Agents for DDD – Back and Forth

Alessandro Ricci[1], Andrei Ciortea[2],
Samuele Burattini[1], 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, matteo.castellucci@studio.unibo.it
[2] School of Computer Science, University of St.Gallen, Switzerland
andrei.ciortea@unisg.ch

Abstract. In this paper, we are interested in exploring the bi-directional conceptual interaction between Agent-Oriented Software Engineering (AOSE) and Domain-Driven Design (DDD). Our aim is not only to extend DDD with the proper level of abstraction that would make it effective in designing complex software systems — that is, systems featuring relevant levels of autonomy, interactions, adaptivity, dynamism, etc. — but also to integrate DDD in AOSE. As a first step, we explore two integration perspectives: (1) modelling bounded contexts as cognitive agents, or (2) as workspaces in which multiple agents interact to perform their activities. Both perspective draw a separation of concerns between the application layer and domain layer in a DDD-AOSE integration. We sketch a roadmap for further investigating this 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" [5] — and since has become a reference approach for a large community of designers and developers in mainstream software development [17]. 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 [8]. 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 [10]. 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. We start in the next section 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. Following this analysis, we propose an initial conceptual integration with two perspectives on how to align AOSE abstractions with DDD. The key characteristic of both perspectives is to identify the application layer in DDD as the primary focus for AOSE abstractions. Other integration perspectives using different AOSE abstractions could follow a similar path. We conclude the paper with a roadmap for future work in this direction.

2 DDD Key Concepts

As summarized by Evans in [6], Domain-Driven Design 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. A domain is typically broken down in 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 with the 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.

Historically DDD has adopted an object-oriented meta-model for describing any 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 [17,5].

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, and business rules. State that reflects the business situation is controlled and used here, even though the

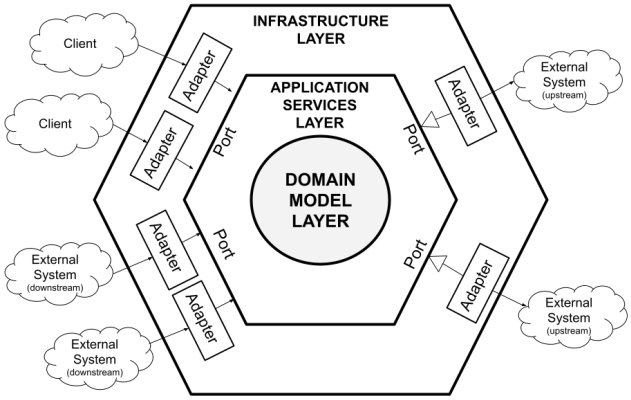

Fig. 1. Hexagonal (Ports-&-Adapters) Architectural Pattern

technical details of storing it are delegated to the infrastructure layer. As pointed out by Evans, this is the heart of business software.

The **application layer** wraps the domain layer and defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for 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 being 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, and drawing widgets for the UI.

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.

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) [2], 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 [4] — the software systems more commonly paired with DDD — but also exploring the adoption of other software engineering practices closely related to DDD such as Test-Driven Development (TDD) [15,1] and Behaviour-Driven Development (BDD) in agent-based software development [3,14].

3.2 DDD Limitations that call for Agents

Landre in [10] 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 [7], 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 (e.g., home appliances, industrial machinery), retail assortments, or project plans. Complexity arises from intricate internal state models, rules, and the extent of the 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 arbitrary connectedness. Domain-driven design helps to mitigate structural 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 remarked by Landre in [10], dynamic complexity is not really addressed in DDD. One step in that direction has been the introduction of *domain events* [16]. 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 the Internet-of-Things (IoT) and cyber-physical systems.

4 Empowering DDD with Agents (and Artifacts)

In this section, we discuss an integration of AOSE abstractions in DDD to make it effective to deal with dynamic complexity in a domain. The AOSE literature provides a large range of abstractions and meta-models that have been introduced to model and design the many different aspects and levels of domains and systems. In the following, we will consider first a single agent (dimension) perspective, then we will look into MAS and introduce also an environment dimension as proposed in A&A [12] and applied to MAOP [2]. Other alignments may of course be possible and necessary to apply the DDD approach to all the different dimensions of MAS engineering. Here we lay a sketch of two intuitive alignments on the agent and the environment dimension to motivate the community towards a deeper exploration of the concept in future works.

4.1 Bounded Contexts as Cognitive Agents

A first straightforward way to integrate agents in DDD so as to overcome the limitations described in [10] is to model and design a whole bounded context as a single coarse-grained agent featuring an autonomous, pro-active and reactive behaviour in order to accomplish the business tasks as defined by the business use cases. At the architectural level, the three layers of the hexagonal architecture can be mapped into three distinct layers of an agent architecture. The very interesting point here is what happens if we consider high-level, cognitive architectures – such as BDI [13], SOAR [9] or alike – that is a vision of agents at the Knowledge Level [11]. In this case, both the dynamic knowledge about the world of the agent, i.e. its beliefs, as well as the knowledge about its tasks and 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 (see Fig. 2).

Around this layer, in this case, we have the agent execution architecture: following the hexagonal architecture this is the place of the application layer. Differently, from DDD, in which the application layer is still customised depending on the business tasks when considering cognitive agents, this layer is meant to be fixed and fully domain-independent. Finally, the infrastructure layer concerns the implementation of sensors and actuators that enable interaction with the external environment.

4.2 Bounded Contexts as Workspaces

A conceptual extension of the previous approach is to model the bounded context as a workspace where one or multiple agents can work together to accomplish the

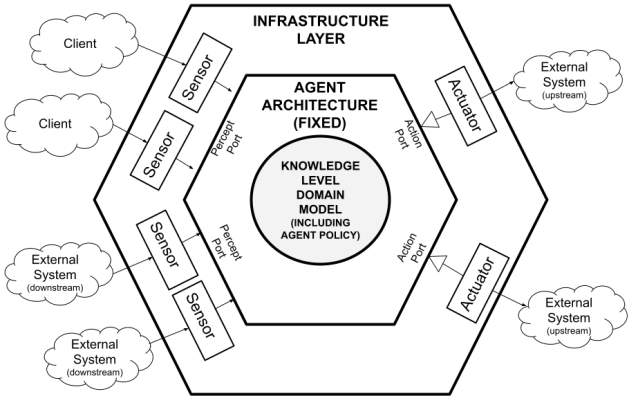

Fig. 2. Modelling a Bounded Context as a single agent: the infrastructure layer provides the means to receive perceptions and make actuation, and the domain model is the agent's internal knowledge base including beliefs and policies.

business tasks defined for that context. In this case, agents are used as building blocks at the application layer for designing that application logic which must exhibit forms of autonomous behaviour to fulfil business use cases.

By putting agents at the application layer, both the domain layer and the infrastructure layer can be naturally conceived as the environment where the agent(s) are logically situated, that they need to work with and manipulate, to get jobs done. Accordingly, environment first-class abstractions can be adopted [18], defining then the set of actions that can be executed in that dynamic context, and what agents can perceive/observe of that context in terms of observable state and events. Therefore, in a bounded context, we may have one or multiple application agents sharing the same environment, in charge of possibly different tasks, and possibly cooperating either by means of direct communication using some communication language, or mediated interaction through the environment.

By adopting the A&A metamodel [12], the environment first-class abstractions that can be used to modularise the environment are called *artifacts*, and the application layer can be designed as an A&A *workspace* (see Fig. 3). Each artifact exposes a usage interface composed of observable properties, operations, and observable events (signals) [12]. On the agent side, artifacts' operations correspond to the actions that agents can do, and observable properties and events to their percepts – to be mapped onto, e.g., beliefs in BDI agents.

Some artifacts can be used to enable and mediate the access to domain objects, part of the domain layer—let's call them *domain artifacts*. Useful constraints when mapping domain objects into domain artifacts can be derived from DDD building blocks' properties. For instance, an obvious but fundamental one is that artifact observable properties and events should be defined in the domain model (and the ubiquitous language), and their value should be represented by

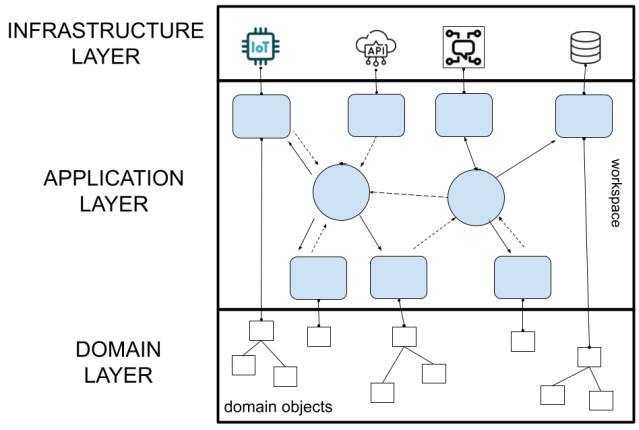

Fig. 3. Modelling a Bounded Context as an A&A workspace: artifacts (rectangles) represent domain objects or external interfaces, whereas agents (circles) encapsulate the application logic to connect external perceptions to changes in the domain

value objects. Another one is that entities or aggregates – i.e. domain objects with state and identity – should be referenced and managed by a single artifact; nevertheless, the same repositories and factories can be shared and referenced by multiple artifacts. Some other artifacts may represent the means that allow agents to communicate with the external world—encapsulating the access to e.g. API infrastructural components. Some artifacts may represent entities that are both bound to domain objects and exploit infrastructural means to be acquired or to have an effect on the external world.

As an illustrative example, Fig. 4 shows a smart thermostat application designed using this approach: a single agent implements the application logic to interact with several artifacts proactively and achieve the goal of keeping the temperature within the preferences expressed by the user through some form of API. Different artifacts represent different aggregate roots and keep the separation that is established within the domain model. Being at the application layer, they implement the logic of managing the interaction between the infrastructure components and the domain objects. The agent encapsulates the proactive behaviour that is required in this scenario, interacting with different artifacts.

5 The Road Ahead

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. As a first step, we explore two integration

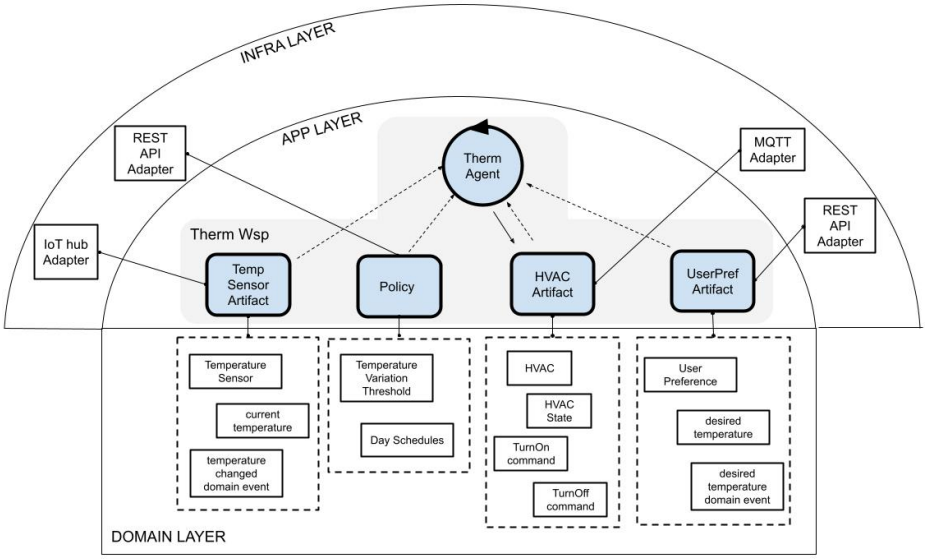

Fig. 4. A Bounded Context for the management of a Smart Thermostat mapped as a workspace showing how artifacts serve as bridges towards the domain model.

perspectives: modelling each bounded context either as a single cognitive agent or as a workspace in which multiple agents interact and perform their activities. Both perspectives draw a separation of concerns between the application and domain layers in a DDD-AOSE integration. Still, many open issues remain, such as *knowledge-oriented domain modelling*, *agent coordination*, or *agent mobility* (i.e., moving across different bounded contexts when represented as workspaces).

The abstractions used in DDD for modelling the domain layer betray its focus on object orientation. For instance, while it is possible to encapsulate procedural knowledge or the state of an interaction protocol into domain objects, AOSE provides abstractions specifically designed for engineering such concepts into software systems. Agents are programmed at the knowledge level and, as we move towards agent-based systems, the domain layer could benefit from such AOSE abstractions. Then, in both perspectives we put forward, agents are ascribed within bounded contexts. A challenge that remains is coordination across bounded contexts. While DDD provides abstractions for communication across bounded contexts (e.g., via domain events), they are inadequate for capturing the complexity of multi-agent coordination. Finally, if we model bounded contexts as workspaces, restricting agents to one workspace might simplify the design of the system but it also limits the agents' autonomy and adaptivity. Allowing agents to operate and migrate across multiple bounded contexts would require further conceptual support and clarity for the overall organization of the MAS.

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.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
3. 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, https://doi.org/10.1007/s10796-013-9438-5
4. 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
5. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
6. Evans, E.: Domain-Driven Design Reference: Definitions and Pattern Summaries. Dog Ear Publishing (2014)
7. Hitchins, D.K.: Advanced Systems Thinking, Engineering and Management. Artech House (2003)
8. 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
9. Laird, J.E.: The Soar Cognitive Architecture. The MIT Press (2012)
10. Landre, E.: Domain-Driven Design: The First 15 Years Essays from the DDD Community, chap. Agents aka Domain objects on steroids. Lean Pub (2024)
11. Newell, A.: The knowledge level. Artif. Intell. **18**(1), 87–127 (jan 1982). https://doi.org/10.1016/0004-3702(82)90012-1
12. 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
13. Rao, A.S., George, M.P.: BDI agents: From theory to practice. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95). pp. 312–319 (1995), http://www.agent.ai/doc/upload/200302/rao95.pdf
14. 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
15. 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, https://doi.org/10.1007/978-3-540-70945-9_10

16. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley, Upper Saddle River, NJ (2013), https://www.safaribooksonline.com/library/view/implementing-domain-driven-design/9780133039900/
17. Vernon, V.: Domain-Driven Design Distilled. Addison-Wesley, Boston, MA (2016)
18. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous Agents and Multi-Agent Systems **14**(1), 5–30 (feb 2007). https://doi.org/10.1007/s10458-006-0012-0