Extreme Programming of Multi-Agent Systems

Holger Knublauch
Research Institute for Applied Knowledge Processing (FAW)
Helmholtzstr. 16, 89081 Ulm, Germany
Holger.Knublauch@faw.uni-ulm.de

ABSTRACT

The complexity of communication scenarios between agents make multi-agent systems difficult to build. Most of the existing Agent-Oriented Software Engineering methodologies face this complexity by guiding the developers through a rather waterfall-based process with a series of intermediate modeling artifacts. While these methodologies lead to executable prototypes relatively late and are expensive when requirements change, we explore a rather evolutionary approach with explicit support for change and rapid feedback. In particular, we apply Extreme Programming, a modern agile methodology from object-oriented software technology, for the design and implementation of multi-agent systems. The only modeling artifacts that are being maintained in our approach are a process model with which domain experts and developers design and communicate agent application scenarios, and the executable agent source code including automated test cases. We have successfully applied our approach for the development of a prototypical multi-agent system for clinical information logistics.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.2 [**Design Tools and Techniques**]: Evolutionary prototyping; D.2.5 [**Testing and Debugging**]

General Terms

Design, Experimentation, Human factors

Keywords

Agent-Oriented Software Engineering

1. INTRODUCTION

An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [7]. Agents differ from conventional software particularly in their complex interaction and communication scenarios. While the properties of agents make multi-agent systems a very natural and efficient approach for solving many types of problems, the complex communication scenarios and the emerging system behaviors often lead to situations that are difficult to predict and plan [11]. Any development methodology for multi-agent systems must take these difficulties into account.

The currently discussed Agent Oriented Software Engineering (AOSE) [3] approaches face the complexity of agent development by defining modeling languages, processes and tools to systematically divide complexity into a collection of inter-related models. AOSE approaches follow the traditional Software Engineering paradigm, in which a relatively large chunk of project resources in spent on up-front analysis and design while implementation and evaluation are moved to phases when the requirements are thought to be sufficiently understood.

Recent years, however, have provided evidence that alternatives to systematic engineering approaches exist in software development, in particular when requirements are unclear or prone to change. These so-called *agile* methodologies focus on producing executable code early and on exposing this code to evolution in the face of customer feedback. Instead of putting resources into well-defined modeling phases and artifacts, agile methodologies focus on keeping code easy to change. While agile approaches like *Extreme Programming (XP)* [2] are rapidly gaining industrial acceptance for conventional types of software [13], it still remains to feed the ideas of these approaches into the agent community. In this document, we describe an XP approach for the development of multi-agent systems and report on an encouraging case study in which we have applied it.

This document is organized as follows. Section 2 provides a brief review of current Agent-Oriented Software Engineering methodologies. Section 3 gives a short introduction to XP in general. Section 4 gives an overview of our methodology of applying XP to multi-agent systems. Section 5 reports on a case study in which a prototypical multi-agent system has been successfully implemented using our approach. Section 6 discusses strengths and weaknesses of XP for agent development, followed by a conclusion in section 7.

2. AGENT-ORIENTED SOFTWARE ENGINEERING

Research in *Agent-Oriented Software Engineering* [3] aims

AAMAS'02, July 15-19, 2002, Bologna, Italy.

at defining methods, techniques, tools, and modeling languages for the construction and maintenance of multi-agent systems. AOSE tries to overcome the limitations of general-purpose development methods which fall short of adequate, intuitive and natural modeling techniques for agents, as well as the implementation of complex communication.

In most of the existing AOSE methodologies, the developers are supplied with new notations, like extensions of UML for agents [1], so that the transition from a high-level requirements document to executable code can be supported by an adequate series of intermediate modeling artifacts. The *Gaia* methodology [15], which is a representative state-of-the-art AOSE methodology, guides developers through various analysis and design activities, which result in a design that can serve as a starting point for traditional object-oriented design and implementation techniques. As illustrated in figure 1, a basic model of Gaia is the roles model, in which responsibilities, permissions, activities and life-cycles of the later defined agents are specified. Gaia suggests various formal and semi-formal languages for this specification. Other phases of Gaia result in models which specify interactions, agent types and instances, services, and the communication channels between agents.

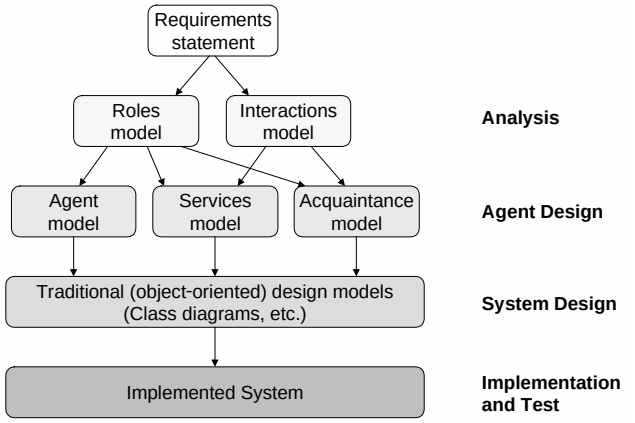

Figure 1: The Gaia methodology for AOSE.

Gaia can be characterized as a rather waterfall-based approach with relatively many steps to follow prior to implementation and test. The same holds for most of the other AOSE methods. *Multiagent Systems Engineering* [4], for example, is divided into seven successive phases with intermediate models between each phase. Advantages of building such models are that development processes become reproducible and (at least apparently) planable, and that design models are not constrained by implementation details. However, the general weakness of such waterfall-based approaches is the overhead when models need to be changed. Gaia's design model is relatively decoupled from the implementation, i.e. the entire design (perhaps even the analysis) has to be revised in order to develop a model that can actually be implemented [11]. Customer feedback is available late, so that systematic AOSE methods are suitable only if requirements are relatively stable. In our opinion, this is often unrealistic, because the complexity of potential agent interaction scenarios and the emerging behaviors within a multi-agent system can make pre-planning very difficult [11].

3. EXTREME PROGRAMMING

In this document, we explore alternatives to the waterfall-based AOSE methods. In particular, we focus on so-called light-weight or *agile* methodologies, from which *Extreme Programming (XP)* [2] is the most widely-known. XP is being increasingly used in projects with uncertain or changing requirements like those typically encountered in the internet age. Instead of a strict software process with well-defined activities and modeling artifacts, XP relies on a rather evolutionary style in which the implementation and evaluation of executable code are given priority over a comprehensive documentation. In support of evolution, efforts are made to keep the cost of change as low as possible.

An XP project is guided by four long-term goals, or *values*, namely feedback, communication, simplicity, and courage. These values are put into daily practice by means of twelve *practices*. The four values and some of the supporting practices are briefly described in the following.

Feedback suggests to expose prototypes frequently to customer feedback and automated test procedures. Feedback is ensured by delivering prototypes frequently and having a real *customer on-site* at all times, so that ill-understood requirements can be detected and eliminated early. Feedback is furthermore achieved by forcing the developers to write test cases together with the production code. Testing tools like *JUnit* [6] allow to execute tests automatically to ensure that the planned features work properly and that changes in one module have not accidentally destroyed other parts. Designing and writing tests is also a good way of clarifying and documenting requirements.

Communication suggests to encourage direct face-to-face collaboration instead of collaboration by means of comprehensive modeling artifacts. The principle of *traveling light* suggests to maintain as few formal models as possible and to rely on rather direct communication channels instead. Fluent communication is also expressed in the practice of the *planning game*, which suggests to let customers and developers jointly write and prioritize requirements on so-called "story cards". Beside the practice of the on-site customer, communication is also fostered by the practice of *pair programming*, in which programmers are working in pairs on a single machine. This helps to detect errors early, improves creativity, and helps to spread knowledge within the team.

Simplicity suggests to focus on solutions that are easy to design and implement, so that new features can be built rapidly when requirements change. The simple solutions are improved or generalized only on demand, by applying the practice of *refactoring* [5], which helps to improve the design of existing code without altering its functionality.

Courage is often needed to escape modeling dead-ends in which teams might find themselves after many incremental changes and refactorings. XP also encourages the single developers to feel responsible for the project and thus improves team motivation.

While none of the single values and practices of XP is particularly new, XP is more than plain hacking under a new label. As illustrated in figure 2, the practices of XP support each other, i.e. the weakness of one practice is compensated by the strength of another. The consequent and disciplined combination of the practices allows teams to spend less resources on an up-front design while maintaining flexibility in the face of changing requirements.

XP is not the best choice for projects in which require-

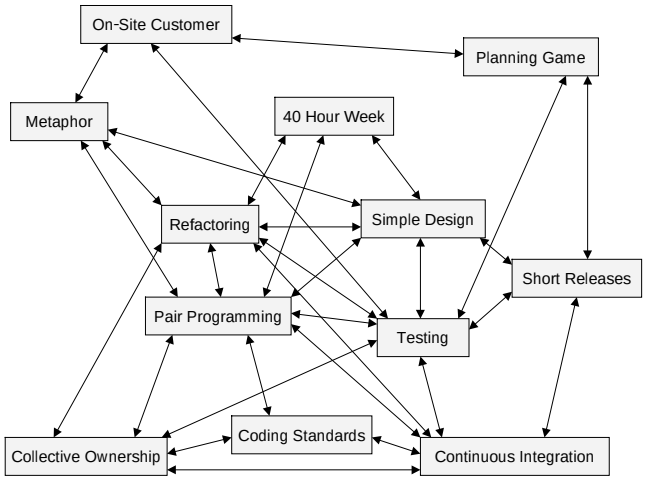

Figure 2: The practices of XP support each other.

ments are relatively stable and easy to formalize. Furthermore, XP is assumed to be limited to small to medium-sized projects of about 10 programmers. Finally, it requires management commitment [2].

4. XP OF MULTI-AGENT SYSTEMS

In this section, we describe an Extreme Programming approach for developing multi-agent systems. Note that this approach is *one* potential process model with XP, while many other variants of the XP practices are possible and should be explored in future work. XP does not prescribe certain tools or models, so we could "invent" our own.

Our approach is guided by the XP values of simplicity, communication and feedback. In particular, we rely on very simple models and metamodels so that agent interactions can be modeled, changed, and communicated quickly. Following the principle of traveling light, we suggest to build and maintain only two models: The source code of the executable agents (including test cases) and a process model.

The process model describes the agents' application scenarios, or – in the XP terminology – the story cards. While methods like Gaia specify agents from a rather local point of view, i.e. the agents' roles are modeled in isolation and integrated later, our approach takes a bird's-eye view on the process. We can thus bundle the information that is distributed across several Gaia models in a single, graphical design model. As illustrated in the lower part of figure 3, this process model is the starting point of the implementation and test activities. The process model allows to generate parts of the source code (details below) and serves as a requirements statement that can be communicated between the team members. Our approach explicitly assumes that the initial process model will be incomplete or wrong, so that it will need to be updated in the face of feedback from the implementation. We therefore yield a cyclic development process that allows one to switch between implementation and model updating arbitrarily.

Optionally, in cases where agents are designed to be introduced into an existing workflow, our XP development cycle is preceded by an analysis step in which the existing process is formalized. The resulting model of the existing process can serve as the base for the "agentified" process

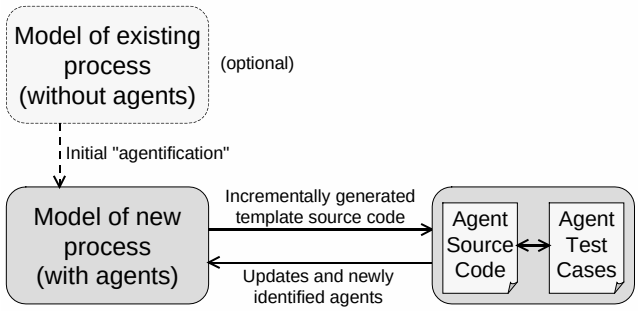

Figure 3: An overview of our Extreme Programming approach for multi-agent system.

model. Both old and new process models are edited in the same metamodel with the same tool, so that agents can be introduced incrementally.

The following subsections will provide details on our process modeling approach (4.1) and a prototypical implementation framework that we found to be useful for XP (4.2).

4.1 Process Modeling

Our approach relies on process modeling to capture and clarify requirements, to visually document agent functionality, and to enable communication with domain experts. The process metamodel was designed to be easy to comprehend and use by end users of the agent application, to be extensible for specific types of agents, and to allow for automatic and semi-automatic transformation into executable code. Figure 4 illustrates some modeling symbols.

Figure 4: A process model with two agent activities and one message.

A schematic overview of our process metamodel is provided in figure 5. A *process* is a collection of sub-processes and *activities*. An activity is an atomic unit of work that is performed by an actor who takes a specified *role*. A role can be taken either by a *human* or an *agent*. Agents represent types of software agents that can be arranged in an inheritance hierarchy like object-oriented classes. We found it useful to adopt the classification by Sycara et al. [14] who distinguish between interface, task, and information agents. Of course, completely different types of agents could be used, too. Activities and processes can be arranged in arbitrary predecessor/successor sequences, although we use this ordering mostly for illustration purposes only.

Activities can read and write *media*. A medium is either analog, like a phone call or a letter, or an *agent message*. An agent message can be characterized by its command (or performative) and its ontology. In our Java-based projects, we found it very comfortable to rely on a Java-based ontology and to pass Java objects as message content, so that it became easy to transfer and process information between

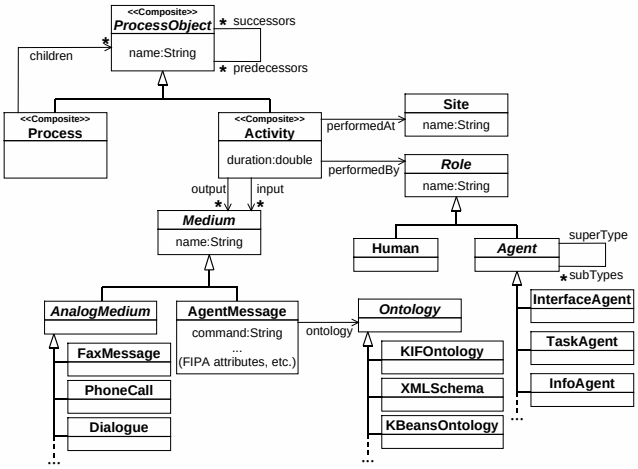

Figure 5: A schematic view of our metamodel for agent-based process models.

agents. In particular, our content objects are *KBeans* [8] instances. KBeans is an extension of JavaBeans that allows to attach semantic constraints to class attributes, so that agents can automatically reject invalid message content or perform reasoning.

Figure 6 shows a visual modeling tool, called *AGIL-Shell*[1] [9], with which instances of the process metamodel can be edited by dragging boxes and arrows. Note that various off-the-shelf tools like Visio could be used for process modeling as well. AGIL-Shell is able to automatically generate several types of views in addition to the normal bird's-eye process view. For example, it allows to visualize the message flow between agents in a graph, with agents as nodes and messages as edges. Another view can be activated in which life-cycle, input and output of a single role (agent) are shown. By the way, these views are very similar to the acquaintance and life-cycle models from the Gaia methodology.

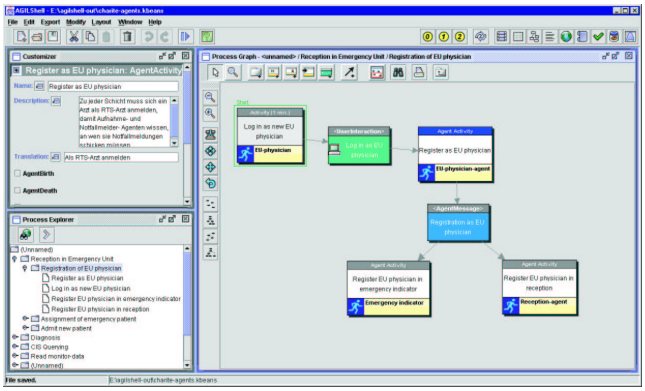

Figure 6: The AGIL-Shell, a process modeling tool with support for editing and analyzing agent interactions, and code generation features.

Beside the different views, which can help to clarify agent scenarios and to detect bottlenecks, the AGIL-Shell provides code generation features, with which Java source code can be created. We have so far only experimented with source code for our own agent prototyping platform (see below), but templates for other platforms are in progress. Note that our metamodel is rather an informal base which can be custom-tailored for specific needs. It is well possible to add new attributes or concepts that might be needed to express specific agent characteristics like beliefs and desires, or to improve code generation. The AGIL-Shell has a very open architecture that automatically provides editors when custom attributes are added to the metamodel classes.

4.2 Implementation and Test

Process modeling as described above aims at identifying and defining the types of agents, their input and output messages, and an informal description of their behavior. The implementation activities in our XP approach lead to executable source code, e.g. Java classes that implement the agents and the remaining modules. In our experiments with an XP process for agents, we used a very simple and lightweight agent platform that focuses on the main characteristics of agents, in particular their communication through messages, life-cycle management, and directory services. In order to make our experiments reproducible by other researchers, the main classes of this platform are illustrated in figure 7.

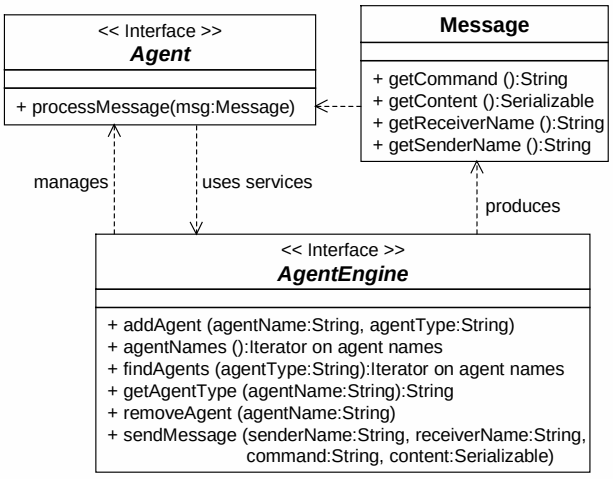

Figure 7: A simple agent engine for prototyping.

The agents on our platform must inherit the `Agent` interface and thus implement a `processMessage` method that handles incoming messages. Each `Message` encapsulates receiver and sender, as well as a textual `command` (or performative) and a `content` object (including all referred objects). The command specifies the type of message that the agent is ready to handle. The `Message` class can be extended for other types of content.

Message passing and agent life-cycles are managed by the `AgentEngine`, which provides a collection of service methods. Agents register themselves in the engine together with a unique name a type which usually corresponds to the name of its Java class. When agents intend to start other agents, to send messages, and to find other agents with given properties, they need to call the appropriate method in the engine. The engine thus serves as an abstraction layer of the com-

[1]Download from `http://www.faw.uni-ulm.de/kbeans/agil`

mon agent platform services so that other platforms can be encapsulated.

Our engine supports the spirit of XP in so far that it is extremely simple and can be used to implement and test agents rapidly. The development of automated test code is an essential cornerstone of any XP project. XP prescribes unit tests, which evaluate whether a given module implements the features it is expected to realize by the other modules. The development of such test cases is difficult in completely open and distributed systems such as multi-agent systems. In order to simplify agent testing despite these difficulties, we have developed two different implementations of our agent engine: One is based on the powerful J2EE platform (here, agents are implemented as Enterprise JavaBeans), while the second is a test engine that executes on local machines only but simulates asynchronous message-passing realistically. The agent source code can execute on both engine implementations, so that code that is tested on the test engine should also work on the (distributed) Enterprise engine.

In this sense, we are developing test cases for all agents individually, whereby all adjacent agents and system units are simulated by "dummy" objects. As illustrated in figure 8, our agent test case classes extend – as usual in XP projects – `TestCase` from JUnit [6]. In addition, each agent test case implements the `AgentEngine` interface and is therefore able to overload all methods that an agent being tested invokes in its environment. For example, the test case can overload the engine's `sendMessage` method to verify whether the agent has sent an expected response message.

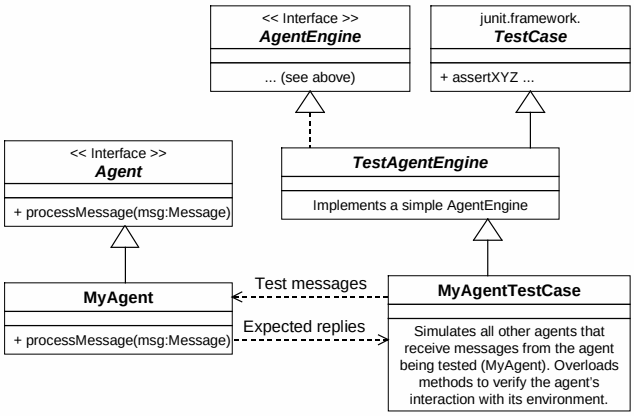

Figure 8: Testing agents with JUnit.

Template source code for agents and their test cases can be incrementally generated from the process models by the AGIL-Shell. The generated source code includes handling methods for all types of incoming messages. We generate abstract interfaces that the agent classes must overload correctly. The developers thus only need to complete the message handling methods and the test cases. This mechanism makes sure that the agent implementation is kept synchronized with the process models. Whenever the programmers identify missing features or mistakes in the specification, they simple update the model, generate the source code and insert the new method bodies. Some details on the development cycle and the application of the XP practices are provided by the following section.

5. AN XP CASE STUDY

The XP approach described in this paper is a by-product of the German research project AGIL [10], the goal of which was the prototypical implementation of a multi-agent system to optimize the distribution of clinical data for emergency patients between the various departments and decision-makers. The project involved a domain expert (an anesthetist) with experience in the analysis of clinical workflows.

In the beginning of the project it was clear that agents, with their ability to fulfill autonomous, pro-active, and distributed tasks, have the potential to optimize clinical information exchange. However, the types of agents, as well as their precise application scenarios were an open issue. Since we were faced with an existing workflow into which agents were to be introduced, we started the project with the acquisition of a model of the clinical processes that we could then modify incrementally by assigning more and more tasks to agents. This model was built by our clinical domain expert with the AGIL-Shell tool. It included 34 sub-processes with 144 activities and 26 human roles.

The next task was to derive a new process model that included an initial set of agents and their communication. For that purpose, the domain expert and the leading developer (the author) met for a three-day modeling session in which the initial application scenarios were defined with the AGIL-Shell. The direct communication and the highly informal modeling approach allowed us to visualize our ideas very quickly. The joint modeling approach furthermore helped to consider both domain and technical points-of-view, although we did not spend much time on formal details. Our visionary model equipped each of the human actors in the original process with interface agents, most of which intended to run on mobile PDA devices. We removed all human actors from the design model in order to reduce complexity and because humans were now represented by their individual interface agents. The new process included 29 sub-processes with 159 activities and 32 agent roles.

The resulting "agentified" process model was fed into an XP implementation process. For the implementation, we organized a practical course for third-year Computer Science students at the University of Ulm. The course involved eight students, a coach (the author), and the domain expert. The domain expert was permanently available to answer questions and to clarify and weigh requirements. The setting was a single office with four personal computers arranged in a circle so that the programming pairs could see each other. As shown in figure 9, the room was equipped with a beamer that was connected to the coach's laptop, so that prototype demonstrations and process models could be visualized for everyone to see.

None of the students had any prior experience or training in XP, some of them were beginners in Java and none of them had implemented agents before. We therefore used the first two days of the course to provide the students with a theoretical introduction to XP and a practical introduction to the development tools (IntelliJ and CVS). On the second day, the students learned to implement JUnit test cases for some rather simple agents. The agent engine, the modeling framework and the tools proved to be sufficiently easy to comprehend.

40-hour-week. The practical work itself was done during one 40-hour week. The students were explicitly not encouraged to work overtime. After the course, the students

Figure 9: Participants of the XP course in which a multi-agent system has been implemented in pair-programming style.

reported that they used to be quite exhausted after a full day of pair programming, but were very disciplined and concentrated while in the office. Nevertheless, the atmosphere was very relaxed and enjoyable and thus stimulated creativity and open, honest communication. We supported this atmosphere by providing free coffee and cookies and by organizing a social evening during the week.

Planning game. At the beginning of each day, the team jointly followed a planning game approach to define the features that were to be implemented next. Since the process model described the phases of a patient's treatment on her way through the hospital in a rather sequential style, we found it most useful to implement the agents in their order of appearance within the process. We locally focused on those agents that – according to the domain expert – promised the most business value.

Pair programming. The agent implementation itself was then assigned to the developer pairs. The code generator described above was used to create one package and template code for each agent. Each pair had to develop and test their individual agent in isolation, using the testing framework from subsection 4.2. The students found pair programming very enjoyable and productive. One student reported later that he felt much more motivated and concentrated than if he had to work alone. Most students reported that they learned many useful programming techniques from each other during the course. However, due to the different experience and background of the team members (e.g., some were not knowledgeable of GUI programming), not all student combinations proved to work equally well. The students' individual experience had a significant impact on code quality, so that the coach tended to delegate difficult tasks to the advanced programmers.

Testing. Our agents proved to be quite easy to test, because their behavior and state changes mostly depended on incoming messages only. Many tests therefore consisted of sending a test message to the agent and of checking whether the expected reply message was delivered back. The students found testing quite useful to clarify requirements although it was considered to be additional work by some. During the course, the students have implemented 30 TestCase classes with 76 test cases, amounting to 4909 lines of code, while the 43 agent source code classes amount to 5673 lines (excluding ontology classes). The students enjoyed using JUnit very much, because the sight of the green bar that indicates correct test runs improved motivation and trust in the code. The main problems with testing were that some refactorings were needed to enable testing of some functionality, and that automated tests of visual interface agents are difficult.

Collective ownership. We applied a relaxed practice of collective ownership, which allows any team member to modify any piece of code at any time. Since each pair only operated on the source package of a single agent, there was barely any overlapping. Only the ontology classes were shared among agents and thus modified by various teams. Coordination of these changes was accomplished very informally by voice and the CVS.

Coding standards. In the beginning of the project, we defined a project-wide coding standard that was very easy to follow, because the Java tool we used provides automated code layout features. Thus it was very simple to shift implementation tasks between the pairs and to change pair members regularly.

Simple design. The students were explicitly asked to focus on programming speed instead of comprehensive upfront designs. This seemed to be sufficient because the agents were rather small units with few types of tasks to solve. Despite the focus on simplicity, experienced students almost automatically identified some useful generalizations of agent functionality. Our initial process model underwent several evolutionary changes, in particular we frequently encountered scenarios where agents were unable to fulfill their task because they did not have access to data or information that other agents possessed. We therefore had to add some activities and messages that pass missing data items between agents. In those cases, we could generate the new source code and ontology classes quite rapidly and feed them back into the XP process. Despite the various small changes, the overall design remained quite stable throughout the project, so that our process modeling framework proved to be sufficient.

Refactoring. Since the agents were rather small units, they were very easy to maintain and refactor. Some medium-sized refactorings, like the introduction of new superclasses to generalize functionality, were performed. Smaller refactorings, like the introduction of temporary variables, were induced by the coach when he found code too hard to read. Some other refactorings were necessary to enable the implementation of automated tests.

Continuous integration and short releases. The agents were uploaded onto the CVS server and integrated at least every evening. Since the students were only allowed to upload those agents that passed all test cases, there were almost no integration problems. Agent interactions were tested and presented on the beamer with the help of a small simulation environment that could trigger external events. A student later described the integration shows as the daily highlight, because the agents that were programmed and evaluated in isolation suddenly interacted with real other agents.

On-site customer. In the questionnaires that were filled

out by the students after the course, the presence of the domain expert was very positively evaluated. He was asked to provide clinical knowledge regularly, at least once an hour, so that expensive design mistakes were prevented. His presence did not even mean an overhead for him, because he could use the "spare time" for other types of work on his own laptop. We found that the communication process is characterized by reciprocities between engineers and the experts. As the domain expert got more and more used to the formal view of the developers, he adjusted his modeling style, and vice-versa.

The XP project resulted in the full implementation of 17 agent types, some of which with complex graphical interfaces. These agents cover about the first third of the original process model and solve tasks like notification, information filtering, and patient monitoring. The agents make use of 38 ontology classes. Details on the original and final scenarios can be found on the course homepage[2]. In the implemented excerpt of the overall process, 5 additional agent types were identified in the coding phase, in particular agents that provided generic services for other agents. The agent scenarios, however, underwent much more significant changes. The implemented model includes almost twice as many activities and agent messages as originally designed. The additional activities often concern information transfer between agents, because many agents lacked access to resources that they required to solve their tasks. About half of the additional messages, however, add new application scenarios and functionality that was originally not thought about. This functionality had emerged from the creativity of the team members during the implementation phase.

6. DISCUSSION

The significant increase of the process model's size during the implementation phase indicates that our initial agent design was inadequate. This is no surprise, because we intentionally spent only very little time on this pre-modeling process, so that we were able evaluate XP with a realistic starting point. However, it is yet unclear whether a traditional AOSE approach would have produced a better model more efficiently, because we did not have the opportunity to run corresponding modeling experiments yet. Such experiments and independent comparisons are certainly necessary to clarify whether agile approaches like XP are in fact superior to methodologies with a comprehensive up-front design. We can therefore base our discussion of XP only on our subjective experience, and by drawing parallels between multi-agent systems and reports from mainstream software technology.

Retrospecting on the case study, we are very pleased with our first XP experience. Although neither the team members nor the coach were experienced in XP or agent development, the approach has shown to be quite efficient for the development of a prototypical, medium-sized multi-agent system. All team members reported that XP was in general much more enjoyable than following a strict, rather bureaucratic methodology. Student feedback about the course was overwhelmingly positive. The team worked highly concentrated and disciplined, and the fluent communication within the team fostered creativity. The personal relationships between the students were intensified and the team members felt responsible for "their" agents. Although not all students were equally well team players, this clearly indicates that XP is quite a natural way of developing agents, which works with people's instincts and not against them.

Following the successful first XP course, we have conducted a second case study in April, 2002. This project involved 12 students and completely strengthened our approach. Additionally, the second XP course demonstrated that regular code reviews by changing pairs of programmers can significantly improve code quality. Furthermore, the course has shown that in addition to unit tests, where agents are tested in isolation, programmers should write automated *integration tests* which verify complex scenarios involving multiple agents. These tests can be derived from the process models.

Our observations about XP are consistent with industrial reports on the development of "conventional" software [13]. However, care must be taken to simply generalize those positive reports to the domain of agents, because multi-agent systems have specific properties that should be reflected in the methodology. Therefore, a theoretical assessment of whether and when XP is suitable for multi-agent systems should be derived from the specific attributes of agent systems.

- The single agents are typically rather small and loosely-coupled systems, which solve their tasks in relative autonomy. As a consequence, writing automated test cases is quite easy for agents, because the single agents have a small, finite number of interaction channels with external system units.
- Simple solutions, like those suggested by XP, appear to be sufficient for most agents, because the agents themselves are quite small. Even if an agent evolves into a performance or quality bottleneck after a series of refactorings, it is possible to completely rewrite it from the scratch (following the XP value of courage) without risking the functionality of the overall system.
- Agents, like those in our clinical scenario, represent various types of human process participants with individual goals and knowledge. In order to map those multi-facet viewpoints onto a consistent system, people from the domain must be involved very closely into the development process – and they must communicate with each other. In XP processes, domain experts and end users are neatly integrated with rapid feedback, fluent communication, and in the requirements planning cycle.
- The close involvement of real users allows to clarify misunderstandings and terminology much faster than in engineering approaches, in which users are comparably excluded from the analysis and design phases.
- It appears to be very probable that the agent requirements will change. The agent modeling process itself often produces new knowledge about potential agent scenarios, and the self-observation performed during analysis of the existing work processes into which agents are to be introduced can lead to new insights [12]. For the new system, process knowledge is being translated and reorganized, and thus evolves. The existing

[2]`http://www.faw.uni-ulm.de/kbeans/knublauch`

work processes are challenged when analyzed ("Redesign during modeling" [12]).

- Systematic engineering approaches are difficult to apply to agents, because the resulting analysis and design models are often based on wrong assumptions. When agents are intended to solve tasks on behalf of human stakeholders with all their individual beliefs, desires, and intentions, the humans are required to transparently expose their daily practice. However, this "practice necessarily operates with deception" [12], so that mental processes can often only be incrementally translated into agent programs.
- The autonomy and "intelligence" of agents allows to characterize many multi-agent systems as distributed knowledge-based systems. Such systems are best designed in an evolutionary style with rapid feedback [12].
- Frequently, agents are experimental systems, for which bureaucratic processes are too rigid.

While the above reasons indicate that agile approaches will have a significant impact in agent technology, methods like XP are not the best choice for all kinds of projects. In cases where requirements are quite stable and well-understood, or where agents are to be introduced into a fixed environment with only little inter-agent communication, a systematic process will most probably be a better choice, because it is easier to plan and manage. Due to its barely traceable process, XP is also considered to be not the best option for the development of highly generic and reusable components, and for safety-critical systems [2]. Last but not least, it is important to note that XP can only be applied if management and customers are convinced of its benefits. More positive case studies from industry (e.g., those found in [13]) and progress on the theoretical foundation of XP will help achieve this commitment easier.

7. CONCLUSION AND FURTHER WORK

This document has provided evidence that agile approaches like XP are suitable for the development of many multi-agent systems. Compared to traditional AOSE methods, XP involves end users and multiple viewpoints much closer into the process, produces feedback and executable systems more frequently, and puts a stronger emphasis on system evaluation. Last but not least, our case study has shown that an XP process is very enjoyable and motivating for the developers, because it works with people's instincts.

However, further work is needed to strengthen our results in comparison to traditional AOSE methods. We made several simplifying assumptions that need to be removed in order to generalize our approach. In particular, we have implemented the agents against our very light-weight agent platform that simplified testing. We are therefore currently generalizing our test and code generation approaches for a FIPA-compliant standard agent platform.

Acknowledgments

The author would like to acknowledge the eager participants of the XP courses at the University of Ulm in fall, 2001, and in spring, 2002. Many thanks to Holger Köth, M.D., for providing valuable clinical domain knowledge in the AGIL project. AGIL is being funded by the German Research Foundation (DFG) in the context of multi-agent research.

8. REFERENCES

[1] B. Bauer, J. Mueller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. In P. Ciancarini and M. Wooldridge, editor, *First Int. Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, Limerick, Ireland, 2000.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering*. Springer-Verlag, Berlin, Heidelberg, New York, 2001.

[4] S. A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems. In *Proc. of Agent Oriented Information Systems*, Seattle, OR, 1999.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, 1999.

[6] E. Gamma and K. Beck. JUnit: A regression testing framework. `http://www.junit.org`, 2000.

[7] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Int. Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[8] H. Knublauch and T. Rose. Round-trip engineering of ontologies for knowledge-based systems. In *Proc. of the Twelfth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Chicago, IL, 2000.

[9] H. Knublauch and T. Rose. Tool-supported process analysis and design for the development of multi-agent systems. In *Proc. of the Third Int. Workshop on Agent-Oriented Software Engineering (AOSE-02)*, Bologna, Italy, 2002.

[10] H. Knublauch, T. Rose, and M. Sedlmayr. Towards a multi-agent system for pro-active information management in anesthesia. In *Proc. of the Agents-2000 Workshop on Autonomous Agents in Health Care*, Barcelona, Spain, 2000.

[11] J. Lind. *Massive: Software Engineering for Multiagent Systems*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[12] W. Rammert, M. Schlese, G. Wagner, J. Wehner, and R. Weingarten. *Wissensmaschinen: Soziale Konstruktion eines technischen Mediums. Das Beispiel Expertensysteme*. Campus Verlag, Frankfurt, Germany, 1998.

[13] G. Succi and M. Marchesi, editors. *Extreme Programming Examined*. Addison-Wesley, 2001.

[14] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6), 1996.

[15] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.