typically formulated at a high conceptual level. Little is said about ways of implementing them. The interaction of agents with the environment is not treated separately.

V. Julian and V. Botti (2012), Gaia does not support the implementation phase.Therefore it is difficult to check whether agents really implement a certain role. Especially when different roles containing several responsibilities are joined into an agent type. Although permissions seem to be norms, it is unclear how they are actually translated to the system itself. Should the agent itself make sure that it will only perform actions it is permitted to do? Do the resources force the agent to refrain from forbidden actions? Does the agent *know* about its permissions? Finally, Gaia cannot support open agent systems, because it does not treat agents as given entities.

### 2. AII METHODOLOGY

Franco zambonelli, Nicholas R. Jennings and Michael wooldrige (2010), The AAII methodology makes no distinction between the analysis and design phase.

The methodology generates a set of models, based on existing object-oriented models. From an external viewpoint (inter-agent), the system is decomposed into agents, which are modeled as complex objects characterized by their purpose, their

responsibilities, the services they perform, the information they require and maintain, and their interaction.

N.R Genza and E.S Mighele (May 2013), AAII is one of the few approaches that takes the intra agent perspective seriously.

Roles can be considered as responsibilities, which can in turn be considered as sets of services. Services are activities that are not natural to decompose any further. Hardly any reasoning is required by the agents. The methodology is very practice-oriented which leads to graphical models, but without much semantics of the concepts. It is left to the programmer to fill in the gaps.

Like Gaia, AAII does not support open agent systems. The organization of the system is almost completely hierarchical in a truly object-oriented manner. No norms or rules are specified as such.

## 3. SODA

The SODA methodology has a clear distinction between analysis and design. The methodology is only concerned with the inter-agent viewpoint.

N.R Genza and E.S Mighele (May 2013), The analysis phase provides three models: the role model, the resource model, and the interaction model. The role model defines global application goals in terms of the tasks to be achieved. Tasks can be individual or social. Individual tasks are assigned to roles while social tasks are assigned to groups. A group is an abstract concept that can be analyzed as a set of roles. The resource model captures the application environment and identifies the services that are available. The resource model defines abstract access modes (permission), modeling the different ways in which the services associated with a resource can be exploited by agents. The interaction model defines the interaction between roles, groups and resources in terms of protocols.

Leon florin (2010) The design phase refines the abstract models from the analysis phase and provides three models: the agent model, the society model and the environment model. The agent model specifies the mapping from roles onto agent classes. An agent class is characterized by the tasks, permissions and interaction rules associated to a role. It also specifies the cardinality (the number of agents in that class), their location (fixed for static agents and variable for mobile agents) and their origin (inside or outside the system). The society model specifies a mapping from groups onto societies of agents. An agent society is characterized by the social tasks, the set of permissions, the participating social roles, and the interaction protocols. Finally, the environment model specifies a mapping from resources onto infrastructure classes.

Infrastructure classes are characterized by the services, the access modes for roles and groups, and the protocols for interacting with the environment.

SODA is a very usable development methodology. The inter-agent aspect is well developed. The interaction among agents, but also the interaction between agents and the environment is taken seriously. Garcia, A., Silva, V., Chavez, C., & Lucena, C. (2002). However, SODA does not specify the design of the agents themselves. Therefore it too leaves a gap between the design and implementation of the multi agent system. Due to the fact that SODA recognizes explicit organizational structures and rules, it becomes applicable for open agent systems.

Dam, K. H., & Winikoff, M. (2003Although many concepts are used for the inter-agent specification, they are not formalized. Therefore it becomes difficult to check whether agents fulfilling a role comply to all the organizational rules. Another worry is that the use of procedural specifications of behavior, like standardized tasks, will bias the design. It suggests traditional imperative programming constructs. Such a simple choice is nice, when it is enough. However, such a view may limit the potential benefits of multi-agent systems, such as flexibility and robustness, because it does not take advantage of the autonomy and possible intelligence of the agents.

## 4. Tropos

The Tropos methodology distinguishes between an early and a late requirements phase, and between architectural design and detailed design. It considers both inter-agent and intra-agent issues.

Dam, K. H., & Winikoff, M. (2003). The early requirements phase, which is based on the *i*¤ organizational modeling framework is concerned with understanding an application by studying its organizational setting. This phase generates two models: a strategic dependency model and a strategic rationale model. These models specify the relevant actors, their respective goals and their inter-dependencies. In particular, the strategic dependency model describes an ‘agreement’ between two actors: the depender and the dependee. The strategic rationale model determines through a means-ends analysis how an actor’s goals (including softgoals) can actually be fulfilled through the contributions of other actors. The late requirements phase results in a list of functional and non-functional requirements for the system.

(Bresciani, P., Giorgini, et al 2010) The architectural design defines the structure of a system in terms of subsystems that are interconnected through data, control and other dependencies. The detailed designdefines the behavior of each component. Agent communication languages like FIPA-ACL or KQML, message transportation mechanisms, and other concepts and tools are used to specify these components. Moreover, communication protocols are used to specify communication patterns among actors, as well as constraints on the contents of the messages they exchange. Finally, the internal processes that take place within an actor are specified by plan graphs.

The implementation phase maps the models from the detailed design phase into software by means of Jack Intelligent Agents. Jack extends Java with five language constructs: agents, capabilities, database relations, events, and plans. It is claimed that these constructs implement cognitive notions such as beliefs, desires, and intentions.

(Hongyuan Sun et al, 2010 )The drawbacks of Tropos are that it doesn’t have a formal semantics and therefore it is hard to specify an implementation for the design models. It also neglects the environment, and fails to notice that roles affect the access modes or permissions for executing certain actions, or for accessing resources.

Also Tropos is meant to design closed systems, in which the designer has control over the agents that enter. However, if a system would allow external agents to enter and interact,

an interface between such external agents and the environment and the other agents is required..

## 5. The Prometheus Methodology

(Garcia, A., Silva, V., et al 2002) Prometheus is a detailed process for specifying, designing, and implementing intelligent agent systems The goal in developing Prometheus is to have a process with defined deliverables which can be taught to industry practitioners and undergraduate students who do not have a background in agents and which they can use to develop intelligent agent systems Prometheus distinguishes itself from other methodologies by supporting the development of intelligent agents:

i. providing start-to-end support,
ii. having evolved out of practical industrial and pedagogical experience,
iii. having been used in both industry and academia, and, above all, in being detailed and complete

Dam, K. H., & Winikoff, M. (2003). Prometheus is also amenable to tool support and provides scope for cross checking between designs

The methodology consists of three phases: system specification, architectural design, and detailed design

## 6. OperA + Environment

(Toronto, May 2010), OperA contains three models.

The *social model* describes roles and their dependencies. Roles have objectives: the goals the organization expects an agent to fulfill when enacting that role. OperA allows for agents to have their own goals which should be combined with those of a role when enacting that role. Therefore OperA caters for open agent systems. A role can be dependent on another role to fulfill (part of) its objective.

The *interaction model* describes the process flow of the system, in terms of scenes and transitions between scenes. This is similar to the Islander approach. The scope of a role is limited to a scene. Each scene contains an abstract and declarative specification of the landmarks to be achieved during interaction. Scenes do not (have to) specify complete protocols; they specify landmarks that can be reached in many different ways. Transitions between scenes are subject to constraints, and to a temporal ordering. E.g. an agent cannot enter a 'conference presentation' scene as a presenter if its paper was never accepted.

The *normative model* contains all the different types of norms that regulate behavior in the system. For example:

(1) Norms for roles; e.g. a PC member should not review a paper submitted by another member of the research group he is working in.

(2) Norms for scenes; e.g. the reviews have to be returned to the PC chairs before a certain deadline.

(3) Norms on scene transitions; e.g. a delegate should pay the registration before coming to the conference.

(M. Birna van Riemsdijk, et al, 2009), Norms cannot be translated into a design model directly. They will be distributed over the various models of the design phase. Although normative concepts are found in most of the methodologies discussed in this paper, they are usually immediately associated with roles. They are not formulated in a general way, or associated with activities or scenes. Therefore, norms for roles already bias the design of a system. By contrast, OperA allows one to first formulate norms, and then discuss the various ways of translating them in a society.

(P. Yolum, et al 2010), The final model of the analysis phase, which is not included in OperA, is the *environment model*. In this model we specify the resources that are available for the agents, like databases, etc. We also specify the available services.

## 7. 3APL

(Carles Sierra, et al, 2009), The implementation phase is based on the 3APL language and environment. It has facilities defined in the infrastructure and environment model of the design phase, such as communication and coordination facilities, access to knowledge sources external to an agent, a way of mediating between different agents, and an underlying architecture that supports low level programming facilities, such as arithmetic and a user interface.

(Hongyuan Sun et al, 2010), Many of these facilities are accessed through the *agent management system*, based on the FIPA Agent Management Specification. The platform can be used through a graphical user interface (GUI) which enables the programmer to load agents from a library, implement and execute them, and observe their behavior.

*Communication Management* The 3APL agent platform provides communication by means of message passing. A message will be delivered by the underlying transport layer, provided the agent management system knows the identifier of the agent being addressed. The agent can be located on a different platform running on a different machine as long as the address is recognized and unique. The messages themselves have the structure of communicative acts, with a sender, receiver and a content, which is compliant with the FIPA standards for agent communication.

*Environment* In the current 3APL platform an agent can only interact with an environment through a Java class called a *plug-in*..

*Service facilitator* The platform contains a very simple service directory facility. Agents can register the services that they offer with the AMS. If they are interested in the services offered by other agents, they can query the AMS. This functionality of the agent platform may be extended in the future with more elaborate directory services (yellow pages) that allow more intelligent searching and matching.

*Agent library* The most important development support, is a library of software templates for common tasks and applications. In this library the templates for the facilitation agents belonging to different organizational structures can be found. E.g. templates for matchmaker agents, notary agents, etc. Thus it implements parts of the organizational model of the design phase. Typically a template will implement a particular kind of behavior that is part of an interaction pattern. A template consists of an initial belief base and goal base, a set of capabilities and a set of practical reasoning rules. As such they are the implementation counterparts of the agent types.

# VI. CONCLUSION

Software Agent technology has drawn much attention as the preferred architectural framework for the design of many

distributed software systems. Agent-based systems are often featured with intelligence, autonomy, and reasoning. Such attributes are quickly becoming alluring to both legacy and new systems. Agents are building blocks in these software systems, while combinations of attributes are composed to form the software entities. The more complex an Agent-based system is, the more sophisticated the methodology to design such systems must be. There are also many challenges when it comes to the development of multi-agent systems.

This paper has explained the need for agent oriented software engineering (AOSE) and in detailed how different AOSE methodologies can be applied to address the many challenges of developing multi-agent systems. Some of the core issues of developing multi-agent systems can be solved through.

i. Integrating design and code better manner
ii. Extending AOPLs with the ability to represent social aspects and the environment;
iii. Developing practical tools for verification and validation that are tailored specifically for multi-agent systems.

## REFERENCES

[1] Huib Aldewereld and Virginia Dignum. OperettA: Organization-oriented de- velopment environment. In Proceedings of the 3rd International workshop on Languages, Methodolo-gies and Development Tools for Multi-agent Systems (LADS2010@Mallow), 2011.

[2] V. Dignum, editor. Handbook of Research on Multi-Agent Systems: Semantics and Dy-namics of Organizational Models. Information Science Reference, 2009.

[3] V. Dignum and F. Dignum. Designing agent systems: State of the practice. Inter- national Journal on Agent-Oriented Software Engineering, 4(3), 2010.

[4] Koen V. Hindriks. Programming rational agents in GOAL. In Rafael H. Bordini, Mehdi Dastani, Ju¨rgen Dix, and Amal El Fallah Seghrouchni, editors, Multi-Agent Programming: Languages, Tools and Applications. Springer, Berlin, 2009

[5] Nick Tinnemeijer. Organizing Agent Organizations. SIKS Dissertation Series 2011

[6] Birna van Riemsdijk, Virginia Dignum, Catholijn Jonker, and Huib Aldewereld. Program-ming role enactment through reflection. In Proceedings of the Joint Inter- national Conference on Autonomous Agents and Multi-Agent Systems (AAMAS- 2011).

[7] M. Birna van Riemsdijk, Koen V. Hindriks, and Catholijn M. Jonker. Programming organ-ization-aware agents: A research agenda. In Proceedings of the Tenth Inter- national Workshop on Engineering Societies in the Agents' World (ESAW'09), volume 5881 of LNAI, pages 98–112. Springer, 2009.

[8] Estefan´ıa Argente, Ghassan Beydoun, Rub´en Fuentes-Fern´andez, Brian Henderson-Sellers, and Graham Low. Modelling with agents. In Marie-Pierre Gleizes and Jorge Gomez-Sanz, editors, Agent-Oriented Software engineeringX, volume 6038 of Lecture Notes in Computer Science, pages 157–168. Springer Berlin / Heidelberg, 2011.

[9] Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Sim˜ao Sichman, editors. 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2. IFAAMAS, 2009.

[10] Hongyuan Sun, John Thangarajah, and Lin Padgham. Eclipse-based prometheus design tool. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lesp´erance, Michael Luck, and San-dip Sen, editors, AAMAS, pages 1769– 1770. IFAAMAS, 2010.

[11] K. S. Decker, J. S. Sichman, C. Sierra, and C. Castelfranci, editors. Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Sys- tems, Budapest, Hungary. International Foundation for Autonomous Agents and Multiagent Systems, May 2009

[12] W. van der Hoek, G. A. Kaminka, Y. Lesp´erance, M. Luck, and S. Sen, editors. Proceed-ings of the 9th International Conference on Autonomous Agents and Multia- gent Systems, Toronto, Canada. International Foundation for Autonomous Agents and Multiagent Sys-tems, May 2010.

[13] P. Yolum, K. Tumer, P. Stone, and L. Sonenberg, editors. Proceedings of the 10th Interna-tional Conference on Autonomous Agents and Multiagent Systems, Taipei, Taiwan. International Foundation for Autonomous Agents and Multiagent Systems, May 2011.

[14] Juan, T., Pearce, A., & Sterling, L. (2002). Roadmap: Extending the gaia methodology for complex open systems. Autonomous Agents and Multi-Agent Systems.

[15] Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems, 3, 285-312.

[16] Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2000). Organizational abstractions for the analysis and design of multi-agent systems. Paper presented at the AOSE 2000

[17] Bresciani, P., Giorgini, P., Hiunchiglia, F., Mylopoulos, J., & Perini, A. (2004). Tropos: An agent-oriented software development methodology, technical report #dit-02-0015. AAMAS Journal, 8(3), 203-236.

[18] Castro, J., Kolp, M., & Mylopoulos, J. (2002). Towards requirements-driven information systems engineering: The tropos project,information systems.Elsevier, Amsterdam, The Netherlands.

[19] Dam, K. H., & Winikoff, M. (2003). Comparing agent-oriented methodologies. Paper pre-sented at the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2003),Melbourne, Australia.

[20] Garcia, A., Silva, V., Chavez, C., & Lucena, C. (2002). Engineering multi-agent systems with patterns and aspects. Journal of the Brazilian Computer Society, SBC, Special Issue on Software Engineering and Databases.

[21] Leon florin (2010) Design of a multi-agent system for solving search problems. Journal of engineering studies and research-volume 16

[22] N.R Genza and E.S Mighele (May 2013), Review on multi-agent oriented software engi-neering implementation. International journal of computer and information technology (ISSN:2279-0764) volume 62

[23] Franco zambonelli, Nicholas R. Jennings and Michael wooldrige (2010), developing multi-agent systems: The GAIA methodology

[24] V. Julian and V. Botti (2012), Developing real-time multi-agent systems

[25] Virginia Dignum, Hulb Gideweld and Frank Dignum (2012), the engineering of Multi-agent systems

[26] Jorges S. Gomez-Sanz, Ruben Fuentes, Juan paron (2011), Understanding Agent oriented software engineering methodologies

[27] Jurgen Lind (2010), Issues in Agent-Oriented software engineering

[28] Danny Weynes (2008), Future of software engineering and multi-agent systems

## AUTHORS

**First Author** – Elizabeth Ndunge Benson, Phd Information Technology Student, Jomo Kenyatta Univerity, Kenya