SOCIAL PATTERNS FOR QUALITY CONTROL IN MULTI-AGENT SYSTEMS

Thi-Thuy-Hang Hoang and Manuel Kolp
Information System Unit, Louvain School of Management, Université Catholique de Louvain
Place des Doyens, 1, 1348 Louvain-la-Neuve, Belgium

Keywords: Social pattern, Multi-agent system, Quality requirement.

Abstract: Modern softwares are not only required to perform some specific functions but also have to satisfy all the quality constraints described by the initial requirements. In this paper, we focus on some useful social patterns that will facilitate the developers' task when dealing with quality constraint in multi-agent systems. These patterns define some agents and their interactions that help to monitor and to react to any changes of quality at the runtime.

1 INTRODUCTION

Nowadays, software plays a crucial role in every corner of the modern life. The numerical era requires people to be equipped with at least a minimal knowledge of information technology. Online shopping, e-newspapers, cyber games, social networking, etc. become daily activities. Daily bank transactions, administrative obligations… are also computerized. This *electronic* life has transformed people into e-citizen, e-learner, e-banker, e-shopper, etc.

Being an emerging development paradigm that has some advantages over traditional development techniques such as structured and object-oriented, *agent-oriented software development* has become a modern buzzword in software engineering. Software agents, with certain autonomy and intelligence, are expected to substitute human agents in a lot of tasks. Moreover, they are designed to live in a virtual society of agents who interact with each others to exchange their knowledge, to reason about the environment and to act towards individual goals as well as social goals.

It is known that the agent-oriented paradigm favours extensibility and interoperability. But these qualities cannot be automatically attained by just choosing to use an agent-oriented software development methodology. A good analysis of requirements and a sound design could help any software system to have these qualities even if they are not built using agent-oriented software development methodology. Analysis and design are among the deciding factors contributing in the development of quality software.

Extensibility and interoperability are only two among many other *quality requirements*. Quality requirements are also considered as *non-functional requirements* (Chung, Nixon, Yu and Mylopoulos, 2000) describing HOW the system will do, in contrast to *functional requirements* describing WHAT the system will do. In goal-based approach, quality requirements are described as a subset of *soft-goals* (Castro, Kolp and Mylopoulos, 2002) whose satisfaction conditions are not defined in a clear-cut way. Recently, (Jureta, Mylopoulos and Faulkner, 2008) revisited the core ontology of requirement engineering by redefining a list of basic concepts: goal, soft-goal, quality constraint, plan, and domain assumption. According to this, quality constraints are well defined and verifiable while soft-goals are abstract qualities whose verifiability is ill-defined and usually subjective. Another attempt to clarify the role of quality requirements in the *goal-based* approach (Hoang, 2008), (Hoang and Kolp, 2009) separates *quality requirements* from *soft-goals*. Based on the measurability, quality requirements are classified into *measurable*, *partly measurable*, *heuristically measurable* and *immeasurable*. When quality requirements are not *immeasurable*, one can still find some measurements that can provide some idea about the fulfilment of such quality requirements. This might be directly applied in any system to reduce the number of violations of quality requirements.

However, in a multi-agent system, the global goal is designed to be attained only by the individuals' collaborations governed by the individuals'

Hoang T. and Kolp M. (2009).
SOCIAL PATTERNS FOR QUALITY CONTROL IN MULTI-AGENT SYSTEMS.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 336-343
DOI: 10.5220/0002259403360343
Copyright © SciTePress

goals. The system is a 'society' where individuals are free, to a certain extent, to choose what they do. As a consequence, more questions about the quality requirements must be asked on those systems than on traditional systems. It is also more difficult to deal with quality requirements in a multi-agent system than in a traditional system.

In this paper, we will present a catalogue of *social patterns* that help software developers to design and build quality-aware agent software. Social patterns are design patterns (Gamma, Helm, Johnson and Vilssides, 1995) to which some additional dimensions have been added: *social, intentional, structural, communication* and *dynamic* (Kolp, Do, Faulkner and Hoang, 2005). For object-oriented approach, design patterns have been being very useful tools for designing good software system. Therefore it is quite natural to bring the existing design patterns and to create new design patterns for multi-agent systems, as done by a number of works (Gamma, Helm, Johnson and Vilssides, 1995), (Kolp, Do, Faulkner and Hoang, 2005), (Aridor and Lange, 1998).

We adopt *i** (pronounced as eye-star) notions (Yu, 1995) to describe relations between agents. The essentials of *i** are *dependencies* between pairs of agents. In a dependency, the agent that depends on the other is called the *depender* and the *dependee* is the one that is depended upon. The *depender* depends on the *dependee* to have access to a *resource*, to do a *task* or to achieve a *goal* (*soft-goal* or *hard-goal*). Besides, we use also the *Agent Unified Modelling Language (AUML)* (Bauer, 1999) to model agents' structure using class diagrams and agents' interaction using sequence diagrams and collaboration diagrams.

This paper will be organized into four main parts. This introduction is followed by a section that positions the proposed patterns inside our current research. The third part describes the patterns. An example application of a high availability printing service will be presented in the fourth part before the final remarks and conclusions.

2 QUALITY-AWARE AGENT-ORIENTED SOFTWARE

One of our research objectives about agent technology (Castro, Kolp and Mylopoulos, 2002) is to establish a complete *quality-aware* development process for multi-agent systems. The resulted process should cover all the main development phases ranging from *early* and *late requirements* to *architectural* and *detailed designs* and can be extended to the *verification* and the *deployment* of the system.

There are evidences showing that using agent notion and goal-based analysis might reduce the gap between the *requirements* and the final system (Castro, Kolp and Mylopoulos, 2002). This is because the notion of agents and their goals provide a modelling language for capturing *requirements* and are the reflection of real stakeholders and their intentions in analysing *requirements*. Later on, agents and goals become main elements of the system design reflected in the final product using agent-oriented programming languages (JACK, 2002), (Bellifemine, Poggi and Rimassa, 2001). This makes client *requirements* the principal force that drives the development process.

An example of a complete goal-based development process is defined by the Tropos project (Castro, Kolp and Mylopoulos, 2002). However, the current Tropos process does not deal with the *quality requirements*. In (Hoang, 2008), (Hoang and Kolp, 2009), *quality requirements* are added into the goal-based analysis. The main objective of this research is to capture the quality requirements, to analyze and refine all the requirements and to design the final system that meets the functional requirements and is aware of all the quality requirements. The following section is one of the possible measures for the very last stage of design and implementation of the system.

A well designed system should satisfy most of the initial requirements including quality requirements. However, it is quite common that some quality requirements are left unsatisfied by the design and needs to be controlled at the runtime. Moreover, qualities may be influenced by external conditions that do not exist at the design time. As a consequence, quality *control* is needed to assure the quality of offered services at the runtime. T o deal with quality requirements, in the earlier development stages (Hoang and Kolp, 2009), we use the enriched *i** notions to add the notion of *quality requirements*. Quality requirements are captured as early as other requirements. Then, they are analyzed, decomposed and operationalized during the process. As mentioned above, it is common that after some design iterations (i.e. *architectural design and detailed design*), some quality requirements are left unsatisfied as in the simplified example in Fig. 1.

We analyze an online store namely *E-Shop* in which there is a *Banking Agent* that checks the shop's banking account for payment arrivals. Any received payment must be reported immediately to *Order Manager*. Since fast shipping is always an important factor for gaining the confidence of clients,

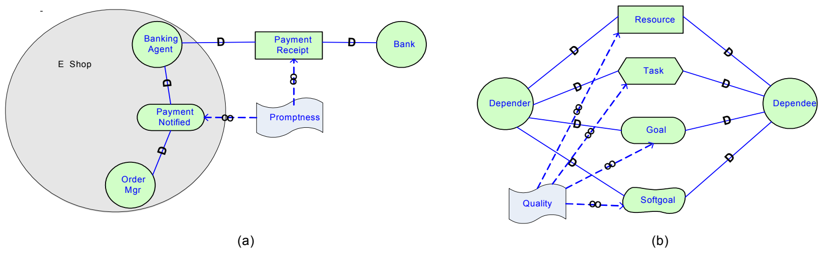

Figure 1: E-shop example (a) modelled by a enriched version of i* notion (b).

we impose (through the requirement analysis) that any orders should be shipped *as soon as* the payment is received. To do that *Promptness* quality requirement is required for *Payment Receipt* (resource) and *Payment Notified* (goal). We decided to look at it closely to detect any suspect delay between *deposit time* figuring in the payment receipt and the *notification time* to *Order Manager* and to adjust, at the runtime, the interval between two consecutive checks.

Above is a very simple example where quality requirements are easily measured and assured at runtime. In real systems, there could be many quality requirements interacting positively and/or negatively with each others. This is why we try to introduce, in the following section, a set of simple social patterns that would facilitate the job of the designers in a systematic way. As we will see, those patterns would increase the awareness of the quality of its services. In the final product, the quality control mechanism can be integrated into a subsystem called *Quality Management Subsystem* that can be switched *on* or *off* at any time.

3 SOCIAL PATTERNS FOR QUALITY CONTROL

It is known that quality requirements could be favourably treated by choosing the right options during the analyse as well as the design of the system. However, in most cases, it is impossible to find a design solution that fulfils *all* the quality requirements. Based on the fact that almost any quality requirements can be at least heuristically measured, we then have a possibility to monitor in order to react whenever any quality requirement is violated at the runtime. The following is the description of some *social patterns* for designing the *Quality Management Subsystem* that can be built as a sub-system inside the system-to-be. Quality requirements to be monitored are those of the final system but not those of the development process. Usually, quality requirements of the development process such as: low cost, time constraint, etc. are usually fulfilled by the choices in means-ends analysis, designs' structures and implementation styles.

The desired subsystem is built based on the following remarks:

- Qualities can be measured at least heuristically or partly. Judgments of human agents can be also considered as a measurement.
- A quality metric can use more than one source of signal. As a consequence, a quality meter can be subscribed to all the necessary sources.
- Signal sources can be states, variable values, etc. Changes in signal needed for the measurement of a quality requirement are usually detected at places having that quality requirement.
- For a quality requirement, there can be more than one quality managers (who keep track of the quality fulfillment and decide to carry out necessary actions). This implies that quality violations can be reported to more than one manager.
- Monitoring continuously necessary signals can be very costly. Hence, the subsystem must offer the polling (or sampling) mode in its implementation. This means that not every changes of signal will be taken into account. Only when the signal is needed, the current value can be pulled out from the signal source.

In the next section, we will detail the proposed patterns that facilitate the designer in designing the quality management subsystem as well as in incorporating this subsystem into the system-to-be. The description of the following social patterns follows SKwyRL Social Pattern Framework (Kolp, Do, Faulkner and Hoang, 2005).

3.1 Social Dimension

The subsystem will have the following possible roles played by agents. Notice that each agent in the system can play many roles at the same time.

- *Signal Source.* emits signals used in the measurement of some quality requirements on the system. Signal source can be also human, for example, users of a website that fill in an evaluation form about their satisfaction.

- *Signal Manager*. maintains a currently updated list of available signal sources together with the type of signal and possible modalities of acquisition (pulling and/or pushing).
- *Quality Meter*. combines different signals to calculate indicative values reflecting the fulfilment degree of quality requirements.
- *Quality Manager*. responsible for detecting any quality violations using the indications given by quality meters as well as for checking required conditions before doing some actions.
- *Quality Assurer*. agents that are required to fulfill some quality requirements for its attributes, plans or operations. They are the ones that trigger the procedure of quality control.

The social dimension identifies not only the relevant agents and roles but also the intentional dependencies between these roles/agents. The following paragraphs are intentional dependencies between agents inside each of the described social patterns.

3.1.1 Signal Pushing Pattern

This pattern specifies the interactions between a *Signal Source* and *Quality Meter*. It is favourable for cases where every change in the signal is pushed to the *Meter*. Main interactions are summarized in the social diagram in Fig.2.

For the meter, in order to receive the signal from the signal source, every *Quality Meter* needs to subscribe itself to *Signal Sources*. At the *Signal Source*, there is a list of current subscribers to which the *Signal Source* must send any change in signal. *Signal Sources* are discovered by *Quality Meters* with the help of *Signal Manager* who keeps track of a signal directory.

3.1.2 Signal Pulling Pattern

Contrary to the pushing pattern presented above *Quality Meters* play an active role in the *Signal Pulling Pattern* presented in this section. The signal is acquired only by demand determined by the monitoring strategy of the signal monitor. The social diagram is as in Fig.3.

For a *Quality Meter*, first he has to connect to the *Signal Source* and keep this connection. Depending on the monitoring strategy usually supported by a schedule, the monitor will pull the current value of the signal from the signal source. In this pattern, the *Signal Manager* helps *Quality Meters* to discover and to keep track of the availability of *Signal Sources.*

One could argue that the two above patterns are similar to the well-known *Observer* pattern (Gamma, Helm, Johnson and Vilssides, 1995) with a *Matchmaker* pattern (Kolp, Do, Faulkner and Hoang, 2005). However, in the context of multi-agent systems and quality control subsystem, we can point out here some reasons of not to separate these patterns:

- The presence of a *Signal Manager* is mandatory, since it guarantees and verifies the availability of signal sources. Without this, quality requirements cannot be correctly controlled.
- Signal sources may not be agents in the system-to-be. This implies that the *pushing* mode might not be available. Quality meters must *sense* itself any changes in the signal.
- The presence of all the three agents *Signal Source*, *Signal Manager* and *Quality Meter* represents the *existence* constraint of these agents inside the system-to-be.

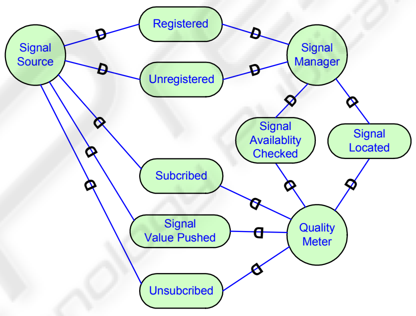

Figure 2: Signal Pushing Pattern.

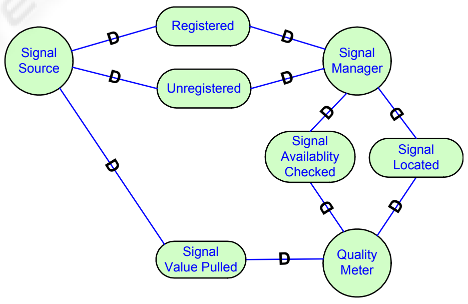

Figure 3: Signal pulling pattern.

3.1.3 Quality Assurance Pattern

This pattern focuses on the interventions into the normal execution of an agent in order to control the fulfilment of any quality requirement. There are three types of interventions:

Precondition check: when agent is about to take an action, it needs to verify whether the preconditions on the quality are met. The preconditions define the states of the world, including the state of the quality fulfilment at which the action can be carried out. If the current state is not favourable for the requested action, the quality manager could carry out some additional plans to bring the system into a favourable state. If he has already tried every possible plan without success, he will have to notify the plan executer to cancel the requested plan.

Intermediate check: during the execution of an action, it is possible that the fulfilment of a quality requirement required is violated. The cause may come not only from the plan itself but also from other agents in their continuously changing environment. This is also possible that at each stage of the action, the quality requirements are different. In either case, intermediate checks are very important. These checks can be carried out by predefining a list of several check points, usually vulnerable points to the quality fulfilment, together with conditions to be satisfied at each check point. When a condition is not met, the managers will try to do additional work to satisfy the condition. In this case, the plan executor is notified and may decide to continue with the plan or to start with other plans or to give up.

Post-condition check: carried out when the plan is done. This can confirm that the plan has been carried out successfully and that all the quality requirements are fulfilled.

The pre-condition and the post-conditions have usually been known before the plan is started, while the intermediate conditions are variable in accordance to the operations really carried out by the plan.

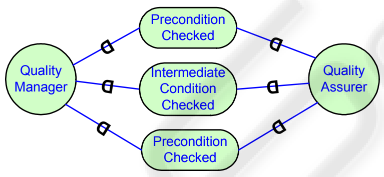

Figure 4: Quality assurance pattern.

The social diagram is presented in Fig.4. Here we omit the relation between *Quality Manager* and *Quality Meter* which can be characterized by the well-known *Observer* pattern in which *Quality Manager* is the observer that observes measurements carried out by *Quality Meter*.

3.1.4 Total Quality Manager Pattern

In a complex system, when the number of quality requirements becomes large, we need to distribute the responsibility among several *Quality Manager*. This partition can be made based on:

- Topology Setup. a system may operate on different geometrical or logical sites. Therefore, to control and assure each quality requirement, one can eventually put a *Quality Manager* on each site.
- Quality Relations. a quality requirement can usually be split into several sub-quality requirements. For example, *Security* can be split into *Confidentiality, Integrity*, etc. One can create a quality manager for each of the sub-quality requirements.
- Manager Hierarchy. one can also decide to put quality managers in to a hierarchy where lower-rank managers have to report to their direct higher manager.

Since our ultimate objective is to build systems in which every quality requirement could be controlled and assured, we include here the *Total Quality Manager* pattern that does the aggregating job among the managers. Top *Managers* will be the ones that communicate the overall status of quality fulfilment to system administrators.

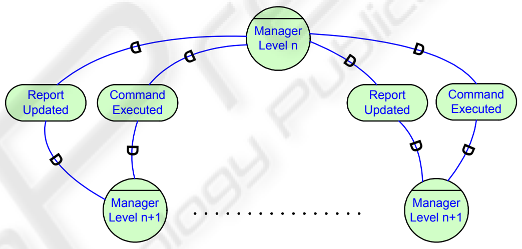

Figure 5: Total quality manager.

With this pattern, one can have a total control of the *Quality Management Subsystem* even in a very complex system. Quality managers can be organized in to several levels depending on the organizational structure of the system. However, the designer should limit the maximum height of the manager hierarchy to reduce the loss of control and the latency of the subsystem due to the overhead of the heavy organization. Furthermore, the system's designer could also decide to give different manager roles to a single manager agent when needed.

As an immediate use, this hierarchical structure can provide a simple way to attach and to detach any part of or the whole quality control subsystem from/to the main system when needed. This is useful when the operation of the system becomes sufficiently stable and the agility of some services becomes occasionally necessary.

3.2 Intentional Dimension

Having identified the interdependencies between agents, we now focus in the rationale of each agent/role. To keep the paper concise, we show here only some important abstract services offered by the above agents.

Table 1: Agents' services in quality control.

Service Names	Informal description	Agent
FindSignalSrc	Find all signal sources that match the specified description.	Signal Manager
AvailCheck	Check the availability of a signal source	Signal Manager
ConflictResolve	Resolve any conflict in fulfilment of quality	Quality Manager
StateReport	Report the fulfilment to the superior	Quality Manager
QualityAssure	Take necessary actions to guarantee the quality	Quality Manager
...	...	...

Among the above services, *ConflictResolve* and *QualityAssure* are two of the most important. While the latter tries to fulfil a quality requirement, the former verifies if the fulfilment of that quality requirement will be harmful to the others in the system. Some trade-offs could be made because, in a complex system, quality requirements are usually contradictory when being fulfilled.

In SKwyRL, services are defined formally using the formal Tropos language which is skipped in this paper. Services are operationalized into plans that are described in the structural dimension.

3.3 Communication Dimension

Communication dimension emphasizes the exchange of events between the agents.

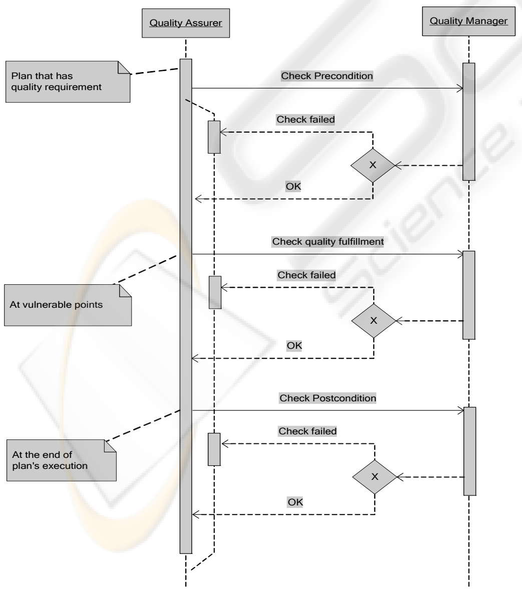

Figure 6: Quality assurance.

Figure 6 shows an execution of a plan that the agent *Quality Assurer* carries out. Since it is required that the execution must satisfy some quality requirements, *Quality Assurer* has to verify the fulfilment of those requirements at the beginning of the plan, during the plan (e.g. at predefined checkpoints) and after the plan.

We skip the descriptions of others plans as well as the definition of the described social patterns in the structural dimension, where services are decomposed into agents' *Belief*, *Event* and *Plan*, and in the dynamic dimension, where the relationships between plans and events are elaborated. We will now take a look at a simple example into which the described patterns are applied.

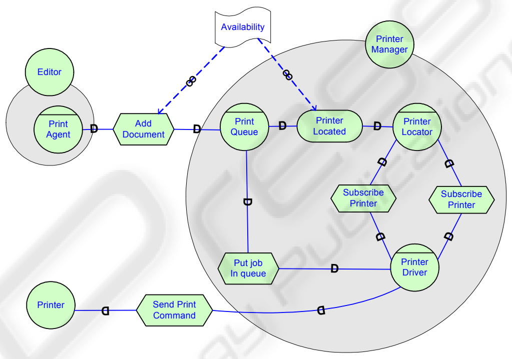

Figure 7: Highly-available print service.

4 HIGH AVAILABILITY PRINT SERVICE

We consider an example about a printing service inside a network where there is a requirement that states: some printers must be turned on so that editors are able to print a document at anytime as described in the architectural design in Fig 7. First, we must identify the signal sources that can provide indicators of printers' availability:

- Printer State. standby, stopped, started, connected or disconnected. This availability signal can be monitored directly from the printers.
- Printer Driver State. absent, present or defected. When a printer driver is present and operational, it can give greater details about the printer, if the printer is online.
- Print Queue State. details of a number of documents in the queue which could make the printing of the newly introduced document unavailable during a period of time.

For the availability signal, since the printers may be disconnected accidently from the network, they cannot notify themselves their absence to the printer manager, therefore we decide to check the availability of each printer by using the pulling scheme.

Since, these signals can be the indicators of other quality requirements of the system such as: *Energy Efficiency* and can be inputs for many quality metrics in the system, we may also implement some intermediate agents that relay and transform the signal before the final signal is provided to the quality managers and quality executors. The first part of the subsystem is the monitoring part by which the *Printer Availability Manager* senses the availability state of the system's printers.

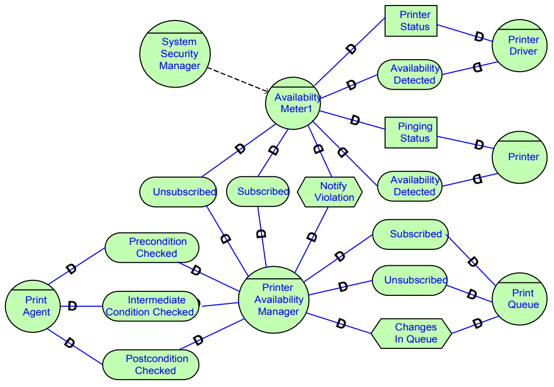

Figure 8: Highly-available print service.

In Fig.8, three identified signal sources are added. For *Printers* and *Printer Drivers*, we use the Signal Pulling Pattern to get the status of the printers and printer drivers. All the printers physically installed in the system are pinged by the agents *Availability Meter1* at every small interval of time (e.g. 0.5 seconds). The drivers of *joinable* printers are also probed at a larger interval (e.g. 5 seconds). The third useful signal used to determine the availability of the printer is the length of the *Print Queue*. Because this signal can be used directly by the *Printer Availability Manager* and this signal is changed by precise events, we use the pushing pattern to report the queue status. Using this structure, the *Printer Availability Manager* has, at every moment, the availability of all the printers available inside the system with the precision of at least 5 seconds (equals to the interval between two consecutive probing of printer driver).

We now detail the *Printing Plan* of the *Print* Agent of the *Editor*. Before printing, the *Print Agent* refers to the *Printer Availability Manager* to check whether the document can be printed in an expected time, e.g. 2 minutes. At the *Manager* side, it checks the printer capabilities and the size of the print queue to estimate the expected delay. If the delay is greater than the requested delay, it tries to start a standby printer. If it finds a way to respect the delay, it returns a positive answer to the editor to begin the printing job. If it fails to make the printing system available, it sends a negative answer to the requesting *Print Agent* and sends a detailed report to the human manager about the overload problem.

A portion of the class diagram focusing on the quality management subsystem is shown in Fig.9. The main class in *Availability* control is the agent *Printer Availability Manager*. It subscribes itself to the *Print Queue* and the *Availability Meter 1* to receive the update of the *Queue* state and *Printer*'s state. From the received data, it can estimate the maximum delay for a new job requested by any *Print Agent*. *Print Agent* has a reference to the *Print Queue* to send printing jobs when it is possible. At the *Printer Driver* end, each driver has a reference to the corresponding *Printer*. We include an additional agent *Printer Monitor* that can be considered as another metric that combines information from *Printer Driver* and *Printer Queue*. The *Printer Monitor* has references to all the *Printers* and their software drivers in order to ping and query the configuration of the *Printers*. It also has a reference to some *Meters*, the *Availability Meter1* in this case, to notify any changes in the *Availability* and the configuration of *Printers* and *Printer Drivers*.

The *Print Agent* may have references to many other quality managers at the same time for example: *Energy Saving Manager* and *Printing Quality Manager* to assure the quality of its printing plan. In Fig.9, we omit other details and only keep the most important parts necessary in the *Printer Availability Management Subsystem*.
The printing plan of the *Print Agent* is triggered when the user hits the print command in the editor's interface. It first determines the maximum delay that this job can allow by consulting the user or taking the delay in the global configuration of the system. It then asks the *Printer Availability Manager* to check the delay constraint using the current state of *Printers* and the *Print Queue*. It will try to do some additional things to try to assure that the printing job will be finished in time.

If the *Printer Availability Manager* fails to make the print plan finish in time, the *Editor* will show a notice to the user and ask for the new instruction. If the printing plan is approved, the *Print Agent* connects to the *Print Queue* and starts to send the data to the *Print Queue*. During and after the transmission, the editor continues to refer to the *Printer Availability Manager* to update the expected finish time of the current job depending on the actual state of the printers and the time used for the transmission, if the delay constraint is violated, the manager will try to intervene again and the user will be also notified. Normally, the expected due time

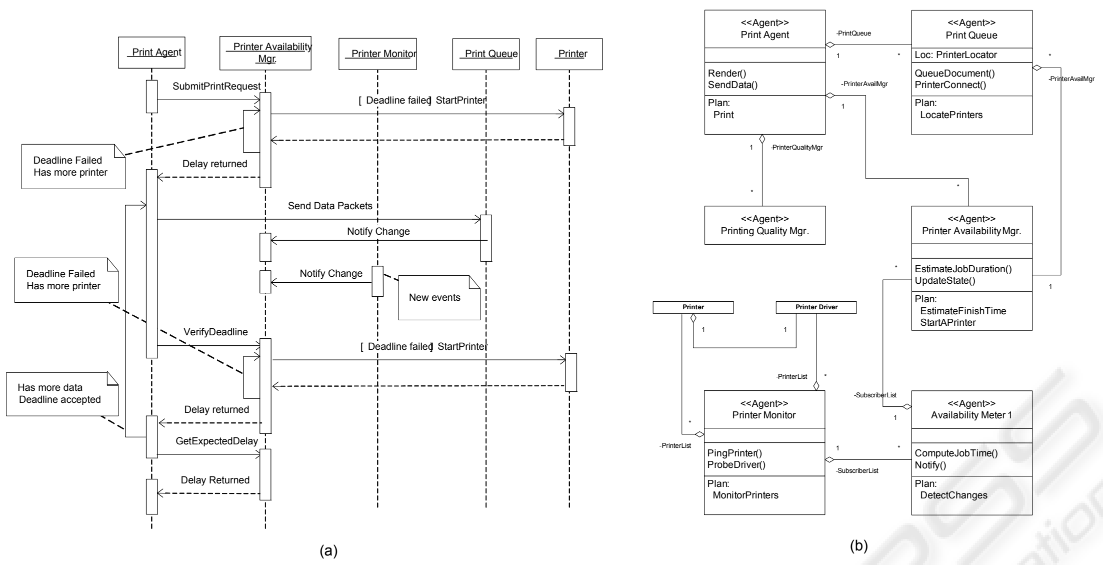

Figure 9: Top-level interaction of Printing Plan (a) and the Agent Diagram (b).

after the transmission is not far from the reality, since the print job is already in the *Print Queue*, unless one of the printers fails or is accidentally stopped. When this happens, the *Printer Availability Manager* will try to make more printers available to guarantee the deadlines.

Figure 9a shows the top level of the *Printing Plan* of the *Print Agent*. More details can be also added using lower-level of sequence diagram or using the plan diagram.

5 CONCLUSIONS

Multi-agent framework has been gaining its popularity. However, there are still not many commercial products that are implemented using the multi-agent technology. The *too* flexible structure of agent-based systems may not encourage the software industry to adopt the agent-oriented methodology. *Flexibility* is not the only quality that is required by software quality. Indeed, qualities such as s*ecurity*, *availability, traceability,* etc. have become vital for modern software.

This paper is an attempt to help multi-agent systems to cope efficiently and systematically with quality requirements. The proposed social patterns together with a sound development process could give the agent-oriented framework a bigger role in the continuously-growing industry of software.

REFERENCES

Chung, L., Nixon, A. B., Yu, E., 2000. Mylopoulos J., *Non-functional Requirements in Software Engineering*, Kluwer Academic Publishers.

Castro, J., Kolp M., Mylopoulos, J., 2002. *Towards Requirements-Driven Information System Engineering: The Tropos Project*, Information System Journal, 27: 365-389.

Jureta, I. and Mylopoulos, J. and Faulkner, S., 2008. *Revisiting the Core Ontology and Problem in Requirements Engineering*, Proceedings of the 2008 16th IEEE International Requirements Engineering Conference: 71-80.

Hoang, T.T.H, 2008. *Quality-aware agent-oriented software development*, internal report, Louvain School of Management, Université catholique de Louvain.

Hoang, T.T.H and Kolp, M., 2009. *Goal, Soft-goal and Quality requirement*, submitted for publication.

Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J., 1995. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley.

Kolp, M., Do, T.T., Faulkner, S. and Hoang, HTT, 2005. *Introspecting Agent-Oriented Design Patterns*, Advances in Software Engineering and Knowledge Engineering, 3.

Aridor, Y. and Lange, D.B., 1998. *Agent design patterns: elements of agent application design*, Proceedings of the second international conference on Autonomous agents, p108—115.

Yu E., 1995. *Modeling strategic relationships for process reengineering*, PhD Thesis University of Toronto.

Bauer, B., December 1999. *Extending UML for the Specification of Agent Interaction Protocols,* OMG document ad/99-12-03, FIPA submission to the OMG's Analysis and Design Task Force (ADTF) in response to the Request of Information (RFI) entitled "UML2.0 RFI".

Agent Oriented Software Pty. Ltd., Mars 2002. *JACK Intelligent AgentsTM User Guide.*

Bellifemine, F., Poggi, A. and Rimassa, G., 2001. *Developing Multi-agent Systems with JADE*, Springer.