TAMPERE UNIVERSITY OF TECHNOLOGY

MIKKO VARTIALA

DESIGN AND IMPLEMENTATION OF AN AGENT-BASED ARCHITECTURE FOR A PROCESS SUPPORT SYSTEM

Master of Science Thesis

Examiners: professor Kai Koskimies
assistant professor Jari Peltonen
Examiner and topic approved in the Faculty of Computing and Electrical Engineering Council meeting 17.8 2005

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
VARTIALA, MIKKO: Design and Implementation of an Agent-Based Architecture for a Process Support System
Master of Science Thesis, 50 pages
March 2010
Major: Software engineering
Examiners: Professor Kai Koskimies and assistant professor Jari Peltonen
Keywords: Software process support, software agents, software framework, agent-based architecture

Tool integration is an important aspect of software development process support. In such systems it should be possible to integrate tools flexibly and incrementally. In addition, for performance and usability reasons, it should be possible to use the tools both on local and remote computers.

To address this problem of flexible tool integration, an agent-based architecture style was designed. The architecture strives to attain the needed flexibility by few simple design rules. One of the rules is to divide the functionality to agents and locations. The locations work as adapters to tools and provide basic infrastructure of the system. The agents move among the locations and implement the high level business logic of the system by using the methods of the locations. A general principle is that each agent implements a single business case. This makes it easy to view, control, and adapt the high level business logic as the logic is located in one place.

The architecture style is not tied to any specific programming language. However, for the purposes of this thesis an agent-based software framework was implemented using C++. A distributed process support system was then implemented by specializing the agent framework. The process support domain provides a good case study for the validity of the agent-based architecture as the process support system needs to integrate various tools supporting the process.

As a result of this thesis, an agent-based architecture style was designed and prototyped. The implementation of the process support system was used to evaluate the agent-based architecture style and to find out the challenges in building systems using the principles of the agent-based architecture. The architecture could be extended in many ways, but it was shown to be usable in the domain of tool integration. In addition, the implemented process support system fulfilled the quality requirements laid out for it.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
VARTIALA, MIKKO: Agenttipohjaisen arkkitehtuurin suunnittelu ja toteutus prosessitukijärjestelmälle
Diplomityö, 50 sivua
Maaliskuu 2010
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Kai Koskimies ja yliassistentti Jari Peltonen
Avainsanat: ohjelmistoprosessituki, ohjelmistoagentit, ohjelmistokehys, agenttipohjainen arkkitehtuuri

Työkaluintegrointi on ohjelmistotuotantoprosessien tukemisen kannalta olennaista. Työkalut olisi myös hyödyllistä saada integroitua joustavasti ja inkrementaalisesti, työkalu kerrallaan. Esimerkiksi ohjelmistoprosessitukijärjestelmän on tärkeää olla helposti muokattava ja erilaisiin tilanteisiin mukautuva, jotta ohjelmistokehittäjät eivät kokisi sen käyttöä taakaksi, vaan omia työtehtäviään helpottavaksi.

Tässä diplomityössä suunniteltiin työkalujen integrointiin hajautusta tukeva agenttipohjainen arkkitehtuurityyli. Arkkitehtuurityyli pyrkii saavuttamaan sille asetetut laatutavoitteet muutamalla selkeällä pääperiaatteella, esimerkiksi jakamalla toiminnallisuuden agentteihin ja sijainteihin. Sijainnit toimivat muun muassa sovittimina työkaluihin ja tarjoavat yleistä järjestelmän perustoiminnallisuutta. Agentit liikkuvat sijaintien välillä ja toteuttavat järjestelmän korkean tason liiketoimintalogiikan käyttämällä sijaintien tarjoamia metodeja hyväkseen. Yleisenä periaatteena on yhden käyttötapauksen sijoittaminen yhteen agenttiin, jolloin korkeimman tason liiketoimintalogiikan hallinnasta ja muokkaamisesta tulee helppoa.

Lähestymistapaa arvioitiin toteuttamalla agenttipohjaisen arkkitehtuurityylin periaatteita noudattava C++ ohjelmistokehys. Lisäksi tätä ohjelmistokehystä erikoistamalla toteutettiin hajautettu prosessitukijärjestelmä. Prosessitukijärjestelmän kokonaistoiminnallisuus saavutettiin integroimalla siihen useita jo olemassa olevia ohjelmistoja.

Työn tuloksena saatiin suunniteltua työkalujen integrointiin tarkoitettu agenttipohjainen arkkitehtuurityyli. Lisäksi luotiin Tampereen teknillisen yliopiston Ohjelmistotuotannon laitoksen käyttöön prototyyppi C++ agenttiarkkitehtuuri-kehyksestä ja prosessitukiympäristöstä. Ohjelmistokehyksen päälle toteutettu prosessitukiympäristö auttoi tarkistamaan agenttipohjaisen lähestymistavan toimivuuden tässä kohdeympäristössä. Lisäksi prosessitukiympäristön toteuttaminen havainnollisti agenttilähestymistavan mukanaan tuomia hyötyjä ja haasteita.

PREFACE

I would like to thank my colleagues and the participants of the original project group this work was started with for their professional support. I would also like to thank the examiners of this thesis, Kai Koskimies and Jari Peltonen, for the invaluable guidance and comments provided for this work. In addition, I would like to thank my family and friends for their support and the motivation provided by their constant enquiries about the status of this work. Finally, I would like to thank Salla for her support and for enduring the time consuming finalizing of this writing work.

Tampere, 18. March 2010

Mikko Vartiala

CONTENTS

TERMS AND DEFINITIONS

CASE Tool	Computer-Aided Software Engineering tool. CASE tools are tools that help the development of software products.
MOF	Meta-Object Facility. MOF is a standard for model-driven engineering. MOF Is used to define UML.
UML	Unified Modeling Language. UML is a modeling language for software systems.
COM	Component Object Model. COM is A technology developed by Microsoft to enable software components to communicate with each other in Windows environments.
API	Application Programming Interface. An application programming interface is an interface enabling other applications to interact with the application providing the interface.
XML	Extensible Markup Language. XML is a textual data format designed to be usable over the Internet.
RPC	Remote Procedure Call. RPC is an inter-process communication technology allowing applications to call other applications.
SOA	Service Oriented Architecture. SOA is a set of architectural principles designed to provide ease of integration of services.
ODBC	Open Database Connectivity. ODBC is a way for software programs to connect to and use database management systems.
COTS	Commercial, Off-The-Shelf. A COTS component is a software component that is readily available for sale to general public. In some cases COTS can also refer to common, off-the-shelf, i.e. including free software.

1. INTRODUCTION

Software process support systems aim at helping the developer to carry out the various activities in a software process more efficiently. Efficiently can mean, for instance, in less time, with better quality, or to overall use less money by, for example, using cheaper tools. However, the software processes used in different software projects vary greatly, and often there is a need to make ad-hoc changes to the process even during a project. Therefore, a process support system must be flexible and maintainable to be usable in real world scenarios. Especially it must be possible to integrate new and existing tools to the process support system easily.

Software process tools and software process support have been a target of research in many projects in the Software Systems Department of Tampere University of Technology. At the start of this work there already existed various tools, including a graphical editor and an engine used to create and run VISIOME scripts [Pel00]. VISIOME scripts can be used to define various kinds of processes. However, the existing tools were not integrated together very well, and there was also a need for additional functionality. For example, there was a need for a user interface that could be used to follow and control the execution of the process. The existing engine running the process was an executable run on a single computer and therefore did not support distribution. In addition, there was a need for concepts not supported in the existing application, including projects, user roles, and guidance for activities. In essence, there was a need for a process support system that would integrate the existing applications together and add a project-related information layer on top of them.

The integration of existing applications and tools is a challenge that concerns not only process support systems, but also many other domains. In many areas of software development it is possible to use existing applications. Good examples of these are various open source applications readily available to any developer. However, rarely do these single applications alone offer the complete needed functionality. In such cases it is usually a better solution to try to integrate these applications together than to try to create a whole new application from scratch.

To answer these challenges it is important that the various applications, and in the case of process support systems, especially the various tools, can be integrated together in a flexible and maintainable way. For these reasons an agent-based architecture style for application integration was designed in this thesis. The architecture style is designed to work primarily in the domain of integrating tools in software development support.

The agent-based architecture was validated by first building a prototype framework using the design principles of the agent-based architecture and then implementing a process support system by specializing the framework. The implemented process sup-

port system utilizes the good points of the agent framework to fulfill the growing demands of the software development process, by, for example, providing easy integration of existing and new tools to the support system.

Chapters two and three introduce the theory and background behind this work. Chapter two is about the general architectural concepts needed in this thesis, and chapter three is more specifically about the process support domain. In chapter four the agent based architecture is described. Chapter five is about the implementation of the agent framework, which was described in chapter four. Chapter six describes the case study process support system, which was built using the agent framework. In chapter seven the pros and cons of the architecture are discussed and related work is presented. Chapter eight presents the conclusions of this thesis.

2. SOFTWARE ARCHITECTURES AND AGENTS

There are various architectural concepts and techniques used in this thesis. Examples of these include software frameworks, agents, and observer-pattern. They are introduced briefly in the following sections.

2.1. Software Architecture

Software architecture is usually understood to mean at least the structure of a system, including communications between the modules in the structure and the dynamic behavior of the system. In addition, an important purpose of the architecture is to define and guide how the system should be built and extended over time, i.e. a kind of a constitution or a philosophy of the implementation of a system [Kos05, Hai06].

Usually a good architecture means that if a developer does not know something about the design of a system, then she can make an educated guess about it on the basis of the architecture philosophy. An architecture philosophy known to work well is also known as architectural pattern. A good example of an architectural pattern is the Model-View-Controller [Bus96] architecture. [Hai06]

2.1.1. Motivation for Software Architectures

To enable larger projects, faster development, and higher productivity there has always been the need to raise the abstraction level in software development. Sophisticated architecture styles and models have helped to achieve this goal by, for example, making it possible to better communicate ideas and to allow developers to concentrate more on the big picture instead of small things.

The rise of the abstraction level has allowed software developers to see the similarities in seemingly different kind of systems, which then allows these similarities to be implemented in one place, making greater amount of reuse possible. In addition, incremental development and the splitting of software development to reasonable work units are qualities that can only be enabled by architecture level solutions. [Kos05]

2.1.2. Software Frameworks

Gamma et al. [Gam94] describes a framework to be a set of cooperating classes that make up a reusable design for a specific class of software. The purpose of a software framework is to allow large scale software reusability in a specific domain area. The difference between frameworks and normal reusable class libraries is that a software framework also reuses architectural design decisions and basic functionality. More spe-

cifically, a framework is usually an almost whole program, where the developer fills the missing gaps according to her needs. This is called specialization of the framework, and the missing gaps are called extension points.

A general problem in developing software frameworks is the decision about target scope. A framework with a too limited scope is in practice a single program, and an all domains covering framework is also called a programming language. To find a good and well balanced tradeoff between these two is a job needed to be done before actually developing a framework.

Benefits of software frameworks include faster development, better quality, and easier developer migration to new projects. Faster development is achieved by reusing existing code [Kos05]. Better quality of code is accomplished because the framework has already been tested in previous products. Possible disadvantages include bloating of code, possibly poorer efficiency, and added complexity of the resulting system.

The types of frameworks include white-box, black-box, and plug-in frameworks. White-box framework is a framework that is open for the developer, i.e. the developer knows the primary structure of the framework and specializes the framework by inheriting classes from the base classes in the framework. A black-box framework is a framework that has already reached such a stage in evolution that the developer does not add any new code related to the framework. Only some initialization parameters and such are given, and then the working program is created by configuring the framework with the wanted set of properties. A plug-in framework is a framework that is mainly extended by creating new plug-ins that implement a certain plug-in interface. The plug-ins are usually loaded dynamically from the file system, so that the whole software does not need to be recompiled each time a plug-in is added. [Kos05]

2.2. Application Integration

Application integration means making different applications to work together. There can be many different levels of cooperation, for example, the applications can only share some of their data, or they can be fully cooperating and reacting to the behavior of each other in real-time. In this section the reasons why messages have been popular in application integration is discussed, and finally the downsides of message based systems in integration are looked into with more detail.

2.2.1. Why Messages in Application Integration?

Messages are often seen as the most versatile option for application integration over file transfer, shared database, and remote procedure calls (RPC) (e.g. [Hoh03]). File transfer and shared database approaches are solutions for sharing data, but not functionality. RPC again makes it possible to share functionality, but couples the applications tightly to each other at the same time. In addition, remote procedure calls are slower and much more likely to fail than local ones, and due to the synchronous nature of communication, a failure in one application may break down the whole system. File transfer, as an

integration approach, is asynchronous and decouples applications well, but does not transmit the data in real time.

Messaging aims at mixing the good attributes of file sharing and RPC by allowing near to real time data transmission and functionality invocation asynchronously. Asynchronous communication is one of the key points when aiming at loose coupling among applications. Sending a message does not require all participating systems to be available at the same time, and the sender does not have to wait for the response, but it can continue on doing other things. In addition, any procedure calls a message actuates are local, which makes the system more reliable.

Architectural styles like Service Oriented Architectures (SOA) [Pap03] and Enterprise Service Bus (ESB) [Cha04, Kee04] emphasize loose coupling by relying on indirect asynchronous message based communication. They work conceptually on higher level than, e.g., traditional client-server architectures, since they do not discuss physical clients or servers, but logical services and their consumers. This detaches the architectures from physical world, and thus from physical addresses. The service consumers also tell what services they want, not how they will be performed. Higher level of abstraction in dependencies is a favourable solution in application integration since it makes loose coupling as the central approach in the architecture.

2.2.2. Deficiencies of Message-Based Systems

In a message based system, a close to real time communication is achieved by sending a lot of small messages and letting the receiver to know immediately when a message is available. This generates easily a lot of network traffic, which may become a problem in larger and more complex systems. In addition, not all of the messages are small and simple, since they are used to transmit all the information in the system. Hence, messaging may put a heavy burden on a communication channel. This is a problem, not only in environments where the communication channels are thin (like mobile environments), but in any environment. Basically, due to need to minimize the network traffic, high granularity in services would be favourable. However, reuse of services would benefit from lower granularity.

Due to various schemas and data formats in different applications, each message goes through a transformation chain, where the message is first formulated, translated to a common format and sent, and in the other end it is received, parsed, interpreted and actuated. This requires some processing power, as well as causes lag for the communication. In addition to the minor inconveniences caused by latencies, the total completion time may grow considerably.

Since the message must be interpreted in the receiver end, both the sender and receiver must understand the exact semantics of the message. This means that a single concern in functionality is always divided across the architecture, and the comprehension, maintenance, and testing of such a concern gets very hard. The problem is even worse when the needed functionality is complex, and there is a need for several messages to get a single thing completed.

Basically, any sequence of service requests in a message is a sequence of commands and can hence be considered as a script. The language for specifying a script just does not have the power of typical scripting languages. There are no other ways in messages to react dynamically for varying or exceptional situations either. Not very much can be done, for example, if a service fails during the execution. The service may be able to send an error message to the service consumer, but again, an amount of messages are sent to various places. In addition, there must be some code to react to that kind of messages too – in all the service consumers who might be interested.

As an example, let us consider a situation where a service consumer wants to calculate a trend based on a large amount of information that is divided on several services. This means that there are several related messages either sent one by one to the services and then the results are collected and interpreted in the consumer, or there is a chain of messages where the information from a previous service is forwarded to the next one, and the following service again interprets the data it gets.

Particularly, if the data divided on the different services depend on each other in the calculation, or the way of performing the calculation is dynamic (e.g., depending on the consumer or data provided by the services), there is either a huge amount of network traffic, or the services become unnecessary complex. Either way, the functionality needed for performing a single calculation is spread across the architecture, the business sequence gets hard to comprehend, maintain, and test, and it is hard to get the whole system robust and fault tolerant.

2.3. Agent-Based Systems

As discussed in section 2.1.1, the rise of abstraction level has allowed significant improvements in software development. Such paradigm shifts include moving from procedural programming to object-oriented development. Many argue that the notion of autonomous and goal-oriented entities, agents, and multi-agent systems offer a similar paradigm shift [Jen01, Zam03]. However, there are many challenges in developing agent systems [Woo98]. The possible benefits offered by agents answer to some of the deficiencies described in section 2.2.2, but on the other hand they create a handful of new ones.

In this section, first a look at the basics of agents and mobility is given, and then the benefits and drawbacks of mobility are discussed in more detail. Finally, the challenges of building agent systems are discussed.

2.3.1. Definition of an Agent

Stan Franklin and Art Graesser [Fra96] define the essence of being an agent as follows: *"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."* Moreover, they note that this definition of agent by itself is not very useful, but further classification is needed. Their classification is listed in Table 1. Additionally, Franklin and Graesser specify that, by their definition,

all agents fulfill the four first listed properties and the five bottom properties are a kind of bonus properties, which can add more usefulness to an agent.

Another way to distinguish between different types of agents is to classify existing agents into different categories. This kind of a categorization is done by Nwana [Hya96]. Nwana classifies agents by whether they are static or mobile, deliberate or reactive and by several primary attributes the agents should implement. Nwana specifies that a minimum of three attributes is needed: autonomy, learning, and cooperation. These three are used in Figure 1 to derive four more specialized agent types. The actual figure is made by Chua [Chu03]. The specialized agent types are interface agents, collaboration agents, collaboration learning agents, and smart agents. It is emphasized that these definitions are not absolute, but more of a guideline to classify agents according to their primary attributes. Nwana also notes that agents may be categorized by their roles, e.g., an Internet agent, and whether they are hybrid agents, i.e. if an agent combines multiple agent philosophies together. Additionally mobility and deliberation could be added to the fore mentioned agent types to create an even more specialized list of agent types.

2.3.2. Mobility

Table 1 defines an agent to be mobile if it can transport from one computer to another. In general, this means that instead of sending messages or using RPC to communicate over network, an agent itself is sent over network. Therefore when a need arises, e.g., it needs new information or has a new task to achieve, it is free to use the network to transport itself to a new host and continue execution in there. There are several different ways to achieve mobility. The minimal way is to require the host to have the execution code in advance and to only transfer the initialization parameters of an agent. On the other hand the most requiring method is to transfer the execution code and the execution state of the agent to the new host. Transferring the execution code and the execution

Table 1: Classification of agents

Property	Other Names	Meaning
Reactive	sensing and acting	responds in a timely fashion to changes in the environment
Autonomous		exercises control over its own actions
goal-oriented	pro-active, purposeful	does not simply act in response to the environment
temporally continuous		is a continuously running process
Communicative	socially able	communicates with other agents, perhaps including people
Learning	Adaptive	changes its behavior based on its previous experience
Mobile		able to transport itself from one machine to another
Flexible		actions are not scripted
Character		believable "personality" and emotional state

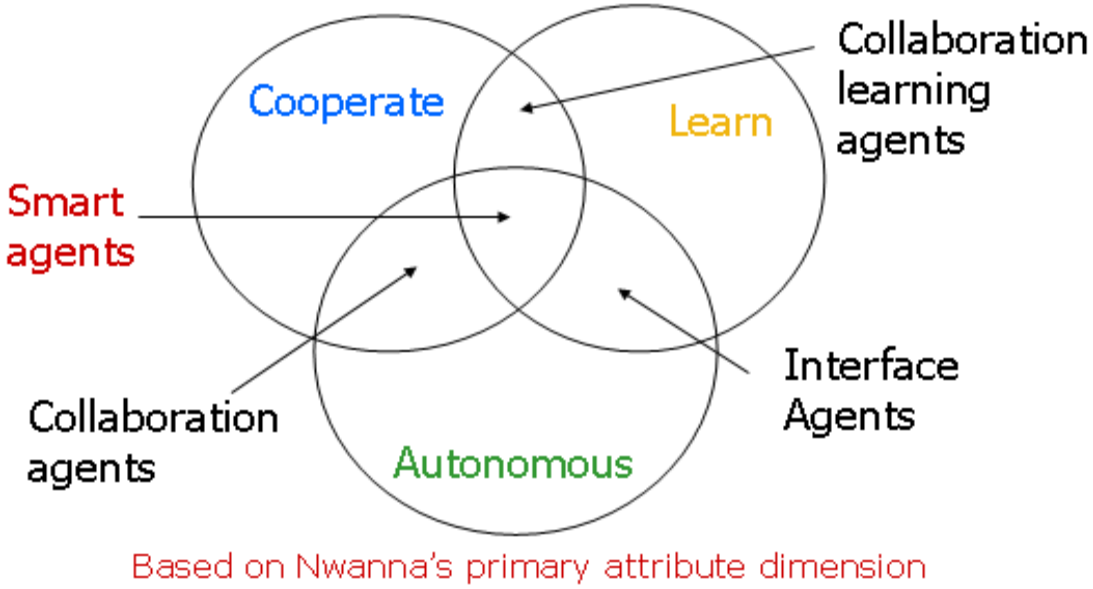

Figure 1 Typology of agents by Nwana [Chu03]

state is called strong mobility, and transferring only the code and possible initialization parameters is called weak mobility.

The primary motivation for using agent mobility should be the benefits it provides, not the technological finesse of using the technology just because it is possible. Lange and Oshima [Lan99] lists seven good reasons for mobile agents: they reduce network load, they overcome network latency, they encapsulate protocols, they execute asynchronously and autonomously, they adapt dynamically, they are naturally heterogeneous, and they are robust and fault-tolerant.

Even though network bandwidth is growing continuously, the reduction in network load is still a needed benefit, as at the same time the amount of data needed to be processed is growing enormously. Mobile agents can be used to reduce network load by, instead of moving data to the agent, moving the agent to the data. In addition, moving the agent to the data helps overcoming network latency. This is critical in real-time systems, but additionally the execution time of complex data processing can be significantly reduced. The reduction is achieved because, instead of having to always wait for new data after making a decision based on previous data, the agent can immediately query the host for new data without any network delays. Asynchronous and autonomous execution provides mobile agents the benefit of being independent from the original creator. For example, if launched from a laptop to another computer, the agent can finish its task even if the laptop becomes disconnected from the network. More generally, the robustness of agents is increased as the agents can react dynamically to unexpected situations like the fore mentioned disconnection of the laptop.

2.3.3. Challenges in Developing Agent-Based Systems

There are many possible dangers in developing agent-based systems. Wooldridge et al. [Woo98] divide the pitfalls into seven different categories: political pitfalls, manage-

ment pitfalls, conceptual pitfalls, analysis and design pitfalls, micro (agent) level pitfalls, macro (agent) level pitfalls and implementation pitfalls. The four last pitfall categories are more related to the actual development of an agent-based system and are therefore the most related to the work done in this thesis. The most relevant challenges in these four categories are summarized and discussed next, excerpted from Wooldridge et al. The situations described here are not automatically mistakes, but situations where great care needs to be given to avoid the pitfalls. Chapter 7 includes a section where the work done in this thesis is reviewed in light of these pitfalls.

Analysis and design pitfalls

One of the pitfalls in designing an agent-based system is trying to do everything yourself with new agent-styled techniques. This leads to slower development and lower quality software than exploiting related technology where applicable. For example, existing platforms for distributed computing and database systems are technologies applicable to many agent systems.

Micro (agent) level pitfalls

Wooldridge et al. lists four relevant pitfalls in this category: building your own agent architecture, believing your architecture is generic, using too much artificial intelligence, and having agents with no intelligence. They are described briefly in this section one by one.

Building your own agent architecture has all the same risks as a typical complex software systems development. In general, developing a distributed system takes time and effort and is error prone. It is suggested in Wooldridge et al. to first study the existing agent architectures and see if any of them is sufficient.

Believing your architecture is generic is an easy mistake to do. After developing a sufficiently good architecture, it can be tempting for the developers to believe that the architecture is suited for more domains and problems than it actually is. It is suggested that before trying to apply an existing agent architecture to a new problem, the characteristics of those domains are reviewed in depth to see if the problem domains really are similar enough.

Having the agents use too much AI is related to the more general software analysis problem of bloated specifications with a lot of nice to have features. In a similar fashion, it should be analyzed, which AI properties are really necessary for the system to work, and start with those. After the system has been built successfully, the intelligence of the agents can be evolved when necessary.

Having no intelligence on the agents is more of a concept related problem than an actual agent problem. For example, calling any complex distributed system a multi-agent system confuses the meaning of agent systems and makes it harder for developers to understand each other.

Macro (agent) level pitfalls

Possible dangers in this category include seeing agents everywhere, having too many or too few agents, spending all time implementing the infrastructure, and having an anarchic system. The first two are related, as seeing agents everywhere can lead to dividing the system to smaller and smaller pieces, until every piece of computation is an agent, i.e. having too many agents. Having too many agents leads to systems that are hard to maintain and whose dynamic behavior is difficult to predict. In addition to reducing the amount of the agents, another way to reduce the complexity of the system is to constrain the ways the agents can communicate. This is additionally one of the solutions to the related pitfall of having an anarchic system, i.e. a system where the agents have just been thrown in on the assumption that no agent hierarchies or constraints are needed. In addition to having too many agents, it is also possible to build a system with too few agents, i.e. having a too monolithic application.

Implementation pitfalls

Two possible pitfalls in this category are listed in Wooldridge et al. The first danger is thinking that it is necessary to implement the whole system from scratch. The second danger is the danger of ignoring the de facto standards. The difference between the first danger, implementing the whole system from scratch, and the danger described under Analysis and design pitfalls, i.e. trying to do everything yourself with agent technologies, is that here it is not merely talked about technologies, but, for example, of proprietary components developed over many years. It is unnecessary, and usually impossible in the timeline of integration projects, to replace such components. A solution offered is to wrap the legacy components with an agent layer that converts the communication to and from the agents to the legacy component.

2.4. Software Architecture Related Techniques and Concepts

In this section two architectural concepts are briefly presented. Both of the concepts are used in this thesis in relation to the agent-based architecture.

2.4.1. Metalevels in Software Design

In software design the term meta- can be understood to mean the abstraction of concepts. For example, the real world is classified with abstract concepts such as animals, dogs, mammals, etc. The real, living animals can then be viewed as instances of these concepts. In a similar way, software architectures can be defined in several different meta-levels. In such a definition each meta-level is built using the concepts defined in the more generalized meta-level. For example, the UML language is defined this way.

An example of the meta-levels in a UML model is shown in Figure 2. The figure is layered in a way that Meta-Object Facility (MOF) [OMG06] is the metametamodel, which is used to specify the metamodel, i.e. the model of UML language [OMG07].

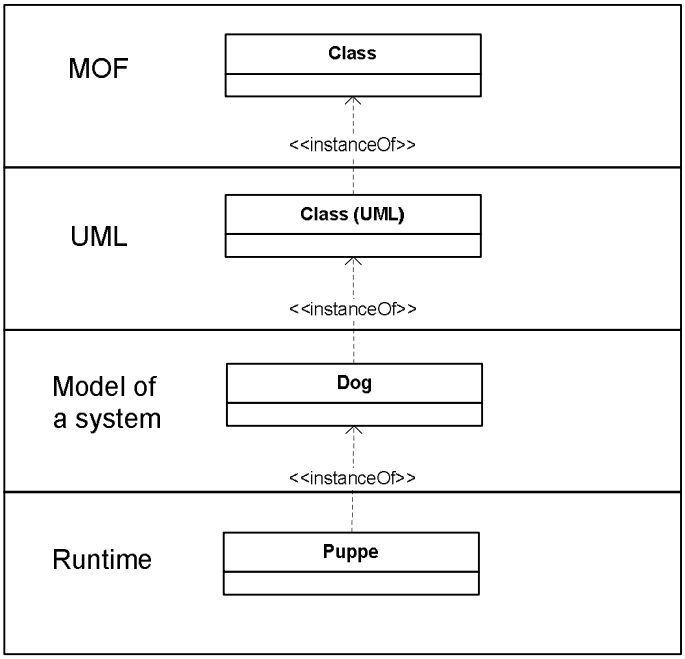

Figure 2 Example of metalevels in UML

The UML language is then used to specify the models used in actual systems. The instantiations of the elements of that model are the actual objects that are created in a program during run-time.

2.4.2. Observer Pattern

Observer pattern is commonly used in situations where one participant, the observer, is interested in the changes of data in another participant, called the subject. Buschmann et al. [Bus96] lists the following forces that should be balanced by the pattern:

- One or more components must be notified about state changes in a particular component.
- The number and identities of dependent components is not known a priori, or may even change over time.
- Explicit polling by dependants for new information is not feasible.
- The information publisher and its dependents should not be tightly coupled when introducing a change-propagation mechanism.

In simplicity, the solution is that the interested participant registers for the subject, and afterwards when the data of the subject changes, the subject informs all registered observers about it. The simplest form of observer pattern with interfaces is presented in Figure 3 using UML component diagram notation. The ISubject interface provides the methods for registration and deregistration, and the IUpdate interface provides the Update method, which gets called when the data in subject changes.

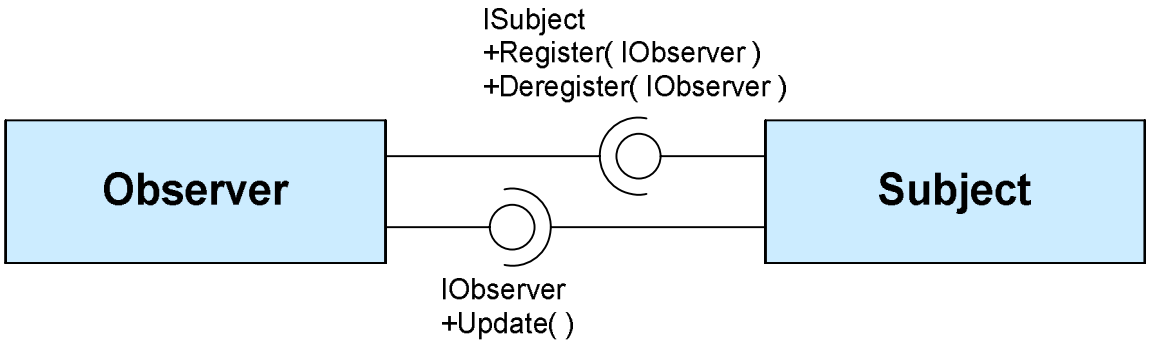

Figure 3 Observer-pattern

A downside to observer design pattern is the possibly large amount of unnecessary update calls. This can happen if the subject has a lot of observable data, but the observer is only interested in some specified slice of data. Without an additional mechanism to provide additional information about the changes to the observer, it may be costly for the observer to find out the exact data that changed.

3. SOFTWARE PROCESS SUPPORT

"An effective software development process is essential for economic and physical survival of society, a society whose dependence on computers increases daily." [Leh91]

In this chapter first an overview of software processes and software process support is given. After the overview general requirements and challenges for a process support system are discussed. Finally, the requirements specific for the process support tool implemented in this thesis are presented.

3.1. Overview of Software Processes

Having tools to support software creation is not a new phenomenon, but the increasing complexity of software and growing business requirements cause a still greater need for them. The higher demands and quality requirements for software also cause the need to improve the development process itself. The first step in improving the process is in taking into account the notion that software development is a complex process itself. A part of improving the process is having better tools and environments to support it. For the support tools to actually be useful in supporting the process, instead of unnecessarily constraining it, such quality attributes as flexibility and integration of new tools becomes vital.

A software process is a set of various kinds of activities used in developing software. A process model is an abstraction of such a process. Well known process models include the waterfall model and evolutionary (a.k.a. iterative) development. There also exists numerous other different process models, but the following essential activities are common to all of them: software specification, software design and implementation, software validation and software evolution [Som07].

Software specification is the activity of describing the requirements of the software. This includes the functional and non-functional requirements. Software design and implementation is the activity of planning and creating the actual software. Software validation is the activity of ensuring that the software meets the demands laid out in the specification. Software evolution is the activity of evolving the software according to the needs of the customer.

The concrete products of all the activities are called software artifacts. An artifact can be, for example, executables, code, or documentation. Documentation refers both to in-house documents such as design documents and project plans, as well as user manuals etc. documents delivered to the customer.

A more complex definition of a software process is given by Fuggetta [Fug00]: A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy and maintain a software product. From this definition Fuggetta derives that software processes benefit from the following concepts:

- Software development technology: technological support, i.e. tools, infrastructures, and environments.
- Software development methods and techniques: guidelines on how to use technology and accomplish software development activities.
- Organizational behavior. Software development is carried out by teams of people that have to be coordinated and managed.
- Marketing and economy. Software must address real customers' needs in specific market settings.

As examples of existing process models, the previously mentioned waterfall and evolutionary development models are given a brief overview in this section. The waterfall model defines a process, in which the basic process activities are done in phases in a specified order: requirements definition, design, implementation, integration, testing, and maintenance. Winston Royce has been generally seen as the original author of the waterfall model, but similar clearly phased models have been published as early as the beginning of the 1960s [Vli00]. In the most pure form of waterfall model, the phases are completed one after another in a completely sequential manner. However, this kind of inflexible development process has always been more like an idealized concept, than a widely preferred way of working. Royce already in his original publication criticized it and suggested various improvements to the model, to make it more usable in real world scenarios [Roy70].

Evolutionary development is based on the idea of starting from small prototypes and gradually building the working system towards the full customer needs. The benefit in this approach is that important issues can be found earlier and therefore it is easier and cheaper to react to them. Another benefit is the easier gathering of functional requirements for the final software product, as the customer can try out prototypes build on initial requirements and review the requirements using that experience. This method can also raise the level of customer satisfaction.

In conclusion, there exists several well defined process models according to different needs. However, software processes are complex entities and the requirements for the final software products can be completely distinct between different domains, customers, etc. This leads to the fact that the software processes can vary greatly among different organizations, projects, time (evolve), etc.

3.2. Software Process Support in General

The idea of supporting software processes in its basic form has been around since the development of first compilers. The idea has since then been evolving and nowadays processes can be supported in many different ways and levels. There are Computer-Aided Software Engineering (CASE) tools from specific tasks to multi-purpose environments. Examples of case tools include code generation tools, configuration management tools, UML design tools, debuggers, and tools for supporting the software process itself.

Fuggetta proposed a classification of CASE tools to three different categories: tools, workbenches, and environments [Fug93]. He defined a tool to mean a component that supports a specific task in a software process. Examples of these include compilers and textual editors. Fuggetta defined workbenches to mean applications that integrate several tools to support a specific software process activity. Examples include analysis and design workbenches and configuration management workbenches. Finally, he classified environments to mean CASE products that integrate a set of tools and workbenches to support an entire software process. CASE Environments can be subcategorized to several subclasses, including toolkits, language-centered, integrated, and process-centered environments. The concept of a process-centered environment is discussed in more detail in the following section.

Process support tools that offer support for the whole software process are also known as process support environments or process-centered software engineering environments (PSEE). These environments are used to create and run a software process model, sometimes defined with a process modeling language (PML). Process modeling languages are used to define the entities used in a process, including activities, artifacts, roles and tools. In addition to fore mentioned documentation, artifacts in this case include the guidance created for the process users for proper execution of the process. This guidance can be, for example, user manuals for the tools in the process. Roles in a software process can include, for example, process manager, tester, and designer. Benefits of process support environments can be various. For example, the environment can automate tedious routine tasks and guide to the use of good practices. In addition, the environment can help the user to find and use artifacts and tools that are related to the current tasks and to the current state of the process.

Sommerville [Som07] lists two main reasons limiting the improvements gained from the use of CASE tools. The first reason is that the software designing requires creative thought. CASE tools can automate routine tasks, but attempts to provide support for the design itself have not been successful. The second reason is that complex software engineering requires quite a lot of cooperation and interaction between team members. CASE tools have not been able to provide much support in that area.

3.3. Challenges of a Process Support System

Process support is in some ways comparable to normal software design. For example, the output artifacts of normal software design and implementation, i.e. the code, must not be too monolithic. The same applies to process support. If the process, or the process support environment, is too rigid and monolithic, then quite similar problems may arise, for instance, latent process requirements may cause more work than they should.

Aoyama [Aoy98] found that many PSEEs have too strict requirements on the execution of the process. Aoyama explains that they have found such constraints to cause inflexibility and loss of productivity, and they believe that their more people-oriented philosophy would lead to better results. Conradi et al. [Con02] make the notion that software process tools: “must adapt to the specific needs of the application; building an advanced tool for the wrong application is technological overkill”. In addition, the growing business requirements of, e.g., using less time and money for development and maintenance, lead to higher demands from the software development process in general. One of the key matters is greater flexibility of the process itself. Other requirements include better overall management of the process, and integration of new tools to the process. Fuggetta [Fug00] lists several key challenges in software process support including:

- Process modeling languages (PML) must be tolerant and allow for incomplete, informal and partial specification
- Process-centered software engineering environment (PSEE) must be non-intrusive. It must be possible to deploy them incrementally.
- PSEE must tolerate inconsistencies and deviations.
- PSEE must provide the software engineer with a clear state of the software development process (from many different viewpoints).

With these general challenges in mind, the next section discusses the requirements in more detail, and also introduces several requirement scenarios for a process support system.

3.4. Requirements for a Process Support System

The work presented in this thesis was done as a part of a research project in Software Systems Department in Tampere University of Technology. The research project presented two main requirements to the process support system described in this thesis. The main requirements were maintainability and flexibility. Some of the rationale for these requirements was presented in the previous section, for example, it was discussed that process support systems in general should be adaptable. In addition, especially in research environments it is important to be able to experiment with how various things

work with different configurations. This subsection discusses the rationale behind the two main requirements a little more profoundly.

When assessing the requirements for the target process support system, in the scope of this thesis, the point is to review the applicability of the agent based approach in implementing a process support system. Therefore the most weight is given to the requirements that are specific to the process support domain.

3.4.1. Rationale for the Requirements

The requirements for a software process system stem from some distinctive properties of process support systems. For example, there are different interest groups involved in the software process, and these groups are primarily interested in different kinds of information from different viewpoints. In addition, it is possible that some information in the process must not be available to all roles and groups involved in the process. For instance, an organization can have sub-contractors that simultaneously work for the competitors of the organization. In such cases it is important that the organization is able to hide the core competence parts of the process and reveal only the minimal needed information to the sub-contractors.

The information level in process support systems can be divided to two: the meta-level where the software process itself is designed, and the instantiation of the process. Most of the used tools and methods are specified at the meta-level. Some of the more common variances could be defined directly at the meta-level, for instance, it could be left to the developer to decide the specific tools used in some design activity. However, not all variances can be anticipated and therefore the instance level needs to be flexible enough to support dynamic deviations from the specified process.

3.4.2. More Specific Requirements for the Target Process Support System

In this subsection the primary requirements for the target process support system are presented briefly. It is essential that existing tools used by the developers can be integrated to the environment. It must to be possible to define the process used and the user must be able to see the state of the process and control it. The state of the process must be persistent and the artefacts produced and used by the process need to be saved. Because of several developers, the process needs to be synchronized among all of them. The inherent nature of software development is such that the process, tools, and environment may change for every project. Additionally, for performance, usability, etc. reasons, it must be possible to execute process activities and use tools both on local and remote computers.

To address the specific requirement of flexibility, a set of specific architecture requirements is used. They are not a complete requirement set, but they give a way to elaborate the general requirements. The flexibility requirements can be divided into several different branches. These include development time flexibility, configuration time flexibility, and runtime flexibility. More specifically, runtime flexibility can still be divided

to two distinct branches: the variance a normal user can achieve in the workflow, and the variance an administrator can achieve. To open up these requirements, at least one scenario is given for each in the following paragraph.

Important requirements for development time flexibility include that it must be possible to add new tools used by the developers to the workflow in reasonable time; and it must be possible to adapt the system to the chosen workflow, and not the other way around. Configuration time flexibility means, for example, that it must be possible to change the toolset used in a workstation easily. The variance a normal user can achieve in the workflow includes adapting the normal process to changing requirements easily. This can mean, for example, skipping a task that is not applicable to the current project anymore. It should be possible to make any such variation easily if not otherwise constrained. The administrator should be able to change things like the amount of information certain people or roles in a project can view, for example, if a sub-contractor is also using the same process support system.

3.5. Architectures of Existing Process Support Systems

Several PSEEs are reviewed and the commonalities in the architecture of those systems are discussed in a publication by Fuggetta in 1996 [Fug96]. This section summarizes the findings made in that publication.

Three types of components are described to be found in all of the considered PSEEs: a user interface facility, a process engine, and a repository. The user interface facility projects a view to the state of the process for the user, allows the user to control the process, and allows the user to view the results of the process activities. A process engine executes the process, invokes tools, and uses process artefacts. Repository is used to store the process data, including the process artefacts. A typical interaction between the components is that the tools and user interfaces interact with the process engine, and the process engine interacts with the repository. In addition, some tools may interact directly with the repository, but a more common approach is that the tools only use the file system directly.

In some of the PSEEs reviewed the user interface was distributed. This led to a typical client-server architecture, where the server constituted from the process engine and the repository, and the client from the user interface. One of the PSEEs also attempted to distribute the repository to achieve a more distributed functionality.

In conclusion, the architecture must support the integration of at least these three types of components. In addition, for reasons described in the previous section, it must be possible to distribute the integrated components in a reasonable way.

4. AN AGENT BASED ARCHITECTURE

In this chapter first the rationale behind the need for an agent based architecture is discussed. In addition, it is described how the specific process support system requirements have shaped the formation of a more general agent based architecture. After the rationale, the agent based architecture is presented. The rationale and the architecture have also been discussed in Peltonen et al. [Pel09] and Vartiala et al. [Var07].

The presented agent based architecture is not constrained to any single implementation style or platform. Therefore first a general architecture is presented and only in the later chapters the details of an example implementation are described.

4.1. Motivation for a General Agent Based Architecture

The main quality attributes for the process support system, i.e. flexibility and maintainability, are also valid for the more general agent based architecture presented in this thesis. More specifically, as the architecture is first of all an integration architecture, the flexibility requirements mean it must be possible to integrate various components together. Often these components are COTS-components that cannot be modified. In the case of a process support system the way these components interact can vary in multitude of ways. As all the possible ways these components interact cannot be predefined, the architecture should not unnecessarily constrain the developer in the ways the components can be used. The architecture should also support easy implementation of new use cases in how the existing components are used.

Maintainability in the case of the architecture means first of all the simplicity and understandability of the architecture, as a too complex architecture can lead to various maintainability problems. For example, Haikala et al. [Hai06] describe that even if a design solution is excellent in theory, in practice the solution can be too complex. For example, the solution can be too hard to explain to all people, or understanding the design concepts can simply require too much effort and time. This can lead to many problems, for instance, if the follow-up developers misunderstand the design concepts then the architecture becomes rapidly unusable [Hai06].

To answer these challenges an agent based approach was chosen. Agents enable the creation of a simple, loosely coupled and easy to understand architecture by making it possible to divide the architecture to agents and infrastructure in a beneficial way. Such a division makes the architecture more flexible and easy to extend. In addition, using the agent based approach allows relocating each business logic case to single place - an

agent. Having the business logic in one place makes it easy to maintain the existing business logic and to flexibly add new business logic functionality.

4.2. An Overview of the Approach

The general idea of the agent based architecture style is that there is an infrastructure offering services for agents, which use the infrastructure to move around and to achieve their goals. It is notable that typical agents are not very complex; on the contrary, most often they are simple task based agents with a predefined behaviour. Additionally, one agent should only be related to a single task for simplicity.

To make a clear distinction between the entities on different abstraction levels, the approach is presented in three meta-levels, where a higher level architecture defines the possible instances of lower level architectures. As seen in the vertical axis in Figure 4 the levels are from the most abstract to the most concrete: meta-architecture, system architecture and runtime architecture. The meta-architecture, i.e. the architecture meta-model, describes the entities that can be used to define new system architectures. Basi-

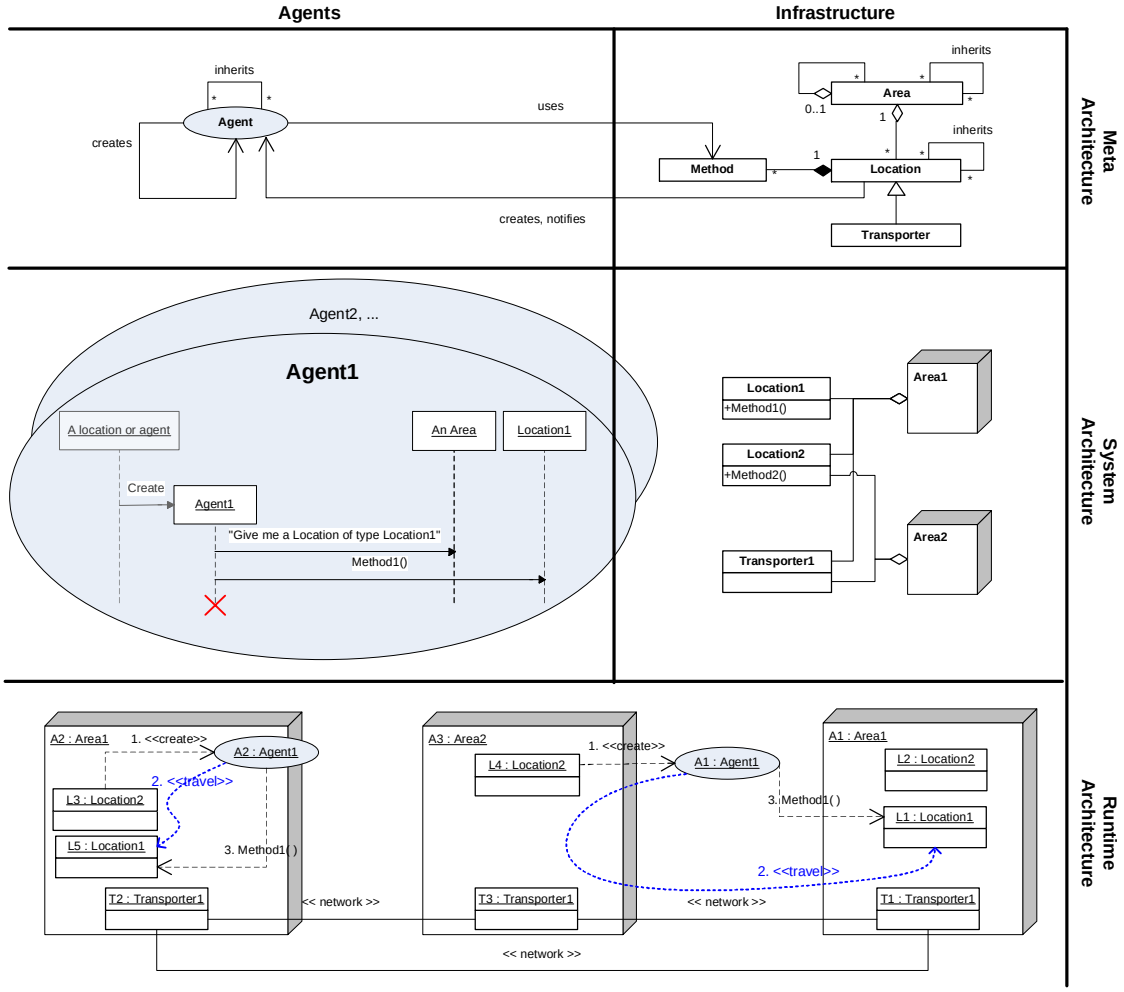

Figure 4 The three metalevels describing the agent based architecture model

cally, a meta-architecture is an architectural style defining a language for specifying possible architectures according to that style.

System architecture is the logical architecture definition of a concrete system and runtime architecture is a possible, physical, runtime instantiation of the system architecture. There is also fourth level, meta-meta level, which defines a language for specifying meta-architectures. In this case OMG Meta Object Facility (MOF) is used as such language [OMG02]. Besides that the architecture is divided vertically to meta-levels, it is also divided horizontally to infrastructure and agents as seen in Figure 4. That is, the business logic is separated from the underlying infrastructure.

The meta-architecture of the infrastructure, as shown in the upper right corner of Figure 4, consists of areas, locations, methods of locations and transporters. An area represents one group of locations typically located in one computer. Locations offer different kinds of services to agents through their methods and they can also create new agents when something needs to be done. Typical locations include user interfaces, as well as interfaces to databases and various other applications.

Transporters are special kind of locations connected to each other. They are used for transporting agents to remote areas. The architecture style allows three different forms of travelling: Agent tells the infrastructure 1) only the type of the location, 2) the type of the location and the type of the area or 3) the type of the location and the ID of the area. The locations, areas, etc. are meant to be built in a way that they do not know anything about the functionality provided by other entities in the infrastructure.

The agents, seen on the left side in Figure 4, use the functionality offered by the infrastructure to achieve their predefined tasks. More specifically, the agents move among different locations, possibly located in different areas, and use the methods of the locations to achieve tasks. The agents do not need to know anything about the runtime architecture, but they can rely on their knowledge of the description of the system architecture. More specifically, they typically only need to know directly the types of the locations they want to use. The only things that get transferred between areas are agents.

The architecture does not limit the amount or type of the above-mentioned entities in any way. On the contrary, one of the key points is that it should be made as easy as possible to expand any system using this architecture by adding new agents, locations, areas and transporters to it. This helps to achieve the needed flexibility, customizability, and incremental development requirements. For the same reason, the maintenance of the system is straightforward.

4.3. System and Runtime Architectures

System architecture is the description of the architecture of a concrete system. It is achieved by instantiating the meta-architecture in any way the architect desires. A possible example of system architecture can be seen in the middle part of the Figure 4. The example consists of two agents, two areas, two locations and a transporter, named according to their types. Notable in the example is that both areas have Transporter1 and

Location2, but Area1 has additionally Location1. A reason for this might be that Location1 requires some special resource or processing power not available in a normal workstation, thus a more efficient server is required to run Area1.

What cannot be seen from the figure is what kinds of connections are allowed by Transporter1. Generally, the type and number of possible connections depends entirely on what kind of transporters there are in an area. For example, Transporter1 could allow connecting to an unrestricted number of other transporters, or it could only allow one connection to a transporter of type Transporter1. In this case there can be an unrestricted number of connections.

Runtime architecture consists of all entities and their states of a system in one moment during runtime. It is possible to have an unlimited number of different runtime architectures using the same system architecture, because typically the amount of entities is not constrained in any way. An example of a possible runtime structure is seen in the bottom level of Figure 4. This runtime structure consists of three areas, and as defined in the system architecture, each area has an instance of Transporter1 and either one or two locations. All of the transporters are connected to each other over the network, and hence they form a kind of a peer-to-peer network in this case. The situation in the example, three areas and two agents, is not caused by any restrictions; an equally possible case would be a runtime situation with, say, tens of areas and hundreds of agents.

The dashed lines in the bottom level of Figure 4 show the behaviour of two different instances of Agent1. The leftmost dashed lines show the runtime behaviour of an agent of type Agent1 when invoked in Area1. First the Location2 wants something to be done; hence it creates an agent of type Agent1 and possibly gives some parameters to it. Then the agent starts the execution and comes to a situation where it needs to use Method1. Thus, the agent indicates to the infrastructure that it needs to use a location of type Location1. Since a location of that type is located in the same area, the agent is moved there. After the short travel the agent calls Method1 and decides that it has done everything it needed and thus the agent stops there.

The rightmost dashed lines show the behaviour of Agent1 when it is created in Area2. As a distinction from the previous example, there is no Location1 in the area where the agent is created. Thus, when the agent wants to use Method1 of Location1, the infrastructure transports it to an area, which has a location of type Location1, in this case to Area A1 is chosen. The second line is a composition of all the events that occur during that travel. After the traveling the agent uses Method1 of the location L1 and stops.

4.4. An Example: Observer-Pattern

The simplest complete system architecture to support observer pattern [Bus96] can be created with five entities in the system level as seen in Figure 5. The meta-level is not described anymore as it is same for all system architectures. On the infrastructure side

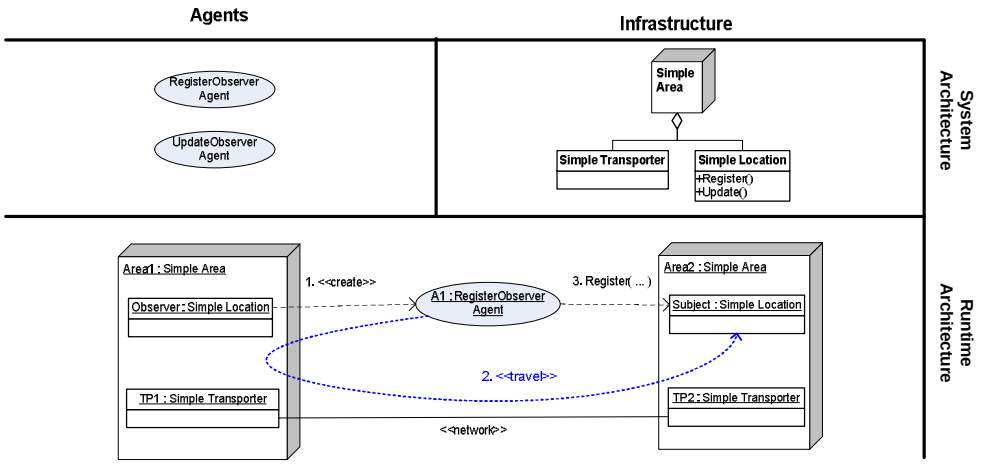

Figure 5 Using observer-pattern in the agent-based architecture

there is one area, Simple Area, which consists of two locations, Simple Transporter and Simple Location. Simple Location works as both the observer and the subject, and it offers methods Register and Update. To achieve the functionality needed in the pattern two agents are needed. RegisterObserverAgent-agent registers an observer to a subject and UpdateObserver-agent is then used to update the registered observer.

In the bottom level of Figure 5 there is the runtime architecture with two instances of Simple Area. The Simple Location in the leftmost area works as an observer and the Simple Location in the rightmost area works as a subject. The dashed lines in Figure 5 show the sequence of events during the lifetime of a RegisterObserver-agent. The sequence starts when the leftmost Simple Location wants to register itself to the Subject and creates an agent for this purpose. The needed parameters are also given to the agent at this point. These parameters include at least the type of the subject-location and the ID of Area2, because the agent needs to know exactly who to register and to whom. Additionally the initialization data could include, for example, the type of events that the observer is interested in. The second line is a composition of all the events that occur during the travel from the observer to the subject. Line 3 shows the actual registration of the Observer-location. After that the agent stops and is destroyed.

4.5. Agent Characteristics

The general idea is that an agent usually implements a single business logic case. The idea is that agents would be quite simple and there would not be much overhead confusing the developer, but instead letting her write the business case in a straightforward manner.

There is no direct support for agents to communicate with each other, but specified locations can be created to provide similar functionality. An agent can use other agents to achieve its goals, for example, by creating other agent and if needed, then possibly getting the other agents output through a location.

In some cases there can be a need for many similar agents, which could be categorized into group of agents. The architecture does not constrain the developers from creating such groups of agents. They could be created, for example, by creating a general base agent for all similar agents to specialize.

In conclusion, the framework makes it possible for an agent to achieve the properties listed in Table 1, but the framework does not require that the agents to support all of these. This also summarizes the design philosophy of the agent architecture well, as the intention has been to keep the architecture simple and easy to understand. In this case this means that, for instance, it is not required that the agent developers learn complex agent technologies and AI-concepts. As a downside, if some complex operations are needed, then additional support from the architecture could make implementation process easier for the developers

5. IMPLEMENTATION OF THE AGENT FRAMEWORK

In this thesis a prototype agent framework has been implemented to validate the approach presented in the previous chapter. In this chapter first the implementation of the agent framework is presented, and then the extensibility possibilities of applications using the framework are looked at.

5.1. The Infrastructure Supported by the Agent Framework

The framework implements all described entities in the meta-level (location, area, agent, and transporter) of the architecture and makes it possible to specialize system level architectures from it. The framework also implements several other helpful entities to make the implementation of a working system easier. There are also some implementation specific details not part of the architecture model itself. These details are described in the following paragraphs.

All locations in the infrastructure offer some basic functionality to agents. They allow the agents to travel to other locations and they can redirect an agent to a transporter if a wanted location is in another area. They also allow asking the current area and the type of the current location. The type of the location is important information, since the agents typically navigate in the infrastructure using them. Areas only know the types of their locations and have no other knowledge of them or other areas, i.e. areas are autonomous and running an area does not directly require the presence of any other areas. All areas have a type and an ID; these can also be used by agents to move among them. Each area also has at least one transporter.

Agents are transported by first serializing the state and data of an agent in a transporter, then creating a similar agent at another transporter in a remote location and deserializing the state and data for this new agent. One transporter can have multiple connections to other transporters. All transporters support the operation of asking all the areas currently connected to that transporter. An agent can be transported to any such connected area through the transporter.

There are several common features to the whole framework. For instance, the framework takes care of concurrency, network communication, and all other things that are not related to the business logic. By providing these common features, the framework allows the agents to focus on implementing the functional requirements of the system.

5.2. The General Characteristic of Agents in the Framework

An agent has current location, current state and a home area. The home area tells where an agent originates from, and where it should navigate if it wishes to come back from a remote location. Current state is used to determine what the agent has done, and what it should do next. There is no predefined state behaviour or other constraints for the states of the agents, but it is hard coded to them, i.e. it is left to the creator of an agent to use the agents any way she prefers.

Agents can create other agents and in some cases even interact with them, but they can only coordinate their movement according to locations and have no knowledge of other running agents unless a location provides this information. Agents cannot create themselves, but otherwise their lifespan is completely handled by themselves. Agents can duplicate themselves at will and in normal situations they are only destroyed from their own initiative.

Agents do not directly need to handle lower level things like concurrency in any way, but the framework takes care of them. Of course there can be many agents under execution at the same time, but agents should not have to care about this. Still, they might have to wait before the execution of any called method of any location. The order of the queued agents may also change in some cases; therefore it is not always guaranteed that a preceding agent can use a location before a later arrived agent. Also it is completely possible that when calling two non-related methods in the same location another agent comes and calls the same location in between the two calls. If the ordering of methods or something similar is a problem, then the location can offer this kind of a quality of service. For example, a location could give each agent a unique transaction id that would be used to handle a series of operations made by an agent as a transaction.

The whole execution path of an agent should typically not be considered a transaction since the framework does not currently offer any means to recover an agent, which is in a disconnected area or to detect the loss of an agent. Agents can of course try to offer quality of service, but it is usually easier to just try to notify the user about an error and then leave the rest to her.

5.3. Threading and Process Boundaries

Threading is mainly handled by the framework, only some concurrency related locking needs to be done in the implemented locations. In normal situations agents do not need to handle things related to concurrency in any way. Small exception to this rule is the usage of the main thread. For instance, some UI-operations can require that they are launched from the main UI-thread. In such situations the agents can indicate to the framework that they need to be executed in the main thread.

In this implementation a single area is a single executable, i.e. one process. In this implementation there was no need to go over process boundaries. However, there are at

least two ways to do it. There could be two separate areas started in a single computer, and the agents could normally transport between these areas. Possibly the other area could be a main area and the other area a sub area which is directly visible only to the main area. Another possible solution could be that a location would work as an adapter over the process boundary. Depending on the situation either of these can be the better solution.

5.4. Core Classes of the Framework

The system is implemented with C++. Figure 6 shows a more complete structure of the implemented system. ExecuteActivity agent has been added to the figure as an example agent. ExecuteActivity-agent will be described in more detail in Chapter 6. The figure shows the big picture of the system regarding class inheritance and dependencies. In addition, the helper classes for agent implementation, creation and threading can be seen in the figure. Note that *abstract classes* in the figure do not mean only interfaces, but the C++ version of abstract, i.e. a base class that cannot be instantiated by itself. The new classes in the figure are described in the following subsections.

SerializableI

SerializableI is a base class for all serializable classes in the system. It provides functionality for serialization and deserialization of the object.

AgentI

AgentI is a base class for all agents. All agents need to implement it. It provides many

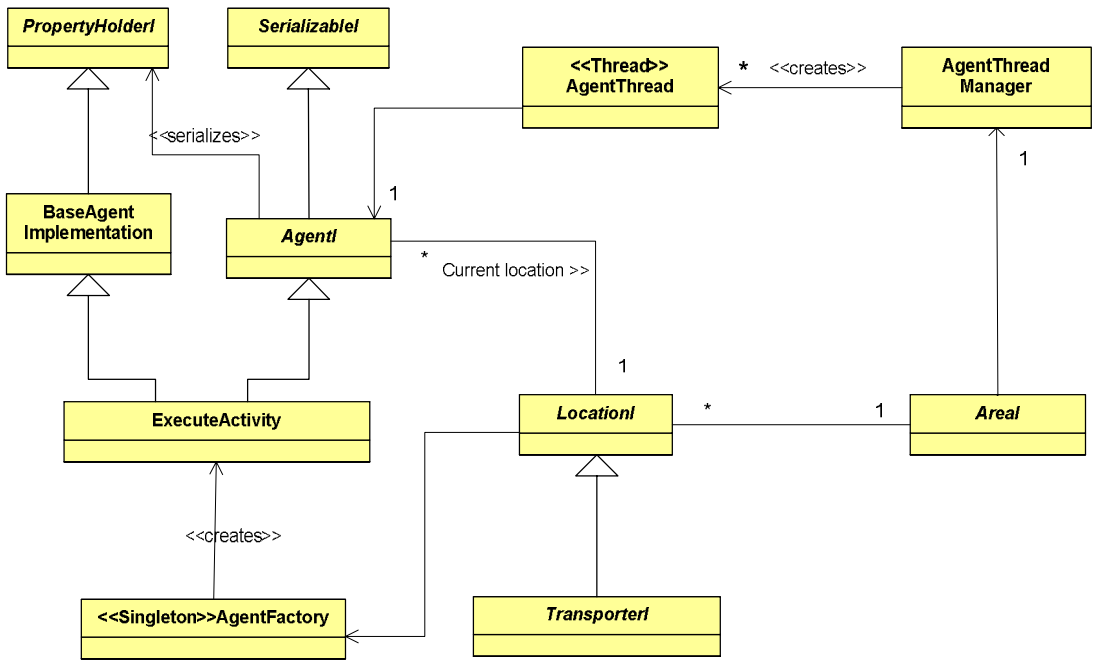

Figure 6 The core classes in the framework implementation

kinds of basic functionality, for example, for initialization and for automatic serialization and deserialization.

PropertyHolderI

PropertyHolderI is a base class for all objects with automatic properties. These properties are used in, for example, serialization. All agents must implement this class either by implementing class BaseAgentImplementation, or by implementing this class directly. The reason AgentI does not derive directly from this class is because of implementation reasons, more specifically because of the language constraints of template classes in C++.

BaseAgentImplementation

BaseAgentImplementation class has functionality that most of the agents have in common. For example, it has common properties used in most of the agents. It is not mandatory for an agent to implement this class if it does not need this functionality.

AgentFactory

AgentFactory is a singleton class responsible for agent creation. Each supported agent type is registered to the AgentFactory at the initialization of the program. The code of the agent factory does not need to be changed to support new agents. All locations, including transporters have access to AgentFactory and can use it to create new agents.

AgentThread

Each AgentThread is responsible for the execution of a single agent. AgentThread handles, for example, the stopping, pausing, and resuming of the execution of an agent according to the wishes of the agent.

AgentThreadManager

AgentThreadManager handles the creation and management of threads in the system. AgentThreadManager allocates free threads to new agents and handles the cleanup of old threads.

5.5. An Example Agent, RegisterObserver

In this section a possible implementation of the RegisterObserver-agent presented in the previous chapter in section 4.4 is given as an example. The RegisterObserver-agent is inherited from AgentI and BaseAgentImplementation. The agent implements a constructor and one method: ContinueExecution. The method ContinueExecution is overwritten from AgentI. ContinueExecution is the main method of all agents; it handles the actual business logic situated in the agent. Methods for initialization, starting, etc. are provided by the framework, i.e. the AgentI class. A simplified ContinueExecution method of the RegisterObserver-agent can be seen in Program 1. The method tries to be as

```
AgentResult RegisterObserverAgent::ContinueExecution( )
{
  if( currentAgentState_ == REGISTERING_TO_SUBJECT )
  {
    AgentResult currentTravelState = this->GoToLocation( subjectAreaID_,
      SIMPLE_LOCATION ); // travels to the right location until arrives
    if( currentTravelState == ARRIVED )
    {
      SimpleLocation* subject = dynamic_cast<SimpleLocation*>
        this->GetCurrentLocation( );
      subject->Register( this->GetHomeAreaID() );
      currentAgentState_ = READY;
      return DELETE_AGENT; // registration complete, agent can be deleted
    }
    return currentTravelState; // agent is still traveling to the subject
  }
}
```

Program 1 ContinueExecution method of RegisterObserver-agent

complete as it would be in the real application, but, for example, error handling has been left out for simplicity.

In addition to the code in ContinueExecution-method, the agent needs the ID of the area where subject location is. This ID is called subjectAreaID_ in the algorithm and is given to the agent at initialization. In the RegisterObserver-agent implementation it is defined that the variable subjectAreaID_ is a persistent variable, i.e. a variable that is always kept with the agent when moving among areas. For language specific reasons, in this implementation the definition of persistent variables is in the constructor of a specialized agent. These definitions are executed and converted into internal data structures once per each agent type. This is always done the first time an agent of a new type is created in the current execution of the application. Nothing else is done in the constructor of the ContinueExecution-agent.

The agent assumes that the subject area has a location of type Simple Location that provides the Register-method. All calls to this-pointer in the algorithm are calls to the AgentI-abstract parent class. After initialization, the ContinueExecution method of the agent is called by the framework automatically. In the method, the agent first travels to the right area and to the right location, and then it registers the home area to the subject. In this example, only the observer area ID is enough, as the subject assumes that the registered area also has a location of type Simple Location that wants to listen to the changes.

The ContinueExecution method can be called multiple times during the traveling to the right area and location. For example, after transporting from the observer area to the

subject area, the ContinueExecution-method is just called again after the framework has done the required initializations, including the deserialization of the state of the agent. The agent could decide to change its behavior at any time during the calls to the ContinueExecution, but here the agent just calls the GoToLocation-method until the framework indicates that the agent has arrived to its destination location.

The agent continues its execution until it signals to the framework that it wants to be destroyed. This happens by returning the DELETE_AGENT value from the ContinueExecution-method. The RegisterObserver-agent only had one execution state, REGISTERING_TO_SUBJECT, after which it was destroyed. If the agent would have had more states, then the functionality for all of them would still have been located in the ContinueExecution-method.

5.6. Expanding the System

One of the most important reasons for choosing the agent based system is the simplicity of adding new elements to the system. These elements include new agents, new methods to old locations, new locations, and in the case of a bigger design change also whole new types of areas can be added. All of the former changes can be done without losing the old functionality of the system. Existing methods in the location interfaces must only be changed with the utmost care, as all agents must be checked to be sure that the method can be changed. Another important feature of the system is the flexibility of the configuration. There is nothing that constrains the structure of the framework. All of the examples mentioned later in this section refer to the Observer-example given in section 4.4.

5.6.1. Adding New Methods to an Old Location

Adding a new method to an old location does not require any other changes than the actual implementation of the method and adding it to the interface of the location in question. However, it does not benefit anything before an agent uses it. For example, adding a SetValue-method to the Simple Location in the fore mentioned Observer-example would not change anything else.

5.6.2. Adding a New Agent

Adding a new type of an agent does not change any interface in the system. The only thing that needs to be done is to create it and register it to AgentFactory. Of course the actual agent instance also needs to be created somewhere.

For example, a new ChangeValue-agent that would go to a given area and use the previously added SetValue-method of the Simple Location could be created. Adding such an agent to the Observer-environment would require the following things. The agent must be inherited at least from the AgentI abstract class. The functionality of the SetValue-agent itself must be implemented, including the registration of the agent to the AgentFactory. To be useful, some location needs to create the agent. For instance, Sim-

ple Location could create the agent and at initialization give the agent a target area ID where the agent would transport to use the Simple Location.

5.6.3. Adding a New Location

To add a new location to an area an ID for the location needs to be created so that the agents can navigate to it, and then the location must be registered to the wanted areas, i.e. let them know the id of the location and the ids of the interfaces that it offers. If the locations increase a lot in the future then the possibility of creating them in a factory similarly to the agents should be considered.

For example, adding a Calculate Average Location to the Observer-example would mean the following things: a new location inheriting from the LocationI would need to implemented, an ID would be created for the new location, and the creation of the location would be added to the initialization routine of the Simple Area. At this point any agents that know the ID of the location could transport to it and start using the location.

5.6.4. Adding a New Area

The most important aspect of adding new areas is that the structure of how areas are connected to each other is not limited in any way. Again, adding such new functionality to the system does not influence the old functionality in any way. Nevertheless, adding a new area is a more complex operation than the other adding operations. For example, it has no locations or transporters until it is properly configured.

As an example of adding a new area, a Calculator Area could be added to the Observer-example. The Calculator Area could be configured to consist of the locations Calculate Average Location and Simple Transporter. No changes are needed to the existing areas, but the Calculator Area can immediately connect to the existing areas. There are no limitations why the existing agents could not transport to the new Calculator-area, but in the implementation done in this thesis there were no agents that would have dynamically used new types of areas. Therefore, in practice, for the new area to be useful, a new agent would need to be created that would transport to the Calculator Area and, for example, use the Calculate Average Location.

5.6.5. Changing the Topology of the System

Topology of the system in this thesis means how the areas are logically connected to each other. The topology of the system is controlled by the types and configurations of the transporters. For instance, if all transporters in a system could be connected to any other transporter, then the topology during runtime could be anything. Another example of a topology is a client-server topology, i.e. in the system there would be two types of transporters: client transporters that can only connect to a server transporter, and server transporters that only wait for connections from a client transporter.

There are no constraints regarding changing the transporter configuration, i.e. if needed, the transporter configuration could be changed runtime. This could be done, for instance, according to some information received from an agent. If the transporter types are flexible enough, then there is no restriction on the creation of an arbitrary topology.

In the case of insufficient support from transporters, then a new transporter type could be added to the system. This kind of change does not need to cause any changes to other parts of the system. Especially it would not cause a need to change the existing agent implementations, as long as the remote locations needed by the agents would still be achievable.

6. CASE STUDY: PROST PROCESS SUPPORT TOOL

The agent based architecture style was used to implement a process support environment. The environment is used to execute a software development process, where there are several tools and developers.

6.1. Decisions Regarding the Environment

Several architectural decisions regarding the environment were made. These include that the process is defined as a Visiome script [Pel00], which is run in a Visiome Engine. On top of Visiome Engine runs a model processing platform called xUMLi [Air02, Pel04]. Both of them will be part of the architecture and existing modeling tools (like Rational Rose) are integrated through xUMLi. The existing tools could of course be also integrated directly to the architecture, but since there is an existing implementation, which fulfils the requirements for the system, it was deemed unnecessary.

A frontend is needed for following and controlling the state of the process. It was decided that the persistency is handled by saving the state of the process to a database and the artefacts to a version control system. A process backend is used to make it simple to synchronize the process among different frontends and to allow remote processing at the backend.

6.2. Existing and Available Components

The system is used to execute a process defined in a Visiome Script. A process mainly consists of activities, but it also defines the order, in which the activities should be executed. These activities are COM-components, therefore they can, in general, be transferred to remote computers and executed in there. An activity may require user interaction and communication with Rational Rose when under execution. The execution of activities is handled using a specialized version of Visiome Engine over COM. The size of the output of an activity is not constrained in any way; therefore it must be prepared that the size is usually many megabytes. Visiome Engine also keeps track of the current state of the process. Under these circumstances multiple server/client configurations were considered. Three of these configurations were considered more thoroughly:

1. A system with only one Visiome Engine, located at the server.
2. A system with a VENG located at each client, no VENG at the server.
3. A system with a VENG at each computer, a master VENG at the server.

The first one was discarded because of the following problem: What to do with the activities requiring user interaction when they can only be ran from server. The second one was discarded mainly because it would be harder to keep the states of the Visiome Engines in synchronization. A specific problem in this configuration is the execution of fully automatic activities. Using a master engine at the server means a simpler approach as the server always has all current data about the process. In this approach the server can execute all automatic activities. The last approach also makes it simple to take backups and version snapshots of the whole state of the process.

The existing version of the Visiome Engine did not support dynamically changing the process in run time. This was not consistent with the need for greater flexibility. The problem was solved by making it possible to add new activities, which are not added to the Visiome Engine, during the process. The dynamically added new activities are handled by a layer on top of the Visiome Engine.

An example of a software process is presented in Figure 7. The activities, roles, artefacts and guidance parts of the figure are marked with light blue text. The process is read from left to right, the icon on the right top of an activity indicating the status of the activity. Even though not directly seen from the figure, activities are hierarchical items, which can consist of many sub activities. For example, Component Specification is not a single task, but a high-level task consisting of many subtasks.

6.3. An Overview and the System Architecture of the Example System

The prototype framework was used to implement the example system, i.e. all the used locations, areas, transporters and agents were inherited from their corresponding base classes. These inherited entities can be seen in the upper part of Figure 8. The current instantiation of the architecture can basically be seen as a kind of client-server architecture. The system infrastructure consists of two different kinds of areas, Backend and

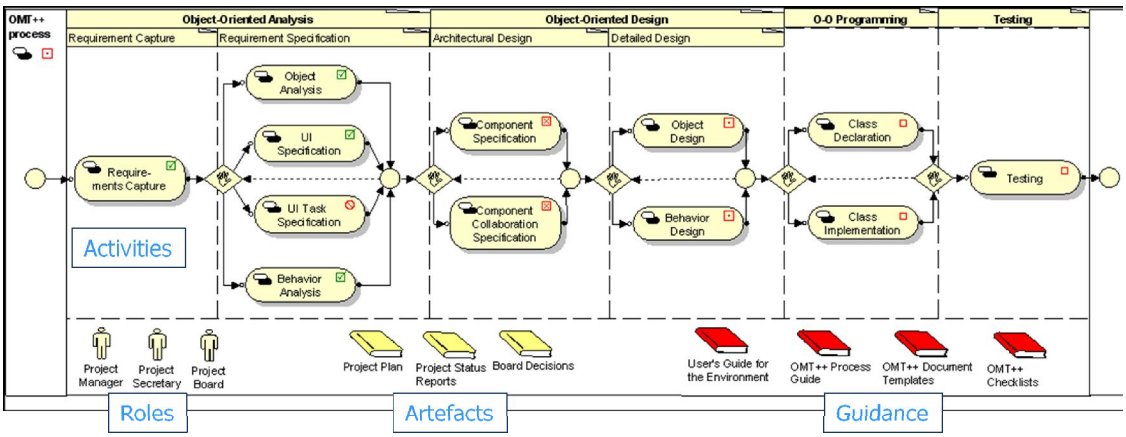

Figure 7 OMT++ process in a Visiome Script style.

Frontend. Both of these have their own transporter. The frontend transporter can only connect to one backend transporter at a time, and the backend transporter can only receive connections from frontend transporters. To add support for multiple backend areas either the backend transporter should offer functionality to connect to a backend transporter or a frontend transporter would have to be added to the backend area.

Common locations for both of the areas are Database and VersionController. Database-location offers an interface to the shared database of the system and VersionController-location offers an interface to use the shared version control of the system. FrontendEngine and UI are only located at Frontend area. FrontendEngine handles the communication to a Visiome Engine at the Frontend area. UI is the main user interface. Project-Handler is only located at Backend area. It manages the relations between users and projects and creates new Visiome engines. There are several different kinds of agents in the architecture; these include a StartProject-agent and an ExecuteActivity-agent. Most of the agents in the architecture are typically started by a software developer who uses the UI in a frontend.

6.4. An Example Run-Time Architecture and Experiences

Example runtime architecture with one Frontend area and one Backend area is presented in the bottom of Figure 8. The dashed lines show the sequence of events during the lifetime of an ExecuteActivity-agent. ExecuteActivity-agent is one of the most complicated agents in the implemented system. ExecuteActivity-agent executes an activity at the area it was created in. The travelling between the areas and locations has been omitted for simplicity.

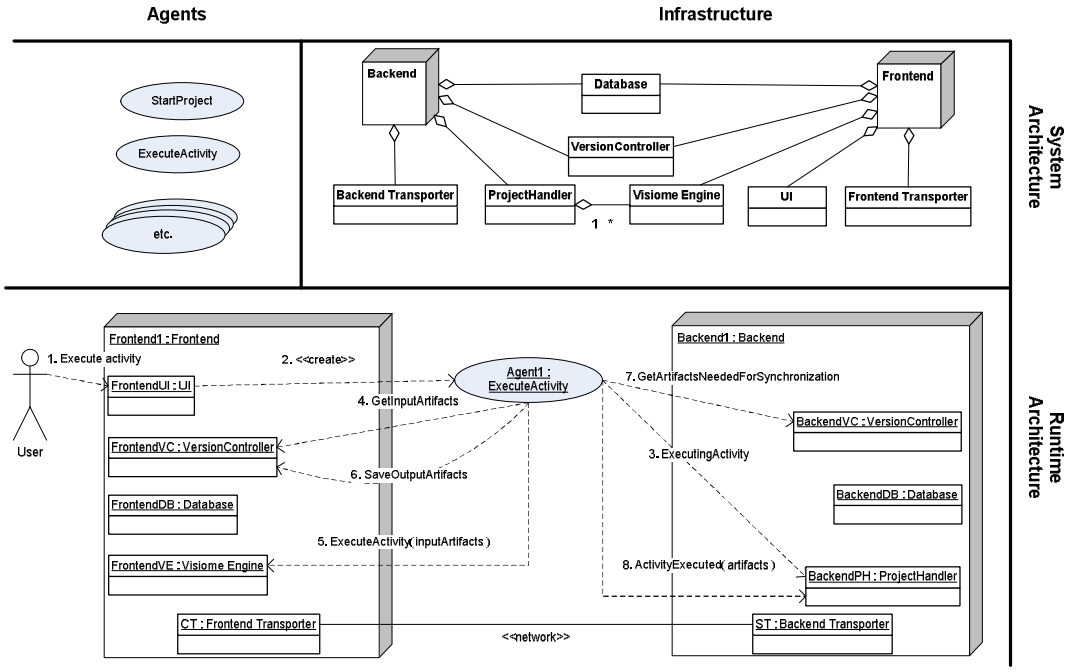

Figure 8 Specialized Architecture and the sequence of an ExecuteActivity-agent

The sequence starts when a user implies her wish to execute an activity. Then an agent is created, and it uses the locations in the order shown by the numbers. First it must travel to the backend and use ProjectHandler to lock the activity so that no other user can execute it at the same time. At this point the agent fetches the needed input files of the activity from the VersionController and starts to execute the activity at the Visiome Engine of the frontend. The activity itself can be of several different types, including an automatic activity with no user intervention or an interactive activity that requires interaction during the execution. After the execution the output is saved to VersionController at the frontend and synchronization is done at the backend using the outputs.

The framework and the complete system were implemented quite painlessly and successfully in reasonable time; therefore the case study can be considered a success. In addition, if the user interface is left out of the row count of the code, then the implemented system architecture has only a little more code lines than the implemented meta-architecture, i.e. the framework.

The division to the framework and to the system itself was quite viable and the framework implements several functionalities in their entirety. These include the transporting of the agents; including the moving over network; handling of the concurrency and general structures for managing locations and agents. Additionally there was only a minimal need to put non-requirements related things in the implementation of the system architecture. In the example system the methods of the locations are individual in the sense that there is no session between locations and agents using them, i.e. the locations do not provide methods, which require that a specific agent calls them one after the other.

6.5. User Interface

Main user interface of the system is located at the UI location. The user interface in an example situation can be seen in Figure 9. The idea behind the small UI is that if the user feels she needs, for example, guidance during the performing of an activity, she can keep the Prost UI and the current working tool both visible at the same time. The parts of the UI from left to right are: topic bar, menu bar, activity tree, HTML item selector, and HTML browser. The menu bar can be used to open different kinds of menus and

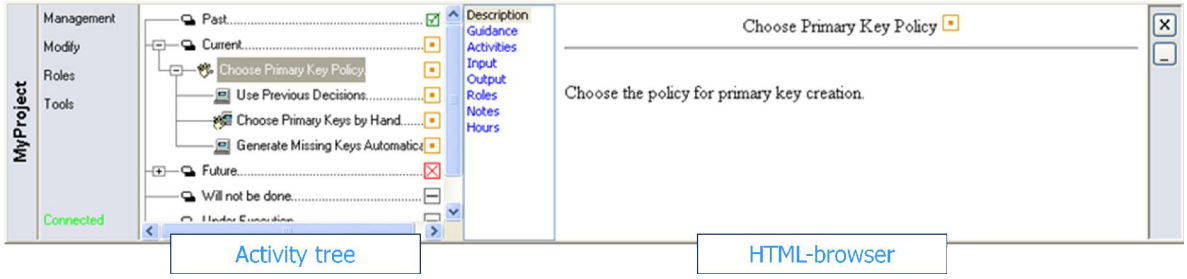

Figure 9 Prost Main User Interface

dialogs, for example, to add new activities, and to control the roles of a user.

The symbols on the left side of activity tree indicate the type of the activity. For example, the currently selected activity has a hand symbol on it, meaning that the activity is a manual activity. Other types of activities include fully automatic, automatic, and computer assisted. The activity types are explained more fully in section 6.7. The symbols on the right side of the activity tree indicate the status of the activities. Possible statuses include past, current, future, will not be done, and under execution. The HTML item selector is used to control what information is shown about the current activity in the HTML-browser. These include, for example, the guidance for performing the currently selected activity.

6.6. Specialized Agents

This section lists the most common agents implemented and used in the project. LoginUser agent is used to log a user in to the system. LoginProject agent is used to log an already logged in user to a project. StartProject agent is used to handle all tasks related to the starting of a project. ExecuteActivity is used to execute a single activity. ActivityStateChange is used to change the state of activities without actually executing them. AddActivity is used to create new activities to an existing process instance already under execution. ConnectionObserver agent is used to listen and deliver the connection status of the different areas to interested parties. ActivityTreeChange agent is used to notify interested parties about changes in the activity tree, for example, if another user executed an activity in the same project. In addition, there are agents, for example, for handling version controlling, i.e., for saving and getting files from version control.

As an example of agent states, the possible states of ExecuteActivity agent are shown in Figure 10. Even though not marked to the figure, Error and Cancel states can be reached from any state; the connections to them have just been omitted for clarity reasons. The needed current area of the agent in that state is marked with the dashed rectangle. The agent in this case starts from Frontend area, but there is no reason why the agent could not be started from any other area. There just has been no situation in the current implementation where this kind of functionality would have been needed. The cancel state has been added to the figure for design reasons; there is currently no implementation for canceling the execution of an agent. In practice, the user can achieve the cancel functionality by just changing the state of the executed activity back to the previous state.

6.7. Use of the System

A general usage pattern of the system is that a user starts the program, logs in to a project. The default setting is that the user can now see all the activities assigned to her roles in the activity tree. The user now sees the current status of her work, and can now decide what activity she should currently work on. The user selects the wanted activity,

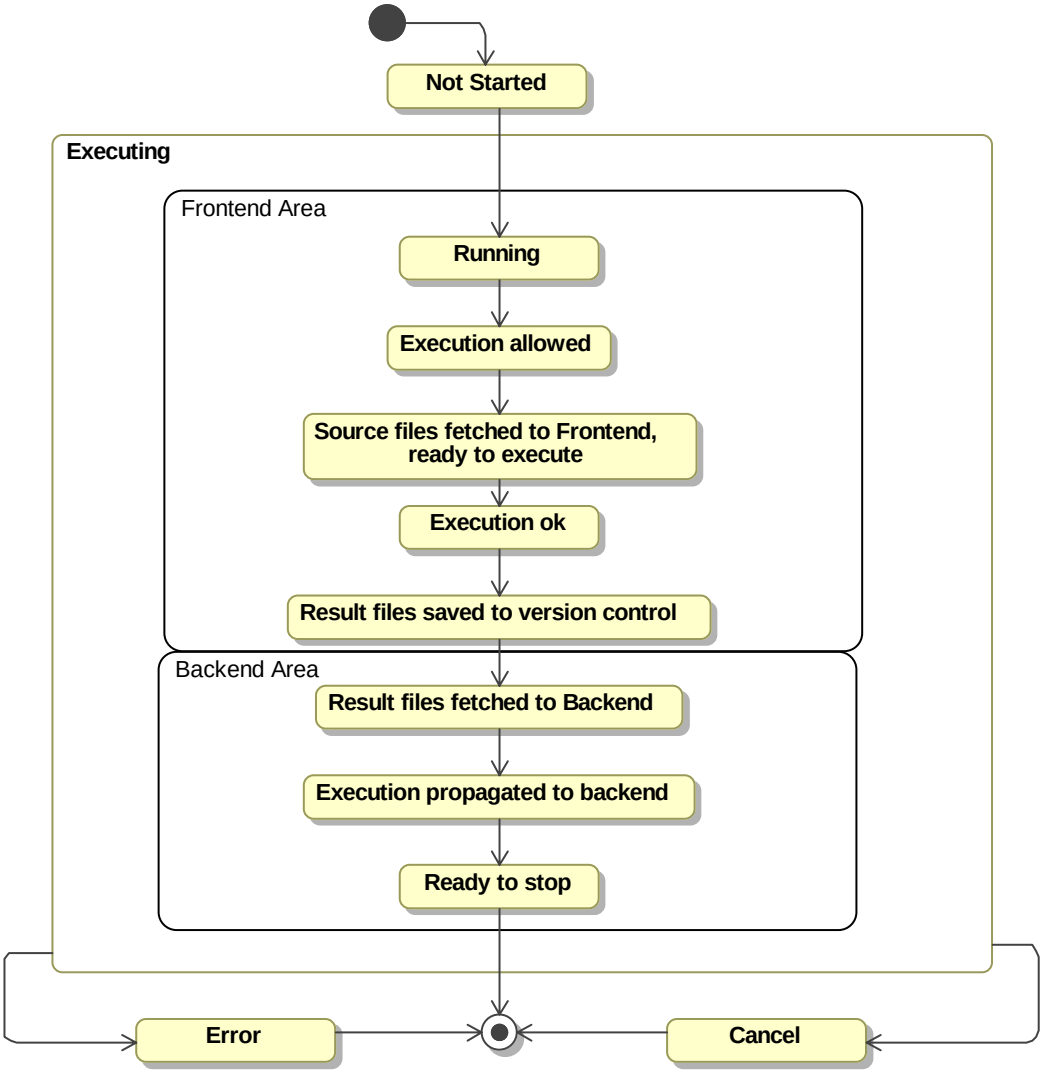

Figure 10 Possible states of the ExecuteActivity-agent.

possibly reads the related guidance in the HTML-browser, and then starts to execute the activity from the activity tree. In the case of a manual activity, this just means that the user indicates that the activity has been done, and the artefacts created by the user, for example, a class diagram, are saved to the version control.

In the case of computer assisted activities, the user is guided through the activity, possibly inside the current tool. For example, the user could be asked to select the classes needed as a source of transformation. In case of an automatic activity, the user just verifies that it is acceptable to execute an activity and after that the computer does all the related processing. Fully automatic activities do not require user intervention, but are automatically executed on the server when possible. The state and output of the fully automatic activities can be viewed by users from the activity tree similarly to all other activities.

Even though the system guides to work on the activities that are in the current state, there are no general restrictions to keep the user from reviewing or working with pre-

vious or future activities. However, there are some activities that require a specific input. For example, the transformation of a sequence diagram to a class diagram requires the sequence diagram as in input. Activities requiring such an input cannot be executed before their input is available.

In addition, executing an already executed activity with a specified output again may cause some unwanted consequences. For example, it is currently not possible to have different versions of the same process instance running concurrently. The default functionality in case of activities that have been executed multiple times is to use the latest versions of the inputs and outputs of all activities.

6.8. Expanding the System

As an example of useful functionality, which could be added to the system, is a way to add tasks to a running project from other tools. For example, activities from a project management tool could be imported to the system. This could be achieved by adding a new location and an agent to Frontend area. The new location would prove the core functionality to import tasks from that specific tool. The new agent would provide the information about what to import, where, and other information that needs to be decided before importing. In addition, the agent would be used to trigger the functionality when needed. The location would then import the needed information from the wanted tool and the agent would then get the information from it, possibly transform it to the right form and then add the activities to the process using the proper locations. In the case of missing or invalid information, for example, an activity with no roles assigned, the agent could notify the user about the situation and possibly ask how to react to the situation.

Scaling the system up in new instances of existing area types is a normal situation for the architecture. This kind of change does not require any changes to the existing architecture.

6.9. Implementation Techniques

The system is implemented with C++. The UI of the system is located in the Frontend area and is implemented using wxWidgets [Wxw09]. The browser in UI is implemented using Gecko [Gec08]. Gecko is an open source layout engine used in many applications including the Mozilla Firefox web browser. Frontend and Backend communicate with Visiome Engine through COM [Mic10]. Frontend and Backend communicate over network using the socket library in wxWidgets. The communication between Frontend and Backend consists mostly of agents. Frontend and Backend communicate with the Subversion server using a Subversion client and with the database using ODBC. The physical architecture of the system and the communication methods of the components can be seen from Figure 11. Backend, Database and Subversion can all be located on differ-

ent servers, but it is recommended that they remain on the same server as this helps maintenance and allows the server to perform faster.

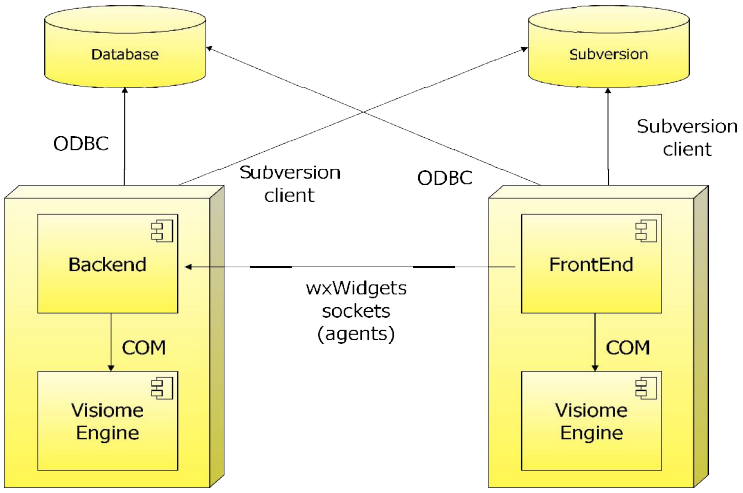

Figure 11 Physical architecture of the system

Backend only has a basic UI for observing the status of the backend. Backend only stores temporary data not directly related to the performing of the process, for example, the clients currently connected to it. All permanent data related to the process is saved to the database. All files are saved to the Subversion. Concurrency in file handling is automatically handled by Subversion. Concurrency in internal execution is handled by using specific libraries of wxWidgets.

6.10. Implementation Classes

The classes specialized from the framework are presented in Figure 12. The connections between the classes have been left out for simplicity. For example, several locations use many agents and showing all of these connections would make the figure unreadable.

Some of the classes in the figure are quite simple, including most of the agents. On the other hand, for instance, the UI location consists of tens of self implemented classes and brings with it the whole Gecko [Gec08] engine.

6.11. About Agent Implementations

All implemented agents are quite small in code size, most of them varying between 100 and 200 lines of code. The largest agent is ExecuteActivity agent with little over 300 lines. Most of the agent code is very straightforward and the sequence of the business case can be easily seen from the code. The functionality regarding agent transfer was implemented in a way that the agent implementer is not required to code anything extra for situations when an agent transports to another area. All that is required is that the agents define a default set of their state that they want to always carry with them. This

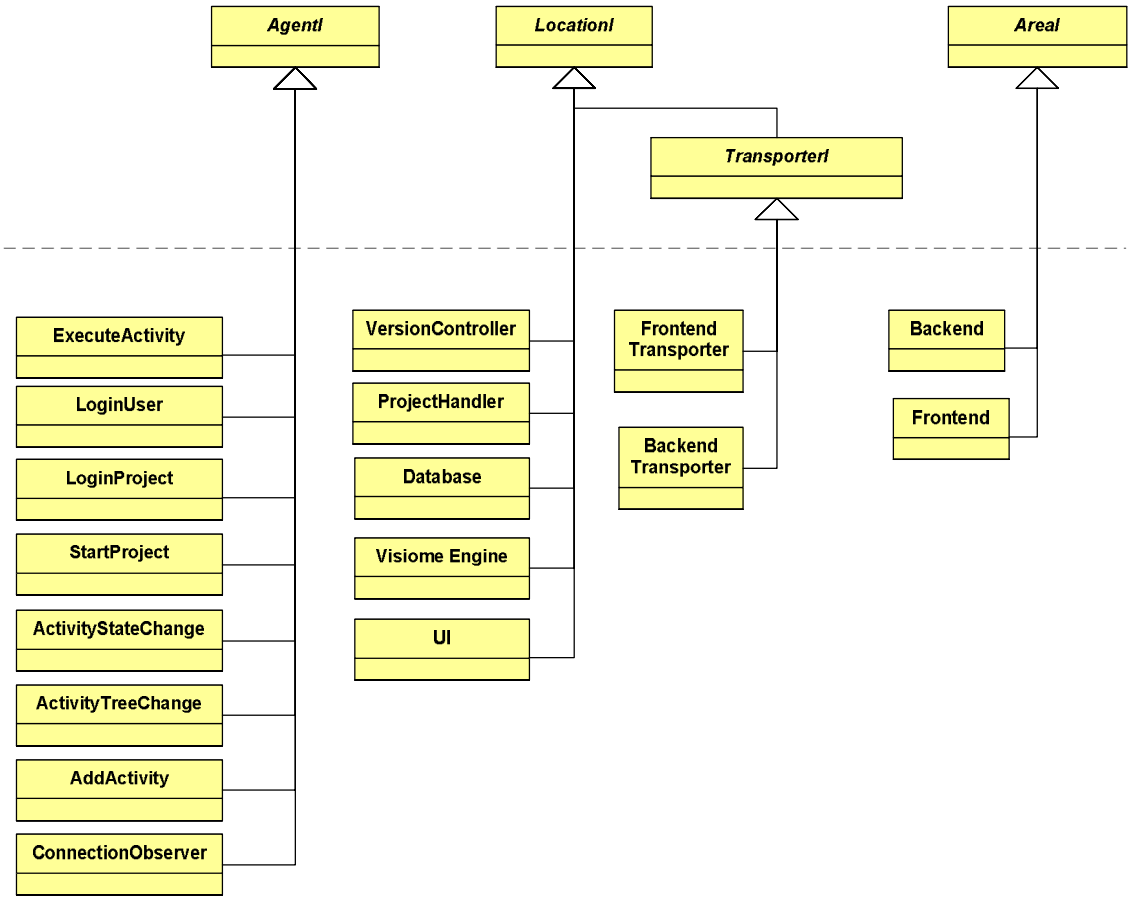

Figure 12 Specialized classes in the case study implementation

set is automatically serialized and deserialized when the agent transports to a different area.

A minor downside with current agent implementation is that adding a new agent type is not completely dynamic, but the parts of the system that use the new agent type need to be recompiled after adding the code of the new agent. This was not a problem in the current system, as there was no real need to add agent types dynamically. This would be a problem, if, for example, it would be wanted that the users could create new agents to do specific tasks for them. One way to address this challenge would be to make it possible to specify some agent functionality with Python or some other interpreted language.

6.12. Error and Exception Handling

All agents handle the errors occurred during their execution independently. More specifically, this means that the framework does not force, nor guide, the agent to handle errors in any specific way. In the Prost system it was decided that the implemented system would offer two kinds of procedures for error handling in agents. There are general logging capabilities that can be used by any agent, and then the UI location offers the pos-

sibility to display errors and warnings to the user. In addition, the agents can implement their own error handling procedures for specific error cases.

One important matter to consider is the behavior of an agent when an area is prematurely disconnected. For example, the previously discussed ExecuteActivity agent at the time it has reached the state *execution allowed* in Backend Area can be considered. Now the state of the activity has already been changed to *under execution* and the next thing the agent would do is to go back to home area and start the execution, but if the home area has been disconnected, then what should it do? Obviously the first requirement is that it must not crash the system or anything like that. There are several possibilities it could try to do. It could try to cancel the reservation of the activity so that somebody else could do it, i.e. change the state of the activity back to, for example, *current*. It could wait for the area to connect again or it could simply die out without doing anything. The latter is currently used. This simple behavior can be used because all operations of the highest abstraction level are designed so that they do not unnecessarily constrain the system and can be redone if necessary. In the case of the ExecuteActivity the user should first (after reconnecting) change the state of the activity to something else than under execution (this unlocks all reservation made to the activity) and then try to execute it again.

7. EVALUATION

The most important requirements for the system can be summarized into two: the overall requirement of flexibility, and the maintainability of the system. A big part of the maintainability and flexibility requirements is that it must be possible to incrementally develop the system. For example, it must be possible to integrate new tools easily to the existing system.

The requirement of maintainability of the system was addressed by transferring the business logic to agents, more specifically a single business case to a single agent. The flexibility requirement was tackled by making it possible to easily add new areas, locations and agents to the system. In this chapter, first the benefits and drawbacks of this work are considered, and then related work is discussed.

7.1. Benefits

The transferring of business logic to agents worked out well. The agents are quite simple and easy to understand and implement. When a certain business logic case is required the responsibility of the business case is given to an agent, instead of implementing it on the spot or sending a message to somebody who may be listening. The agent can then choose the proper methods to execute the task according to the current situation. The code that uses agents is also in some ways simplified, as after delegating the task to an agent, no regard for, e.g., error situations needs to be given any more. The overall division into agents and locations made the system simpler to work with and understand, creating a kind of a layered architecture automatically, the locations being the lower layer and providing the core and resource related functionality, and the agents using them to offer the higher level services.

The addition of a new agent type proved to be a simple procedure and did not clutter the code base as all the functionality was constrained to the agent. Moreover, maintaining the business logic code was easy, as all the related code was easy to find from that single agent. Adding new location turned out to be a little more complex operation than adding a new agent, but still relatively easy. Incremental development, for example, integrating new tools to the system, is supported by the fact that new location types can be added to the areas independently and without causing changes to old functionality. The customizability of areas, i.e. to use existing types of agents and locations, also allows the reuse of architectural patterns like the observer pattern with relative ease.

In conclusion, the easiness of adding new types of any of the main entities to the system and the clear division of the responsibilities to the agents, locations, transporters and areas, all helped to achieve the requirements of flexibility, maintainability and cus-

tomizability. With some other environment or in some other domain than the process support domain, there may be some constraints that hinder these required quality attributes, but from the point of view of the case study, the architecture proved to fill all of these requirements satisfactorily.

7.2. Drawbacks

There are some drawbacks in the concept of keeping only the higher level business logic in agents and having the locations to implement the lower level functionality. In essence, this means that the agents become directly dependent on the interfaces of the locations. Therefore the ability to change the interfaces of the locations may become the bottleneck of maintenance. This can become a problem if the amount of agents becomes too great, especially if a large proportion of the agents use the same methods. The effects of this drawback can be tried to minimize by keeping the locations well defined, by composing the agents to use other agents, and by giving the location design an extra effort to avoid too frequent refactoring of the interfaces.

Agent and location versioning has mostly been left out of the scope of this thesis. Having two versions of an agent or a location in different computers in the system simultaneously could cause some serious errors if that kind of a situation has not been prepared for. For example, an agent could use some information, which exists in both versions, but whose meaning has been changed. This would cause the agent to execute without errors, but the result of the execution would be something completely wrong. These kinds of situations would be very hard to detect. The current system was used in a way that all the computers connected to the system had the latest version running. In the case of more users and a more distributed working environment this could not be guaranteed. There are several ways to address this problem. Version numbering could be an automatic and immutable part of the agent state and the transporters would then check the version number correctness at some point before the agent is given execution time on the new area. In centrally maintained agent framework environments it would also be possible to have a top level version numbering on the areas. Then the version number of any connecting areas could immediately be checked and notified if they need to update to a newer version.

The presented architecture may not be proper for low level hardware with real time performance needs as the multithreading and agent serialization may cause too much of an overhead in such environments. On the other hand, a more reasonable comparison target for the presented architecture could be, for example, integration systems with constant transformation or interpreting of large XML-files. In such cases the architecture should fare reasonably well, because the agents use direct method calls to locations instead of, for example, sending XML-messages.

In the current implementation, all areas are compiled into their own programs. This includes the needed locations, agents, etc. as there is no dynamic addition of any new types in the current implementation. Therefore when an agent code changes, all the area

types that need the agent need to be recompiled with the new agent code, and all the areas need to be updated to the newer version, even though that specific part of the code would never be needed in that area. For example, if a minor change is done to the way an agent uses a single location in a very specific location, then in principle all the areas needing that agent still should be updated for version consistency reasons, even though the changed code would not be used in those areas. On the other hand, a system with no dynamic addition of new agent types is also inherently more secure, as there is no risk of any security flaws that could lead to unwanted users having access to add new dangerous agents to the system.

7.3. Comparison to Pitfalls

In this section the design and implementation of the agent architecture is reviewed on the basis of the pitfalls described in section 2.3.3.

Trying to do everything yourself with agent-techniques

In this thesis various related technologies and COTS-components were used in the development of the system. These include wxWidgets and database systems. For example, wxWidgets sockets were used in transporting agents to other areas.

Deciding you want your own agent architecture

A proprietary agent architecture was obviously built in this work. However, we believe we have managed to avoid the most serious problems regarding this pitfall. The actual time spent in developing the architecture framework was relatively small overall. The relative ease in development effort was achieved by keeping the framework simple and well-defined. The end result satisfied the requirements for this work, but for some other systems the property of having no direct communication between agents may prove to be a too great drawback.

Thinking your architecture is generic

We do not consider the architecture presented in this thesis to suit to all distributed systems and domains. However, we do consider the concept of the agent based architecture to suit relatively well to the application integration domain, and especially to process support systems and tool integration systems. This is supported by the generality of the main concepts of the architecture and the independence to any programming languages. We do admit that the current C++ implementation of the agent framework is only a case study and a lot of useful features could be added to it. As such the current C++ framework may not be generic enough to suit very diverse needs even in the process support area.

Having the agents use too much AI

Extensive AI techniques were not used in the framework or in the implemented case study. However, agent developers are not constrained from using any techniques they wish to in implementing specialized agents.

Having agents with no intelligence

This pitfall could be rephrased to calling entities with no agent-like behavior as agents. We consider this not to be the case with the agents in this thesis. Depending on the classifications, the agents are at least mobile, autonomous and goal-oriented. In addition, the architecture framework does not constrain the agents from having, for example, learning behavior.

Seeing agents everywhere, having too many or too few agents

In general, an agent in this thesis implements a single business case, for example, executing an activity. We feel this division was proper for the case study, and see no apparent reason why it would not work on other similar systems. We feel this kind of division made the architecture of the process support system easy to understand and clear to work with.

Spending all time implementing the infrastructure

See the answer to the pitfall *deciding you want your own agent architecture*.

Having an anarchic system

The complexity of the system was reduced by not providing any means of direct communication between the agents, but making them communicate with the locations instead. This reduces the amount of possible communication channels considerably, as there are usually a lot more agents in a system than there are locations.

Thinking it is necessary to implement the whole system from scratch

Existing components were used in implementing the presented system. For example, Visiome Engine and Subversion were used. The way to use existing components presented in this thesis is to wrap the component with a location. This was done to both fore mentioned components, i.e. Visiome Engine and Subversion, in the implementation of the case study. The legacy component wrapped by the location can then be used by agents through the location, and the location can listen to changes in the component, and create new agents to react the changes in needed situations.

7.4. Proposals for Improvement and Criticism

In the implementation done in this thesis there is automatic support for only a simple agent execution state, which gets transferred between areas. This approach is quite li-

mited, and a more versatile solution could be to offer automatic support, for example, for more complex state machine based agent states.

There is currently a thread for each agent in execution. The threading in general can cause some problems if, for instance, an agent crashes in the middle of executing something in a location. There could be some error recovery methods, so that the whole location or possibly the program itself would not crash or jam. One solution could be that the locations would always keep their state saved to the disk, and the locations could be restarted with that state if an agent caused them to crash.

The framework itself provides no way for the agents to communicate with each other in this implementation. This can in some cases be a serious shortcoming. This could be solved with, for example, providing an AgentMeeting-location that could be used to send messages or to leave notes for other agents.

There is currently no automatic support for defining the locations as a composition of role interfaces. This could be supported in some uniform way. For example, the fore mentioned AgentMeeting-location could then be only a role interface that any location could implement.

The agents need to be compiled with the main areas in this implementation. A plug-in system could be used to allow new agents and locations to be added independently of other software changes. In the case of agents this could be added easily. Locations would be a little bit more problematic, but there should be no big problems in that either. Adding locations dynamically would mean that areas with the same type could hold different locations. This would cause that the area type would not be enough to make conclusions about the services that area offers anymore. For example, if an agent would need such information, then it would have to ask the locations of all areas in the system, instead of just using the types of the areas.

7.5. Related Work

Related work is categorized to two different viewpoints: work in the integration domain in general and agent architectures. The viewed agent architectures are not only from the integration domain, but from various different areas. The integration domain is considered first and the agent architectures after that.

7.5.1. Integration Domain

Service-oriented architecture (SOA) [Pap03] is a set of design principles targeting at flexible development of new systems and solving the problem of integration of existing applications. SOA promotes loose coupling of services and the use of high-level languages to orchestrate the use of these services into higher level business logic. Service-oriented architecture has been successfully used at several integration projects, including [Zim04] and [Zim05]. To integrate a system using the agent based architecture with an external SOA system is, at least in theory, relatively easy. It could, for example, be done by creating a location, which accepts external SOA-messages and converts them to

the right agents. Also the same location could convert method calls made by agents to SOA-messages and send them to the right service providers.

In a way, the design principles presented in this thesis are not very far from SOA. The locations could be compared to the service providers of SOA, and the agents to the orchestration of these services. The architectures tackle the same problems, but the implementation of the solutions quite different. In this work it was wanted to avoid the transformation of data to XML and to give the developer of the business logic full control, i.e. the freedom to use the whole expressiveness of the used programming language. Some benefits were gained from this approach, but then again SOA is a more robust solution to some domains. For example, SOA might suite better to inter-organization data and services exchange where the cooperators do not necessarily have a complete trust for each other. The approach presented in this thesis is more suitable for use in a more limited environment with only a single, or possibly a couple of organizations.

7.5.2. Agent Architectures

There exists a lot of research in a multitude of areas involving agents directly or indirectly. For instance, [Man04] gives an overview of agent concepts and applications of agent technology. Baumann et al. [Bau98], Lange and Oshima [Lan99], and Gray et al. [Gra02] have found similar benefits of using agents as were pointed out in this thesis. The experiences with first- and second-year undergraduates successfully developing D'Agent applications [Gra02] also suggested that agents are easier to understand than message- or RPC-based techniques.

There are also numerous agent-based architectures, infrastructures and middlewares, including Mole [Bau98], the Aglet API [Lan98], Open Agent Architecture (OAA) [Mar99], D'Agents [Gra02], RETSINA [Syc03] and Hermes [Cor05]. The middleware presented in Hermes has been successfully used to design an agent-based tool integration system [Cor04]. A summary of several projects using agent technology for enterprise integration and supply chain management is presented in [She99]. Existing agent architectures are discussed and an architectural model for mobile agent systems is described in [Sch03]. Additionally, [Mül02] considers the use of agents in electronic business, including complex integration of existing infrastructures.

A common difference with approach in this thesis and many of the mobile agent systems is that in this approach focus lies in simplicity, which is achieved by restricting the mutual communication of agents to be between agents and locations. This allows the architecture to support flexibility in a controlled manner while still keeping the system easily maintainable. A more specific difference with other agent-based architectures is that there is a special entity called location that provides local services. The decision to call the service provider a location, instead of service agent or static agent, comes from the fundamental differences between agents and locations in the presented architecture. The most relevant differences being that locations are not mobile or goal-oriented and they are permanent.

Architectures containing this kind of an entity are typically the most similar ones to the approach in this thesis. These include EMAA [Len98], which has servers providing services, as well as Hermes and Mole [Bau98] with ServiceAgents. Also docks in EMAA have some similarities with the transporters presented in this thesis, but distinctively the transporters only handle things related to the communication over network. This makes the architecture clearer and reusable, since if many communication protocols are needed, an area can contain several transporters of different types. In addition, the approach presented in this thesis does not rely on the need for each node or transporter to be able to connect to all other areas or to a centralized naming directory or resource server. On the contrary, the architecture model can be built in a way that the transporters work like routers and only know the next destination while asked for a certain type of a service. This is beneficial in several cases, for example, if communicating through several firewalls.

7.5.3. Process Support Systems

There are several resemblances and differences between the case study implemented in this thesis and the architectural commonalities of existing PSEEs presented in section 3.5. The common components in those existing PSEEs were a user interface facility, a process engine, and a repository [Fug96].

The implemented case study has similar components. However, the interaction of the components is different. For example, the state of the process is at all times saved to the repository, and the user interface can then use the repository to show the state to the user. In addition, the process engine does not use the repository directly, and therefore is not tightly coupled with it. Finally, the case study implementation is inherently distributed by the use of the agent-based architecture.

8. CONCLUSIONS

In this thesis, an agent based architecture style was presented and specified in three meta-levels. It was also shown how higher level of abstraction in dependencies, and agent based communication are feasible solutions in application integration. The approach was validated by implementing a framework for agents and by using it to create the architecture for a process support environment. In addition, an example showed how the way of specifying the architecture can be used also in specifying reusable architectural patterns (observer pattern example).

The presented architecture style attains a relatively good level of flexibility, customizability, and maintainability, as well as provides means for incremental development. These qualities are attained, for example, because of the easiness of adding new entities to the system and keeping each business logic case in a single place. The architecture style is also simple, concrete, and well defined. There are some similarities to existing architectures, including other agent-based architectures and SOA.

The architecture model and the implementation of the case study could be improved and extended in many ways. For example, graphical specification of architecture metamodel combined with code generation facilities, as well as simple mechanisms for defining at least the simplest agents, like in BPEL (Business Process Execution Language) for Web services [IBM06], might be useful. More extensive practical tests about performance, suitability, etc. would help to understand all the benefits and disadvantages concerning the architecture model.

There were two reasons to create a framework instead of just a single process support application specific to the requirements presented in this thesis. First of all, the concepts presented in this thesis are not only more reusable, but also easier to understand when presented in two distinct parts. In addition, there was a real general need for an integration architecture, and the agent based architecture presented in this thesis addresses this need. More general arguments for the architecture are presented in Peltonen et al. [Pel09]. The architecture has already proved itself useful and applicable to other environments. A working first version of a repository based modeling environment has already been implemented using the concepts presented in this thesis. The repository based modeling environment is implemented in c# and Java. The modeling environment is still a work in progress, but currently the concepts have fit into that domain without great difficulties.

REFERENCES

[Air02] Airaksinen J., Koskimies K., Koskinen J., Peltonen J., Selonen P., Siikarla M. and Systä T., xUMLi, towards a tool-independent UML processing platform. In Proceedings of 10th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'2002), Copenhagen, Denmark, August 2002.

[Aoy98] Aoyama M., Agile Software Process and Its Experience, Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, 1998, pages 3-12.

[Bau98] Baumann J., Hohl F., Rothermel K., Straßer M., Mole – Concepts of a mobile agent system, World Wide Web 1 (1998) 123–137.

[Bus96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996, pp. 339-343.

[Cha04] Chappell D., Enterprise Service Bus, O'Reilly, 2004.

[Chu03] Idea from Hyacinth S. Nwana, Divine T. Ndumu: An Introduction to Agent Technology, Re-Drawn by Mobile Computing, Dept. of IECS, Feng Chua University, R.O.C., 2003., WWW-Document, http://upload.wikimedia.org/wikipedia/en/2/2e/Ch1-Nwanna.gif, referenced 2.2.2010

[Con02] Conradi R., Fuggetta A., Improving Software Process Improvement. IEEE Computer Society Press, Vol. 19, Issue 4, July 2002, pages 92-99.

[Cor04] Corradini F., Mariani L., Merelli E., An agent-based approach to tool integration, International Journal on Software Tools for Technology Transfer (STTT), Volume 6, Issue 3, Aug 2004, Pages 231 – 244.

[Cor05] Corradini F., Merelli E., Hermes: Agent-Based Middleware for Mobile Computing, Lecture Notes in Computer Science, Volume 3465, Jan 2005, Pages 234 – 270.

[Fra96] Franklin S., Graesser A., Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[Fug00] Fuggetta A., 2000. Software process: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM, New York, NY, 25-34. DOI= http://doi.acm.org/10.1145/336512.336521

[Fug93] Fuggetta A., A Classification of CASE Technology, Computer, pp. 25-38, December, 1993

[Fug96] Fuggetta A., Functionality and architecture of PSEEs, Information and Software Technology, 38 (4 SPEC. ISS.), pp. 289-293, 1996

[Gam94] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, pp. 26, Addison-Wesley Processional, 1994

[Gec08] Project homepage, Developer center for Gecko, WWW-document, 2008, https://developer.mozilla.org/en/Gecko, referenced 20.8.2008

[Gra02] Gray R., Cybenko G., Kotz D., Peterson R., Rus D., D'Agents: Applications and performance of a mobile-agent system, Softw. Pract. Exper. 2002; 32:543–573002E.

[Hai06] Haikala I., Märijärvi J., Ohjelmistotuotanto, 11th edition, Talentum, 2006, pp. 318-319.

[Hoh03] Hohpe G., Woolf B., Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003.

[Hya96] Hyacinth S. Nwana, Software Agents: An Overview, Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996.

[IBM06] Specification homepage, Business Process Execution Language for Web Services Version 1.1, WWW-document, 2006, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, referenced 30.6.2006

[Jen01] Jennings, N.R., An agent-based approach for building complex software system, Communications of the ACM 44 (4), pp. 35–41, 2001.

[Kee04] Keen M., Acharya A., et al., Patterns: Implementing an SOA using an Enterprise Service Bus, IBM Redbooks, 2004, http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf

[Kos05] Koskimies K, Mikkonen T., Ohjelmistoarkkitehtuurit. Talentum, 2005.

[Lan98] Lange D., Oshima M., Mobile agents with Java: The Aglet API, World Wide Web 1 (1998) 111–121.

[Lan99] Lange D., Oshima M., Seven Good Reasons for Mobile Agents, Communications of the ACM, March 1999/Vol. 42, No. 3.

[Leh91] Lehman M., Software engineering, the software process and their support, Software Engineering Journal, Vol 6., Issue 5., 1991, pages 243-258.

[Len98] Lentini R., Rao G., Thies J., Kay J., EMAA: An Extendable Mobile Agent Architecture - AAAI Workshop on Software Tools for Developing Agents, 1998.

[Man04] Manvi S., Venkataram P., Applications of agent technology in communications: a review, Computer Communications 27 (2004) 1493–1508.

[Mar99] Martin D., The Open Agent Architecture: A Framework for Building Distributed Software Systems, Applied Artificial Intelligence, 1999.

[Mic10] Microsoft Corporation, Component Object Model, WWW-document, 2010, http://msdn.microsoft.com/en-us/library/ms694363.aspx, referenced 31.1.2010

[Mül02] Müller J., Bauer B. and Berger M., Software Agents for Electronic Business: Opportunities and Challenges, Lecture Notes in Computer Science, Volume 2322, Jan 2002, Page 61.

[OMG02] Object Management Group, OMG-Meta Object Facility, version 1.4, WWW-document, 2002, http://www.omg.org/cgi-bin/doc?formal/2002-04-03, referenced 18.2.2010

[OMG06] Object Management Group, Meta-Object Facility, version 2.0, WWW-document, 2006, http://www.omg.org/spec/MOF/2.0/, referenced 2.2.2010

[OMG07] Object Management Group, Unified Modeling Language, version 2.2, WWW-document, 2007, http://www.omg.org/cgi-bin/doc?formal/09-02-02, referenced 2.2.2010

[Pap03] Papazoglou M, Service-oriented computing: concepts, characteristics and directions, Web Information Systems Engineering, 2003.

[Pel00] J. Peltonen, "Visual Scripting for UML-Based Tools", In Proceedings of ICSSEA 2000, Paris, France, December 2000.

[Pel04] Peltonen J., Selonen P., An Approach and a Platform for Building UML Model Processing Tools, Proc. Workshop on Directions of Software Engineering Environments (WoDiSEE04), IEE Publications, Edinburgh, 2004, pages 51-57.

[Pel09] Peltonen J., Vartiala M., An agent based architecture style for application integration, Annales Univ. Sci. Budapest., Sect. Comp., vol. 31, pp. 3–22, 2009.

[Roy70] Winston W. Royce, Managing the development of large software systems, In Proceedings IEEE WESCON, 1970, pages 1-9.

[Sch03] Schoeman M., Cloete E., Architectural components for the efficient design of mobile agent systems, In proc. of SAICSIT, 2003.

[She99] Shen, W., Norrie, D.H., Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. Knowledge and Information Systems, an International Journal, 1(2), pp. 129-156, 1999.

[Som07] Sommerville, I. Software Engineering, Eigth Edition, Addison-Wesley, 2007.

[Syc03] Sycara K., Paolucci M., Velsen M., Giampapa J., The RETSINA MAS Infrastructure, Autonomous Agents and Multi-Agent Systems, Volume 7, Issue 1 - 2, Jul 2003, Pages 29 – 48.

[Var07] Vartiala M., Peltonen J., An agent based architecture style for application integration, In proceedings of the 10th SPLST, 2007, pages 214-228.

[Vli00] Hans van Vliet. Software Engineering: Principles and Practice, 2nd Edition, Wiley, Sept 2000

[Woo98] Wooldridge M. Jennings, N., Pitfalls of agent-oriented development, AGENTS '98: Proceedings of the second international conference on Autonomous agents, ACM, pp. 385--391, 1998.

[Wxw09] Project homepage, wxWidgets Cross-Platform GUI Library, WWW-document, 2009, http://www.wxwidgets.org/, referenced 25.6.2009

[Zam03] Zambonelli, F., Wooldridge, M., Jennings, N.R., Developing multiagent systems: the Gaia methodology. ACM Transaction on Software Engineering and Methodology 12 (3), pp. 417–470, 2003.

[Zim04] Zimmerman O., Milinski S., Craes M., Oellermann F., Second generation web services-oriented architecture in production in the finance industry, OOPSLA, 2004.

[Zim05] Zimmerman O. et al. Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned. OOPSLA, 2005.