Agents for DDD – Back and Forth

Alessandro Ricci[1], Samuele Burattini[1],
Andrei Ciortea[2], and Matteo Castellucci[1]

[1] Dipartimento di Informatica - Scienza e Ingegneria,
Alma Mater Studiorum - University of Bologna, Cesena Campus, Italy
{a.ricci|samuele.burattini}@unibo.it
[2] School of Computer Science, University of St.Gallen, Switzerland
andrei.ciortea@unisg.ch

Abstract. Domain-Driven Design (DDD) emerged in the last two decades as an effective approach adopted especially in agile software development to tackle "the complexity at the heart of software", to quote one of its main mottos. In this paper, we are interested in exploring the bi-directional conceptual interaction between DDD and Agent-Oriented Software Engineering (AOSE) — and the synergies stemming from their fruitful integration.

Keywords: Domain-Driven Design · Agent-Oriented Software Engineering · Agents & Artifacts · JaCaMo

1 Introduction

Domain-Driven Design (DDD) was introduced about two decades ago by Eric Evans with the so-called "Blue Book" [12] — and since has become a reference approach for a large community of designers and developers in mainstream software development [34]. The original main motto of DDD, i.e. *tackling complexity in the heart of software*, sounds familiar to researchers in Agent-Oriented Software Engineering, where the capability of tackling the complexity of software systems is a main tenet for introducing agent-based approaches [19]. Complexity, though, can be tackled from different perspectives: DDD mainly concerns what is defined as *structural* complexity i.e. challenges that emerge from the inherent complexity of the entities within a domain [21]. AOSE mainly concerns *dynamic* complexity, thus modelling and designing systems for domains that call for autonomy, reactivity, adaptability, and distribution.

In this paper, we focus on the fruitful interaction and integration between these two worlds — discussing how, on the one hand, agents can be integrated with DDD to deal with complex dynamic domains, and, on the other hand, DDD is relevant for enhancing the applicability of agent-based approaches to mainstream software development.

In the next section, we start by briefly recalling the main concepts of DDD. We then analyse the bi-directional benefits that both the DDD and the AOSE communities can gain from one another and tie them to related works that

motivate this exploration (Section 3). Following this analysis, we discuss the main scenarios that we see for integrating agent-oriented modelling in the context of DDD, including some examples based on existing (agent-based) technologies (Section 4). We conclude the paper with an overview of future work that we see in this direction.

2 Domain-Driven Design: An Overview

Since the early 2000s, DDD has emerged as a mainstream practice in software development: it is widely adopted for engineering complex systems based on domain models, which are meant to facilitate a shared understanding between technical and business teams. We present the key concepts underlying DDD in Section 2.1. One such concept is that of a Bounded Context, which allows breaking down complex systems into simpler subsystems. We discuss integration across Bounded Contexts in Section 2.2.

2.1 Key Concepts

As summarized by Evans in [13], DDD is an approach to the development of complex software in which designers and developers:

- Focus on the *core domain*.
- Explore *models* in a creative collaboration of domain practitioners.
- Speak a *ubiquitous language* within explicitly *Bounded Contexts*.

A **domain** is a sphere of knowledge, influence, or activity, that is the subject area to which the user applies a program is the domain of the software. A **model** is a system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain. The concept of model is the heart of DDD. Actually, this is a main similarity with other well-known approaches in software engineering, such as Model-Driven Software Engineering (MDSE) [30]. In MDSE models are used to drive the software development process, representing different aspects of the software systems, with an emphasis on automation, i.e., the automatic generation of code and other artifacts from these models. In DDD, the focus instead is more on using models for aligning the software design with the business domain, through a shared understanding and robust modelling of the domain logic. As Evans pointed out in [12], *the Model-Driven Design approach adopted in DDD aims at discarding the dichotomy of analysis model and design to search out a single model that serves both purposes.*

A domain is typically broken down into several **Bounded Contexts**. They represent the description of a boundary (typically a subsystem) within which a particular model is defined and applicable. The **Ubiquitous Language** (UL) is the linguistic counterpart of the model, that is, the language structured around the domain model and used by all team members working within a Bounded Context to discuss the model and connect all the activities of the team in a pervasive way, even into the code itself. Like contexts in general, Bounded Contexts

are also the setting that determines the meaning of a word or statement of the UL, that is: statements about a model can only be understood in relation to a specific context and should not be considered globally defined.

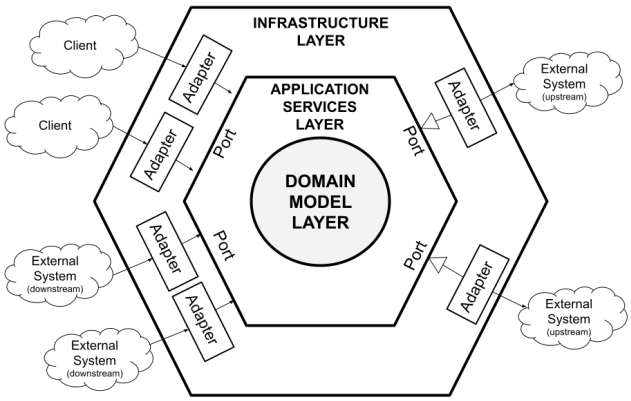

Fig. 1. Hexagonal (Ports-&-Adapters) Architectural Pattern. It uses *adapters* to decouple from the external world, and *ports* to decouple the infrastructure layer from the application layer. All dependencies go inwards, which makes the domain core stable.

Historically, DDD has adopted an object-oriented meta-model for representing any domain model, based on a set of core modelling building blocks composing a domain pattern model, including Entities, Value Objects, Aggregates, Domain Events as well as Services, Modules, Repositories, Factories [34,12].

At the architectural level, DDD calls for a strong separation of the technical concerns from the business concerns by adopting layering. Outer layers should depend on inner layers and the *domain layer* is the heart of the application, isolated from technical complexities (i.e., the *infrastructure layer*) by the application layer, which is in the middle (as depicted in Fig. 1).

The **domain layer** (or model layer) is responsible for representing concepts of the business information about the business situation together with business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure layer. As Evans pointed out, this is the heart of business software.

The **application layer** wraps the domain layer: it defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks for which this layer is responsible are meaningful to the business or necessary for interaction with the application layer of other systems. This layer does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the layer below. Its state only reflects the progress of a task for the final users who, although

not domain experts, should be able to understand and follow the progress of the activities they want to perform.

The **infrastructure layer** provides the generic technical capabilities that support higher layers—e.g., message sending for the application, persistence of the domain, or drawing widgets for the UI.

2.2 Bounded Context Integration

In a project of significance, there are always multiple Bounded Contexts, and two or multiple of those Bounded Contexts will need to integrate [33]. The problem is then of integration in a distributed setting since, by definition, Bounded Contexts are autonomous[3] and loosely-coupled [22] both from a modelling and technical point of view. Models in different Bounded Contexts can evolve and can be implemented independently; however, any two Bounded Contexts may also have inter-dependencies — as they are components that have to interact with one another to achieve the system's overarching design objectives. The Bounded Contexts that have to be integrated may then use different ubiquitous languages, raising the problem of identifying the language to be used for their integration purposes [20]. This problem has to be evaluated and addressed at the design level before the technical (implementation) one.

In DDD, Context Maps are used to explicitly represent, at the design level, relationships among Bounded Contexts. Such relationships are represented in terms of high-level patterns driven by the nature of collaboration between teams working on Bounded Contexts. A main example is given by Customer-Supplier patterns, where a Bounded Context is the supplier of a service for other customer Bounded Contexts. Different specific patterns are used depending on the balance of power adopted (e.g., Conformist, Anticorruption Layer, Open Host).

3 From DDD to AOSE and Back

We believe that a synergy between DDD and AOSE can bring benefits to both communities. In this section, we elaborate on this bi-directional connection and highlight how different initiatives in both communities suggest there might be a growing need for a joint effort aimed at integrating and conceptually aligning the two worlds. We consider this paper as a first step towards this direction.

3.1 DDD Relevance for Agent-Oriented Software Engineering

A main success factor of DDD in the mainstream is its effectiveness as a method for building consensus with stakeholders and developing complex systems that can easily scale and evolve. This is, of course, important in general for software engineering and could have a positive influence on MAS engineering as well.

[3] The term "autonomous" here has the same meaning used in Service-Oriented Architecture, see e.g. [11].

From the point of view of MAS developers, DDD can serve as a valuable way to structure domain models within the MAS itself. Following the different dimensions of Multi-Agent-Oriented Programming (MAOP) [5], domain knowledge can be represented in elements that belong to either the agent, environment, interaction, or organization dimension. As these dimensions can be considered horizontally layered upon each other, complex domains can become hard to represent. DDD and its focus on identifying Bounded Contexts to break down complexity into manageable isolated portions could add a vertical separation between concepts across the different dimensions. Moreover, within the same context, the UL can help maintain a strong consistency in how knowledge is represented across layers, whether it is data managed by agents, encoding norms and policies, or representing external stimuli from the environment.

Finally, as DDD is a generic methodology that does not directly tackle a specific paradigm, it could serve as a common ground in developing integration of MAS with other mainstream software architectures and with mainstream software development in general. For instance, recent efforts in the MAS engineering community have been bridging towards microservices [9] — the software systems more commonly paired with DDD — but also exploring the adoption of other software engineering practices closely related to DDD in agent-based software development, such as Test-Driven Development (TDD) [32,1] and Behaviour-Driven Development (BDD) [8,27].

3.2 DDD Limitations that call for Agents

Landre in [21] remarks that DDD is great for tackling *structural complexity* but not *dynamic complexity*, which appears as a main issue of dynamic systems. Recalling the concept of dynamic complexity and dynamic systems by Derek Hitchins [17], Landre highlights that "complexity is a function of variety, connectedness, and disorder", where we have two types of connections: stable connections, which lead to structural complexity, and arbitrary connections, which lead to dynamic complexity.

Structural domain complexity manifests in nested structures like component hierarchies in products (for example, home appliances, industrial machinery), retail assortments, or project plans. Complexity arises from intricate internal state models, rules, and the extent of connectedness and variability within these systems. In contrast, *dynamic domain complexity* stems from interactions among autonomous components or objects. While objects may possess high internal complexity, dynamic complexity arises from their constantly changing interactions and their arbitrary connectedness. Domain-driven design helps to mitigate structural domain complexity: it provides abstractions such as entities, value objects, aggregates, repositories, and services that bring order, reduce connectedness, and manage variability within and across Bounded Contexts.

However, as Landre remarked in [21], the complexity of the dynamic domain is not addressed really in DDD. One step in that direction has been the introduction of *domain events* [33]. Still, an open issue that remains is the introduction

of proper abstractions specifying how such events are managed to accomplish tasks.

These remarks are even more relevant as soon as we aim at applying DDD for the design of autonomous systems, operating in contexts characterised by uncertainty, and concurrency — such as systems dealing with the physical world, as is the case in cyber-physical systems and the Internet-of-Things (IoT).

4 Empowering DDD with Agent-Oriented Modelling

Given the remarks expressed in Section 3 on the limitations of DDD, a clear benefit that AOSE can bring to DDD is the modelling power of agent-oriented abstractions [31], which we consider at two different levels:

- at the level of a single Bounded Context, that is using agent-oriented abstractions (and metamodels) to define the domain model in Bounded Contexts, beyond the OO domain model pattern typically adopted in DDD;
- at the level of the Context Map, that is defining interactions and relationships among Bounded Contexts at a more systemic level.

In this section, we analyse these two levels, which are orthogonal but could eventually be integrated into a single conceptual view for a methodology that integrates DDD and AOSE. The material associated with this paper [25] includes the source code of the examples discussed in what follows.

4.1 Single Bounded Context

Depending on the complexity of an individual Bounded Context, we can adopt agent-oriented metamodels of increasing complexity as well. In particular, we may consider incrementally the different dimensions typically used for engineering MAS [10,4], which may help to mitigate the complexity inherent to the Bounded Context.

Bounded Context as a Single Agent The simplest case is to map a Bounded Context to a single agent, that is defining the domain model in terms of the concepts typically used to define autonomous, reactive, and proactive behaviour. A minimal set could include concepts to define the agent's objectives (e.g., goals, tasks), how the agent accomplishes such goals (e.g., plans, routines), and concepts defining how the agent interacts with its environment (e.g., perceptions, actions). Beyond this minimal set, we may consider richer (meta)models such as the BDI model, which allows refining the characterisation of objectives into desires and goals, as well as introducing beliefs, intentions, and plans. These concepts become the building blocks of an *agent-oriented domain model pattern*, extending (or replacing) the classic OO version typically used in DDD (and based on concepts such as entities, aggregates, etc.).

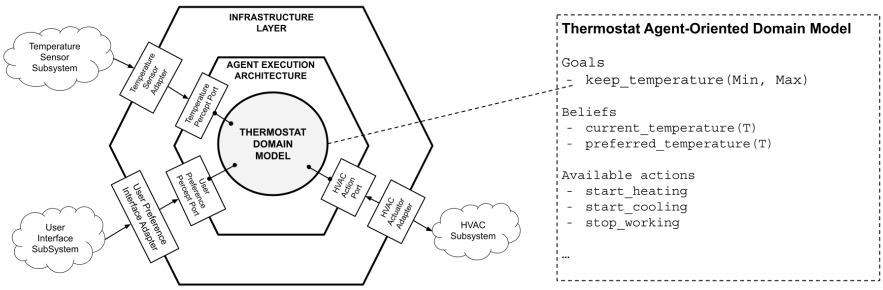

Fig. 2. Modelling a Bounded Context as a single agent: the Thermostat example. The domain model describes the thermostat behaviour in terms of (BDI) agent-oriented concepts. At the architectural level, the hexagonal/clean architecture is preserved: the infrastructure layer includes the adapters for integrating concrete sensing and actuating technologies, as well as technologies for interacting with other subsystems. The percept/action ports and corresponding adapters allow for decoupling the domain model from these specific technologies, yet enabling their effective integration.

As a simple but illustrative example, let's consider the case of a thermostat system modelled as a single agent, designed in a domain-driven perspective (see Figure 2). The domain model modelled using BDI agent-oriented abstractions allows for describing the behaviour of the thermostat in terms of a goal (`keep_temperature(Min,Max)`), beliefs about current temperature as perceived through the temperature sensor subsystem (`current_temperature(T)`) and the preferred temperature as perceived through the user interface subsystem (`preferred_temperature(T)`), and actions to control the HVAC system (`start_heating`, `start_cooling`, `stop_working`).

In DDD, the domain model is "alive": it is meant to have a corresponding computational version that features the specified (or expected) behaviour. The same applies to the agent-oriented version—and, for that purpose, we can then use any agent-oriented programming technology (language, framework, platform) to implement the model. For example, an implementation of the thermostat domain model in Jason [6] is provided in the material associated with this paper [25].

At the architectural level, even when adopting agent-oriented abstractions, we still follow the clean architectural principles, as shown in 2: the agent-oriented domain model remains in the centre and does not depend on any aspect concerning, for example, the technology/infrastructural layer. Accordingly, both the dynamic knowledge about the world of the agent (i.e., its beliefs) as well as the knowledge about its tasks, including the practical knowledge about how to accomplish them or the policies to be used, can be considered part of the domain model at the centre. Around this layer, following the hexagonal architecture, we have the application layer and then the infrastructure layer. When an agent-oriented approach is adopted, the application layer corresponds to the agent execution architecture, which in this case is domain independent. The infras-

tructure layer concerns the implementation of sensors and actuators that enable interaction with the external environment and directly feed the perceptions of an agent. The material [25] includes a complete implementation of the bounded context as a microservice featuring the architecture shown in Figure 2.

Single Agent plus Environment By adopting a single-agent approach, every entity of the Bounded Context must be modelled as part of the state of the agent (e.g., beliefs or plans in the case of BDI agents). A first extension to this approach, enriching the modelling power, is to consider also the environment as first-class modelling abstraction [35]: This allows for realising a separation of concerns between the entities encapsulating the locus of decision-making (agents), and resources and tools used to pursue its objectives (environment). Through this extension, a Bounded Context is mapped into an agent and an environment. In a minimal (meta)model, the environment can be conceived as a single monolithic entity with an observable state (to be perceived by the agent) and a set of actions (to be executed by the agent).

Beyond this minimal metamodel, different approaches have been proposed in the literature for modelling the environment as a first-class abstraction. An approach that was specifically conceived for cognitive/BDI agents is the Agents and Artifacts (A&A) metamodel [23]. A&A allows for modularising the environment in terms of *artifacts* (e.g., resources, tools) each exposing a usage interface composed of observable properties, operations, and observable events (or signals) [23]. On the agent side, artifact operations correspond to the actions that agents can do, and observable properties and events to their percepts, which in BDI agents can be mapped onto beliefs.

As a simple example, Figure 3 shows the thermostat system based on a domain model including a thermostat agent — still described in terms of goals, beliefs, plans as before — and three artifacts, modelling the temperature sensor, the user preference subsystem, and the HVAC subsystem. In this case, the agent sensors/actuators are no longer bound to the external environment, but to the artifact-based environment, which is used to enable and mediate the interaction with the external environment. A full implementation based on the JaCaMo platform [3] is provided in the material [25].

At this stage, the logic governing the interaction with sensors and actuators represented at the infrastructure layer becomes part of the domain as well in the form of *artifacts*. They serve as DDD aggregate roots as the single entry point for a portion of the environment (and inherently of the domain of the bounded context). This option achieves a modularisation of such logic, improving the quality of the model. This also paves the way for more complex systems that may include many different entities in an environment, as well as possibly many different agents implementing the domain logic.

Multi-Agent Systems and Agent Organisations Finally, in a (complex) Bounded Context, the domain may be better modelled by considering multiple communicating agents sharing the same environment. This can be useful,

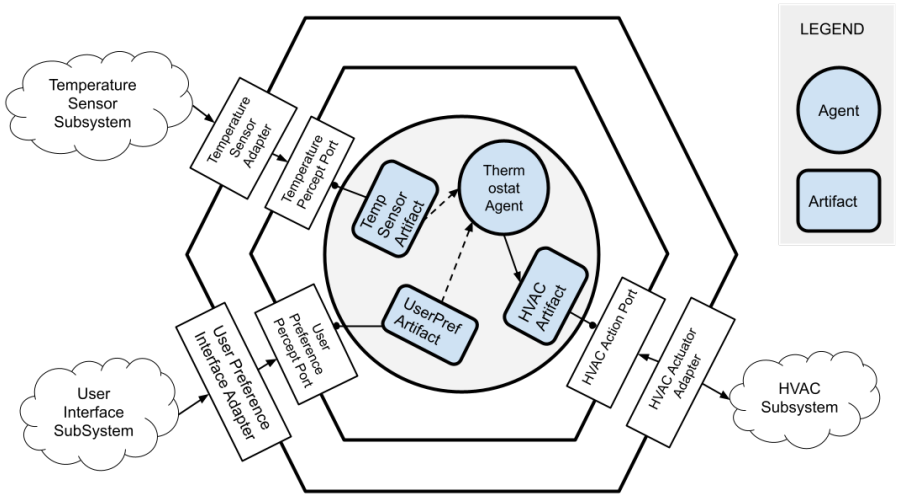

Fig. 3. Modelling a Bounded Context as a single agent plus environment: in this case, the domain model includes also environment abstractions. In the example, the A&A is used and artifacts as environment abstractions are used to explicitly model the temperature sensor, the user preference sub-system and the HVAC sub-system.

for instance, to model explicitly at the domain level different loci of control or decision-making, eventually interacting in some way—either through direct communication (e.g., using agent communication languages) or indirectly through the environment. Organisation modelling approaches in this case can be further introduced to have first-class abstractions to specify the structure of the system as well as coordination activities (e.g., see [2]). Going further, even if in principle MAS and Organisation modelling can be useful for individual Bounded Contexts, their value is particularly important in scenarios involving multiple Bounded Contexts, as discussed in the next section.

4.2 Bounded Context Integration

Typically a project involves multiple "autonomous" Bounded Contexts, that need to be integrated to achieve the system's overarching goals. To this purpose, MAS and Organisation modelling can significantly enhance the approaches used in DDD to deal with this problem (see Section 2.2) while at the same time preserving the key principles and philosophy of DDD.

On the one hand, the integration problem among Bounded Contexts is clearly related to the interoperability problem, for example, in Service-Oriented Architecture (SOA) [11] or in (open) multi-agent systems. A general approach to deal with problems at the technical level accounts for defining an explicit *semantic* layer (i.e., semantic interoperability), in terms of shared ontologies formalising

the meaning of concepts, including those specifying aspects concerning the interaction among parties (e.g., contracts). On the other hand, MAS and organisation modelling can be applied to extend the domain-driven principles beyond the individual Bounded Context, to define models capturing also aspects concerning the interaction, coordination, and cooperation among bounded contexts at the proper level of abstraction. The patterns currently adopted in the mainstream and described in Section 2.2 are only marginally useful for tackling this problem.

In what follows, we explore two main macro cases of increased complexity: modelling multiple Bounded Contexts as a MAS, and modelling multiple Bounded Contexts as an Agent Organisation.

Multiple Bounded Contexts as Interacting Agents This case accounts for modelling each Bounded Context as either an agent (as described in the previous section) or as part of the environment where agents are situated. By designing Bounded Contexts as agents, we can exploit the expressive power of agent interaction models to describe the high-level semantics of interaction as defined at the domain level, in spite of the specific enabling network/communication/implementation technology and protocols. Main examples include high-level Agent Communication Languages such as FIPA ACL or KQML, or more recent information-oriented protocol specification languages such as BSPL [29].

A simple example extending the previous ones concerns a smart room scenario including a number of autonomous subsystems, in an Internet of Things perspective, each one properly designed as a distinct Bounded Context — and modelled as an autonomous agent. In that case, the request to the Thermostat to change/keep the temperature could be modelled, for example, as a request to achieve a goal `keep_temperature(Min,Max)`, possibly sent by the Bounded Context representing the smart room supervisor.

The MAS modelling includes also the case where one or multiple Bounded Contexts modelled as agents interact with one or multiple Bounded Contexts modelled as the (agent) environment, as described in Section 4.1. In the smart room scenario, for instance, the specific sensing and actuating subsystems used by the Thermostat Bounded Context may be designed as separate Bounded Contexts, modelled, for example, as artifacts (as in the A&A metamodel) used by the agent.

It is worth remarking that, in this case, the concepts of agent and artifact as well are used first of all as modelling abstraction for modelling the domain, not as technologies for implementing the system. So we may have Bounded Contexts modelled as agents (and artifacts) implemented using technologies and frameworks used in the mainstream to implement, for example, (micro)services and enterprise applications.

Multiple Bounded Contexts as an Agent Organisation Finally, organisation modelling as developed in the MAS literature may provide a proper abstraction layer to tackle the structural and behavioural complexity of systems composed of a dynamic (and possibly large, open) set of interrelated Bounded

Contexts. The concepts of groups and roles as provided, for example, in AGR [14] and Moise [18] can be effective to model domains where the structure is dynamic. In the smart room case, for instance, the configuration and facilities of the room may adapt and change dynamically depending on the users that enter, possibly having different preferences but also different capabilities, related to their roles.

5 Concluding Remarks

The bi-directional conceptual integration of DDD and AOSE could be beneficial for both worlds: on the one hand, it can extend DDD with the proper level of abstraction for building dynamic, adaptive, and complex systems that exhibit relevant levels of autonomy; on the other hand, it can help structure the development of domain models in AOSE. This paper provides a first conceptual framework to conceive and understand such integration, discussing in particular how agent-oriented modelling could be exploited both at the individual Bounded Context level and for the integration of Bounded Contexts.

This contribution is clearly related to existing work in literature exploring the adoption of agile methodologies for agent-based systems [26,7], including Test-Driven Development (TDD) and Behaviour-Driven Development (BDD) [28]. Domain-Driven Design provides us the background approach and reference, in terms of general modelling and design principles, in which also TDD and BDD can be properly conceptually situated.

The conceptual framework depicted in this paper is meant to provide a baseline for future work. One direction to be explored further is the development of a concrete agent-oriented DDD methodology, one that is based on an agent-oriented domain pattern. In line with the DDD philosophy, defining such a pattern would have to start from domain practice and a set of relevant use cases. For example, in the building automation domain, domain experts are the ones who define libraries of automation programs (i.e., procedural knowledge) that are then assembled to meet the technical requirements of the automation system for a specific building. Mapping such domain concepts and their relations to agent-oriented abstractions could bring insight into what would be a suitable agent-oriented domain pattern in this area.

A second research direction, which we explored less in this paper, is the development of a DDD-inspired agent-oriented methodology. The influence of DDD on AOSE could bring additional structure to the way MAS are designed and built. For example, adopting existing DDD modelling patterns (*tactical patterns* as known in DDD) and integrating them within agent-oriented methodologies could be one of the core challenges for a fruitful integration — and may allow developers to have more structurally sound domain models for MAS. A line of work on engineering MAS relevant to this research direction is the use of domain ontologies for model-driven engineering (e.g., see [15]). If domain models are expressed formally as domain ontologies, then ontology engineering methodologies (e.g., SAMOD [24], ACIMOV [16]) could further inform this investigation.

A third direction for future work is to investigate the design and development of new tools that would provide proper conceptual support for exploiting the integration of DDD and agent-orientation. Such tools may be based on existing ones for DDD and AOSE, but would likely require rethinking the development process from the ground up to provide for a streamlined integration of the two.

References

1. Amaral, C.J., Hübner, J.F., Kampik, T.: TDD for AOP: test-driven development for agent-oriented programming. In: Agmon, N., An, B., Ricci, A., Yeoh, W. (eds.) Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023. pp. 3038–3040. ACM (2023). https://doi.org/10.5555/3545946.3599165
2. Boissier, O., Bordini, R., Hubner, J., Ricci, A.: Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo. Intelligent Robotics and Autonomous Agents series, MIT Press (2020), https://books.google.it/books?id=GM_tDwAAQBAJ
3. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming **78**(6), 747–761 (2013)
4. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A.: Dimensions in programming multi-agent systems. Knowledge Eng. Review **34**, e2 (2019). https://doi.org/10.1017/S026988891800005X
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Sci. Comput. Program. **78**(6), 747–761 (2013). https://doi.org/10.1016/J.SCICO.2011.10.004
6. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons (2007)
7. Carrera, Á., Iglesias, C.A., Garijo, M.: Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. Information Systems Frontiers **16**, 169–182 (2014)
8. Carrera, Á., Iglesias, C.A., Garijo, M.: Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. Inf. Syst. Frontiers **16**(2), 169–182 (2014). https://doi.org/10.1007/S10796-013-9438-5
9. Collier, R.W., O'Neill, E., Lillis, D., O'Hare, G.M.P.: MAMS: multi-agent microservices. In: Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019. pp. 655–662 (2019). https://doi.org/10.1145/3308560.3316509
10. Demazeau, Y.: From interactions to collective behaviour in agent-based systems. In: European Conference on Cognitive Science. vol. 95 (1995)
11. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ (2005)
12. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
13. Evans, E.: Domain-Driven Design Reference: Definitions and Pattern Summaries. Dog Ear Publishing (2014)
14. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J. (eds.) Agent-Oriented Software Engineering IV. pp. 214–230. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

15. Freitas, A., Bordini, R.H., Vieira, R.: Model-driven engineering of multi-agent systems based on ontologies. Appl. Ontol. **12**(2), 157–188 (jan 2017). https://doi.org/10.3233/AO-170182
16. Hannou, F.Z., Charpenay, V., Lefrançois, M., Roussey, C., Zimmermann, A., Gandon, F.: The acimov methodology: Agile and continuous integration for modular ontologies and vocabularies. In: Proc. of the 2nd Workshop on Modular Knowledge (MK 2023). CEUR Workshop Proceedings, vol. 3637 (2023), https://ceur-ws.org/Vol-3637/paper25.pdf
17. Hitchins, D.K.: Advanced Systems Thinking, Engineering and Management. Artech House (2003)
18. Hübner, J.F., Sichman, J.S.a., Boissier, O.: Moise+: towards a structural, functional, and deontic model for mas organization. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1. p. 501–502. AAMAS '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/544741.544858
19. Jennings, N.R.: On agent-based software engineering. Artif. Intell. **117**(2), 277–296 (mar 2000). https://doi.org/10.1016/S0004-3702(99)00107-1
20. Khononov, V.: Learning Domain-Driven Design. O'Really (2022)
21. Landre, E.: Domain-Driven Design: The First 15 Years Essays from the DDD Community, chap. Agents aka Domain objects on steroids. Lean Pub (2024)
22. Millett, S., Tune, N.: Patterns, Principles and Practices of Domain-Driven Design. Wrox (2015)
23. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems **17**(3), 432–456 (dec 2008). https://doi.org/10.1007/s10458-008-9053-x
24. Peroni, S.: SAMOD: an agile methodology for the development of ontologies (11 2016). https://doi.org/10.6084/m9.figshare.3189769.v4, https://figshare.com/articles/journal_contribution/SAMOD_an_agile_methodology_for_the_development_of_ontologies/3189769
25. Ricci, A., Burattini, S., Ciortea, A., Castellucci, M.: Agents for DDD – Back and Forth – Material (06 2024), https://github.com/Agents-and-DDD/EMAS-2024-paper-material.git
26. Rodriguez, S., Thangarajah, J., Winikoff, M.: User and system stories: An agile approach for managing requirements in aose. In: Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems. p. 1064–1072. AAMAS '21, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2021)
27. Rodriguez, S., Thangarajah, J., Winikoff, M.: A behaviour-driven approach for testing requirements via user and system stories in agent systems. In: Agmon, N., An, B., Ricci, A., Yeoh, W. (eds.) Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023. pp. 1182–1190. ACM (2023). https://doi.org/10.5555/3545946.3598761
28. Rodriguez, S., Thangarajah, J., Winikoff, M.: A behaviour-driven approach for testing requirements via user and system stories in agent systems. In: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems. p. 1182–1190. AAMAS '23, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2023)
29. Singh, M.P.: Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language. In: The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2. p. 491–498. AAMAS '11,

International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2011)
30. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Inc., Hoboken, NJ, USA (2006)
31. Sterling, L., Taveter, K.: The Art of Agent-Oriented Modeling. The MIT Press (2009)
32. Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C.: SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) Agent-Oriented Software Engineering VII, 7th International Workshop, AOSE 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers. Lecture Notes in Computer Science, vol. 4405, pp. 156–173. Springer (2006). https://doi.org/10.1007/978-3-540-70945-9_10
33. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley (2013)
34. Vernon, V.: Domain-Driven Design Distilled. Addison-Wesley, Boston, MA (2016)
35. Weyns, D., Van Dyke Parunak, H., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems state-of-the-art and research challenges. In: Environments for Multi-Agent Systems: First International Workshop, E4MAS 2004, New York, NY, July 19, 2004, Revised Selected Papers 1. pp. 1–47. Springer (2005)