LOUVAIN
School of Management

Quality-Aware Agent-Oriented Information-System Development

HOANG Thi Thuy Hang

Thèse présentée en vue de l'obtention du grade
de docteur en sciences économiques et de gestion

Composition du jury :

Prof. Manuel KOLP (Promoteur – UCL)
Prof. Jean VANDERDONCKT (UCL)
Prof. Stéphane FAULKNER (FUNDP)
Dr. David MASSART (European SchoolNet)
Prof. Thierry VAN DEN BERGHE (UCL/ICHEC)
Prof. Per AGRELL (Président – UCL)

Décembre 2010

ABSTRACT

In the modern world, where most activities are computerized, ill-built software could result in disasters. Major flaws keep being spotted in software for space rockets, train schedulers, bank transactions, internet services, etc. In many cases, integrated information systems, although built with a "complete" set of required functions, still do not work adequately in certain circumstances. Such reality forces software developers to widen their focus beyond just functionality and to more effectively take into account quality concerns like security, availability, accuracy, etc. Many research results as well as practical techniques have been introduced, but there are still many areas where existing solutions are not effective enough.

This thesis addresses issues about improving the treatment of quality concerns in the context of multi-agent systems and of goal-based requirements engineering. The originality of our approach relies on a new definition of quality requirements that are presented as constraints bearing on the classical concepts of hard-goals and soft-goals. We extend the goal decomposition tree with a new node type for quality requirement and three new link types: elicitation, qualification, and contribution. High-level business goals (hard-goals and soft-goals) at root nodes are decomposed into lower-lever system hard-goals, at leaf nodes, possibly constrained by some elicited quality requirements. Quality requirements that have not been taken care of in this decomposition process can be controlled at system runtime through a catalogue of design patterns for quality control that we propose in this thesis.

Our goal decomposition with quality requirements and our catalogue of social patterns are applied (1) to extend the Tropos methodology into Quality-aware Tropos, (2) to build a supporting tool for Quality requirement (QCase), and (3) to develop an illustrative case study for improving the availability and customizability of a printing service.

ACKNOWLEDGMENTS

Like every substantial personal work, this research thesis would never have been completed without the help of other people.

First of all, I am grateful to my advisor Manuel Kolp and to my thesis committee (Jean Vanderdonckt, Stéphane Faulkner, Thierry Vandenberghe, David Massart, and Per Agrell) for their comments and feedback during the thesis examination process.

I would like to thank all my close colleagues and staff persons at the Louvain School of Management for giving me a lot of affectionate and encouraging support.

Warm thanks go to the Vietnamese community in Louvain-la-Neuve. You have made my long hard PhD years more pleasant to live, especially during my first years away from home.

My last (but not the least) thanks go to my tender daughter Mai, to my smiling son Nam, to my life partner Diep, to my "anges gardiens" Frédérique and Alain, to my beloved family in Vietnam (cảm ơn mẹ và các anh chị lúc nào cũng bên Hằng), to my gentle family in law (cảm ơn bố mẹ và Minh đã thương và lo cho con nhiều) and to my friends elsewhere for accompanying me through all the ups and downs during the accomplishment of this challenging thesis even at the most difficult moments.

If I forgot to thank anyone for anything, please know that I did not mean that.

Thank you for all – Merci pour tout – Xin cảm ơn!

TABLE OF CONTENTS

Chapter 1 INTRODUCTION

Computing plays crucial roles in modern societies. Ordinary people increasingly need to master at least a minimal knowledge of information technology. Shopping online, reading e-newspapers, playing cyber games, interacting via social networks, handling online banking transactions have become commonplace activities. Mastering the basics of computer knowledge has become a must when applying for a job. This electronic life has transformed people into e-citizens, e-learners, e-bankers, and e-shoppers for whose life electronic devices and software have become required facilities.

Software engineering is central to building the adequate tools. Software must meet various requirements in the most appropriate manner. Software must work seamlessly on various platforms and computer architectures, from desktop PCs to handheld devices. Fully integrated enterprise resource planning (ERP) systems connect all internal business units at global scale of multi-national and multi-continental corporations to manage internal and external resources, e.g., tangible assets, finance, materials and human resources. Information flows both between internal divisions of the organization and across its boundaries to external stakeholders. Still, at such scope where, virtually, no limit exists, the availability, reliability, and safety of software must be constantly maintained, in particular in front of unexpected threats from attackers. Transactions must be secured and sensitive information must remain confidential while user interfaces must be kept as simple and intuitive as possible. Information systems must be able to carry out increasingly complicated tasks on behalf of their users in effective and intelligent ways while remaining robust and scalable.

Powerful tools and methodologies for developing software with those qualities are needed to adequately face those challenges.

1.1. Information System Development Context

Daily intra-enterprise and inter-enterprise transactions have to cope with differences in spatial and economical situations. In order to efficiently coordinate their activities, entire companies are usually modelled and governed by an information system that keeps business transactions compliant with predefined procedural and protocol regulations of the enterprise. Centralized systems are progressively being replaced by distributed servers in production sites and offices between which communications are conveyed through the Internet (ARMBRUST, M. et al., 2009).

Service-oriented architectures (SOA) (ERL, T., 2005) are being widely adopted in software development to build large distributed systems. Software services are loosely-coupled software components that interact with one another across the Internet though standardized protocols. A service receives requests typically in XML (eXtensive Markup Language), processes them, and sends back results in another XML message.

Services can be combined at runtime, since the architecture enables mutual agreement on message structure and on exchange protocol. Services can thus be deployed according to requests for new functionalities or computing power. Since inter-service compatibility is controlled by message-exchange protocols, service developers have a certain freedom in selecting the most suitable programming languages, operating platforms, and other supporting tools suited to specific needs.

Service-oriented architectures have been evolving into agent-oriented architectures where agents enjoy greater autonomy and intelligence.

Agent-oriented software development (ARIDOR, Y. and Lange, D.B., 1998) (HENDERSON-SELLERS, B. and Giorgini, P., 2005) is gaining popularity over traditional development based on functional or object-oriented paradigms. It is becoming feasible to delegate to intelligent agents some time-consuming tasks like searching the web, e-shopping, price tracking, etc. Like their human counterparts, agents exhibit some autonomy and intelligence (ODELL, J., 2000). They are designed to live in a virtual society of agents that interact with each other to exchange knowledge, to reason about their environment, and to act in order to achieve their individual goals as well as common social goals. Sociability distinguishes agents from services. When a service is aware of its environment (neighbour components, operating systems, etc.) and when it can act intelligently and proactively without external request, it should be considered as an agent.

Service-oriented and agent-oriented architectures are successfully contributing to the scalability, dynamics and usability of software systems. Many online applications (like Google, E-bay, Yahoo) now

offer services to millions of simultaneous users worldwide by using geographically distributed replicated servers while keeping the global state up-to-date and consistent.

However, to say the least, applications do not always perform without flaws. For example, it happened several times that a few millions Gmail users lost access to their messages. Other famous cases of software failures include the 2008 Amazon outage, or the 1996 Ariane 5 rocket explosion (LIONS, J.L. and al., et, 1996)(LEVESON, N.G., 2004). Systems do break down because of plain programming errors, but failures often relate to "softer" reasons like, e.g., when networks links get overloaded beyond expected capacities, when program code is reused in a manner unsuitable for new situations, or when exceptions are not properly handled.

Expectations that concern functions and services of the future system, i.e. questions about "*what*" the system should do, are called **functional requirements**. Expectations concerning "*how well*" (e.g. quality aspects of how speedily (performance), how cheaply (costs), how accurately, etc.) the system should realize what it is supposed to do are typically called **non-functional requirements** or **quality requirements**.

Multi-agent systems have to pay much attention to quality aspects, because the global goals of multi-agent systems are to be attained largely by collaborations among agents that are governed by their individual goals. A multi-agent system is a "society" where individuals are free, to a certain extent, to choose what they do. This is why, with functional or object-oriented systems, behaviour is more easily predictable than with multi-agent systems.

Intelligence, extensibility, and interoperability are often mentioned among the most important qualities offered by the agent-oriented software development paradigm (POSLAD, S. and Charlton, P., 2006). Still, obtaining such qualities for a software system always require a thorough analysis of requirements and a sound system design.

1.2. Requirements, goals and qualities

Requirements engineering plays an essential role in software development. Requirements are identified and captured from exchanges of information between stakeholders of the future system and software engineers.

In general, requirements are prescriptive statements to be enforced by the software-to-be, possibly in cooperation with other system components, and formulated in terms of environmental phenomena (VAN LAMSWEERDE, A., 2009). Requirements are analyzed to

provide a global view of the system in its early stages of development and then help construct a specification for system design in later stages. A good analysis of requirements is a necessary condition for a successful system development.

Requirement analysts play the role of architects who progressively transform the informal needs of stakeholders into a precise and possibly formal specification. While constructing software from design depends mostly on programming skills, designing software requires a thorough understanding of the environment and also a great deal of experience. Collecting and understanding stakeholder informal descriptions and performing the informal-to-precise-to-formal transformations are hard creative tasks.

Requirements engineering takes place in the context of human activities. It is therefore sensitive to human values and goals, and it relates to cognitive and social disciplines like cognitive psychology, anthropology, sociology, and linguistics (NUSEIBEH, B. and Easterbrook, S., 2000), that provide both theoretical grounding and practical techniques relevant to eliciting and modelling requirements.

Zave (ZAVE, P., 1997) provides an interesting definition of requirements engineering (RE):

Requirements engineering *"is the branch of software engineering concerned with the real-world goals for, function of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families".*

This definition identifies the principal ingredients of requirements engineering, namely, requirements, real-world goals, functions, constraints, relationships and evolutions. Real-world goals represent the motivations for building the system. Functions and constraints on the system represent important contextual concerns. The definition also positions the information system in its changing environment and it raises the issue of aligning system behaviour with the evolution of the surrounding real world.

Closely linked with requirements is the notion of goal. Goals are conditions or states of affairs in the world that the stakeholders would like the system to achieve. Goals are typically classified into hard-goals and soft-goals.

Hard-goals have satisfaction criteria that are clearly defined. Like functional requirements, they concern functionality (i.e., what the system should do).

Soft-goals are goals whose satisfaction criteria cannot be defined in a clear-cut manner and/or can be subjective in the sense that they may be judged as satisfied or unsatisfied to different degrees by different people. Simple examples are "Increase sales" or "Improve

payment confidentiality". Soft-goals relate to non-functional requirements.

Non-functional requirements are often also referred to as quality requirements in the literature without a clearly agreed-upon distinction. We adhere to that usage of the terms in our literature review in Chapter 2 and Chapter 3.

Starting from Chapter 4, we define "quality requirements" very differently. They are no longer similar to soft-goals, but they are defined as constraints on goals (hard-goals and soft-goals).

There have been many discussions and analyses of requirements in the literature, in terms of functional versus non-functional requirements, of hard-goals versus soft-goals, of non-functional and quality requirements versus soft-goals. In this thesis, we analyze and compare existing definitions for non-functional/quality requirements and for goal requirements.

This is to set the stage and to put our own proposals in context. We do not aim or pretend to sort out, and still less evaluate or rank those definitions. In particular, we do not discuss in any detail the nuances between non-functional requirements and quality requirements. Instead, after analyzing them, we present one definition of quality requirements and goals that we find particularly interesting, and we explore some of its interesting consequences.

1.3. Distinguishing quality requirements and soft-goals

In multi-agent systems, where goals are used extensively for modelling requirements, non-functional requirements are usually considered as a subset of soft-goals since there are similarities between non-functional requirements and soft-goals in their typical difficulty to precisely define satisfaction criteria.

To argue for a difference between soft-goals and non-functional requirements, it is sometimes suggested that non-functional requirements define constraints but do not define system functions, while soft-goals characterize states that the system should attain and thus can describe directly or indirectly system functions. But this is more intuition than an exploitable definition.

Consider some plausible examples of soft-goal requirements:

- All banking transactions must be treated in a secure manner.
- Online banking operations (money deposits, transfers, etc.) must be offered to customers with high availability.

Qualifying terms like "in a secure manner" or "high availability", represent qualities (of security and availability) that are embedded

inside the soft-goal statements. If those qualifying terms are taken out of the soft-goals, then the same situation can be described by a hard-goal (for example, “all banking transactions must be treated”) constrained by what we will call (starting in Chapter 4) a quality requirement (for example, “in a secure manner”, i.e., with quality security). In this case, we will say that quality requirements constrain goals, e.g., the quality requirement “in a secure manner” constrains the goal “all banking transactions must be treated”. We will argue that such a separation improves and simplifies the fulfilment of requirements in both functional and quality aspects.

Concerning the choice of term, using “quality requirement” for a constraint of a different nature than a non-functional requirement may not be the best idea. Just saying “quality” would not have been a better choice. We could maybe have chosen “quality constraint”. We stayed with “quality requirement” and tried to be as clear as possible throughout the text to avoid ambiguities.

General issues drawn from this discussion include the following:

- *Is it interesting to extract quality requirements from some soft-goals?*
- *If so, how would this benefit software development, especially of multi-agent systems?*
- *Which new tools could help software developers in the analysis?*

Although the above simple examples suggest that the answer to the first question could be affirmative, these issues deserve more analysis and we address them directly later in this thesis.

1.4. Our proposals and contributions

To treat quality requirements distinctly from soft-goals in the context of goal-based requirements engineering, it is necessary to formulate a definition of hard-goals, soft-goals, and quality requirements in such a way that they can be differentiated.

The coexistence of several approaches to define software quality has been well summarized in a recent book by Capers Jones (JONES, C., 2009). He argues that there are three principal ways to view software quality: i) conformance to requirements; ii) reliability, portability and other -ilities; and iii) absence of defects. Privileging quantitative methods for defect detection and removal using quality metrics, he prefers the third definition to the other two. For him, qualities like the -ilities are not practical because they are too vague (e.g., survivability) and most of them are useful but irrelevant for a users (like, e.g., portability). Problems with the first approach can arise if

some user requirements are badly selected, i.e. toxic, missing or excess requirements that unintentionally cause requirement-compliant products to go wrong.

Our approach is of course more related to the first definition (quality as conformance to requirements). As we will see the conformance to and effectiveness of requirements is strengthened by a double treatment: an ad-hoc qualitative analysis and decomposition of goals, and a quantitative analysis using social patterns for software quality control.

We propose, in Chapter 4, a definition of quality requirements where they are different from soft-goals, and they constrain hard-goals and soft-goals.

Note that some soft-goals are strongly integrated with quality requirements in the early formulation of requirements, e.g., “Message confidentially sent”. They could be called “quality soft-goals” and quality requirements can be elicited or extracted from them, e.g., “Confidentiality”. Other soft-goals are not thus explicitly linked with quality requirements, e.g., “Increase sales” or “Make customers happier”. However, their sub-soft-goals in goal analysis may be quality soft-goals, e.g., “Reliable connection provided”.

The proposed definition allows us to add quality requirement nodes and three new additional analyzing links to the usual AND/OR goal analysis. “Elicitation” links describe how quality requirements are extracted or inferred from some soft-goals. “Qualification” links describe which quality requirements have to be considered when hard-goals and soft-goals are achieved. “Contribution” links are used to describe how hard-goals can contribute (negatively or positively) to the satisfaction of quality requirements that constrain them. Quality requirements can themselves be refined into subqualities, and they can propagate down in the goal decomposition tree. A technique for resolving quality conflicts is also presented that makes use of those links between qualities and goals.

The objectives of this qualitative analysis of goals are:

- to accompany developers and stakeholders to identify relevant requirements (*clarification of requirements*);
- to propagate the responsibility of satisfying quality requirements, typically elicited from soft-goals, downwards to the suitable goals of the decomposition tree (*quality elicitation and propagation*);
- to fulfill qualified hard-goals by ad-hoc techniques, i.e. operationalization, when possible, e.g., use “Message encrypted” to fulfill “Message sent” constrained by “High confidentiality” (*quality operationalization*).

For qualified hard-goals that cannot be satisfied through goal decomposition, we introduce a set of design patterns for multi-agent system, i.e., social patterns in Chapter 5, that help software designers create a quality control subsystem. Quantitative techniques can be used to *control software quality*, to alert potential defects and to eventually remove defects.

Of course, these new facilities in our approach, i.e., requirement clarification, quality elicitation, propagation and operationalization, quality control at runtime, etc. do not automatically guarantee the success of software development. They only help developers to explore hidden requirements, to trace the consideration of quality, to delimit influenced zones of quality requirements, to give hints about defect localisation, to create quality-oriented design patterns. For example, the CASE (Computer-Aided Software Engineering) tool constructed for managing quality requirement, namely QCASE in Chapter 7, allows developers to automatically update the status of quality considerations during the analysis. A quality requirement is said "fully considered" if and only if it is properly propagated and/or operationalized.

We apply the new goal decomposition techniques and the proposed social patterns to the Tropos methodology to create Quality-aware Tropos, or QTropos in Chapter 6.

We validate the whole idea in a substantial case study in Chapter 8.

1.5. Text Outline

In addition to this introductory chapter, this document comprises 8 chapters and 3 appendices

Chapter 2 (Requirements Engineering) begins with a brief summary of the main concerns in requirements engineering. Then a short introduction of functional and non-functional requirements is given before a review of various definitions of hard-goals and soft-goals in the literature. Existing tools for analysing hard-goals and soft-goals are also presented.

Chapter 3 (Qualities and Non-Functional Requirements) discusses various issues about non-functional requirements such as: definitions, existing classifications, the scope of non-functional requirements, the measurability and the fulfilment of non-functional requirements. We also insist on important qualities of multi-agent systems (like flexibility and interoperability).

Those two introductory chapters (Chapters 2 and 3) do not contain original scientific contributions. They reflect our readings and comprehension of a difficult body of literature, where there is far more disagreement than agreement among researchers and

practitioners. They serve to set the stage and put our own contribution in perspective.

Chapter 4 (Goals and Quality Requirements) formulates the definition of our quality requirement concept, that is central to the thesis. Several aspects of the usual goal analysis process are revised and discussed in order to accommodate the presence of quality requirements that constrain goals and soft-goals.

Chapter 5 (Social Patterns for Quality Control) presents a catalogue of social patterns that help software developers in the design of multi-agent systems. With those patterns, designers can create a sub-system that controls the fulfilment of quality requirements by using measurements obtained from various parts of the system.

Chapter 6 (QTropos – Quality Aware Tropos) proposes enhancements to the Tropos methodology (Tropos is introduced in Appendix A). Using our new analysis of goals presented in Chapter 4 as well as tools developed in Chapter 7, we propose QTropos (quality-aware Tropos) where quality requirements are considered as independent notions constraining dependencies. With our proposals, quality requirements can be taken into account during all the development phases of the revised Tropos methodology: early requirements, late requirements, architectural design, and detailed design.

Chapter 7 (QCase – Tools for Quality Requirements) describes a tool built to support the developments in earlier chapters for quality requirements. The design is kept as flexible as possible so that quality requirements are decoupled from other elements to facilitate their inclusion in various methodologies, as done with two examples: general goal analysis with quality requirements and Tropos.

Chapter 8 (Print Service Case Study) presents a case study that consists of building a Print Manager in which the requirement of availability is one of the main quality requirements that make customers satisfied.

Chapter 9 (Conclusions) concludes and summarizes the presentation.

Appendix A (The Tropos Methodology) presents an introduction to Tropos, an example of multi-agent method that is used in Chapter 6 as a vehicle for our proposals for quality management.

Appendix B (Classifications of Quality Requirement) briefly illustrates some existing classifications of non-functional and quality requirements.

Appendix C (Software Development Process) surveys important issues about software development from the modeling concepts, principal development stages, development lifecycles, etc., insisting on software development methods for multi-agent systems.

Chapter 2 REQUIREMENTS ENGINEERING

This chapter starts with a general overview of requirements engineering in Section 2.1. Important topics that influence our later developments are the formulation of "the fundamental question of requirements engineering" by Zave and Jackson, and its proposed revision by Jureta et al., presented in Section 2.2, and the basic modelling concepts for requirements engineering. We present a short synthesis on functional and non-functional requirements in Section 2.3 and a review of goal-oriented requirements engineering in Section 2.4. Non-functional requirements and quality requirements are further explored in Chapter 3. These literature reviews in Chapter 2 and 3 help us put in perspective our main proposals formulated in Chapter 4.

Requirements engineering interacts with other phases of the software development process, notably with the design phase. Here we do not take into account how those interactions happen in different types of process. We include, as Appendix C, an overview of the most popular software development processes and of inter-phase interactions.

2.1. Overview of Requirements Engineering

Requirements engineering is the first phase of the software development process. In that phase, software developers collect and analyze needs and wishes from stakeholders, i.e. software users and partners, in order to create the first description document of the future system. This document describes "the contextual goals why a software is needed, the functionalities the software has to accomplish to achieve those goals, and the constraints restricting how the software accomplishing those functions is to be designed and

implemented" (VAN LAMSWEERDE, A., 2000) and (ZAVE, P. and Jackson, M., 1997).

Requirements engineering involves not only understanding the needs of stakeholders and the context in which the system-to-be will be used, but also modelling, analyzing, negotiating and documenting stakeholder requirements as well as managing requirements evolution. Five types of activities in requirements engineering have been suggested: elicitation, modelling, analysis, validation, and verification (CHENG, B. and Atlee, J., 2009). Correctly carrying out these activities helps to protect software projects from three leading sources of project difficulty: lack of user input, incomplete requirements, and changing specifications (HANSEN, S. et al., 2009). The main activities in requirements engineering are carried out in various ways according to how the fundamental question of requirements engineering is formulated.

Zave and Jackson (ZAVE, P. and Jackson, M., 1997) states that the main question of requirements engineering is how to find system specifications that satisfy input requirements of stakeholders while respecting common understanding of the applied domain. A refined way to formulate the question was introduced by (JURETA, I. et al., 2009)(JURETA, I. et al., 2008), in which they point out the limits of Zave and Jackson view, especially in treating non-functional requirements such as security, accuracy, performance, etc. We summarize the essential ideas of these approaches in the next section.

Complementary to the formulation problem, practitioners and researchers in requirements engineering have developed modelling languages to represent stakeholder needs and to communicate them to subsequent activities.

Requirements have been typically classified into functional requirements and non-functional or quality requirements. A more modern view of requirements engineering presents the system-to-be as able to bring about desired states of affairs, specified as goals. Goals are typically classified as hard-goals or soft-goals depending on the precision of the satisfaction criteria. Some properties are shared between functional requirements and hard-goals, and between non-functional/quality requirements and soft-goals, but those concepts have not been analyzed precisely enough in the literature so that they could be considered equivalent.

2.2. The Fundamental Question of Requirements Engineering

The importance of requirements engineering is now undisputed in software engineering. However, it took a long time before Zave and Jackson (ZAVE, P. and Jackson, M., 1997) formally formulated the

fundamental question of requirements engineering. They suggested that requirements engineering deals with a trio: ***K*** is the relevant domain knowledge containing the set of relevant *indicative*[1] properties; ***R*** is the set of requirements for a project containing the set of *optative*[2] properties whose satisfaction will fully satisfy the customer; and ***S*** is the set of specifications also containing the set of optative properties but, in principle, implementable. The main task for the requirement engineer is to find the specifications in ***S*** that satisfy the requirements in ***R*** given the domain knowledge in ***K***. Formally, the problem is solved if ***S*** is found such that ***S***, ***K*** $\vdash$ ***R***.

To arrive at this formulation, Zave and Jackson view the future system in the intended environment and divide actions into *machine-controlled* and *environment-controlled*. There can also be *shared* actions in which both the machine and its surrounding environment participate. *Unshared* actions are then those that are private to either the machine or its environment but not both. Based on these properties of actions, they classify any statement of action into requirements, domain knowledge (domain assumptions), and system specifications.

Arguing that the three concepts of requirements, domain assumptions, and specifications do not cover all the necessary concerns of requirements engineering, the work (JURETA, I. et al., 2008) uses the theory of the speech acts (SEARLE, J.R., 1969) to analyze the process of requirements engineering as a communication between stakeholders and software engineers.

They thus distinguish four modalities that correspond to the mental states underlying the speech acts: desires (leading to requirements), beliefs (leading to domain assumptions), intentions (leading to specifications) and attitudes (corresponding to evaluations and preferences) which they argue, are missing in the ontology of Zave and Jackson.

They reconsider the "requirements engineering problem" as formulated in (ZAVE, P. and Jackson, M., 1997) by introducing option selection procedures into each of the concepts of *domain assumptions* ***K***, *requirements* ***R*** (goals ***G*** and quality constraints ***Q***), and *specifications* (plans ***P***). To ease the selection procedure, the *attitude* notion ***A*** is added to represent preferences among options of each considered element. Soft-goals are approximated by quality constraints that are well-defined and correlated with the soft-goals.

---

[1] *Indicative* statements represent things that are currently true and will be true regardless of the presence of the system.

[2] *Optative* statements represent things that the presence of the new system is expected to make true.

The "requirements engineering problem" is reformulated as finding the best option of domain assumptions ***K****, the best option for specification ***S**** in order to satisfy the best option for requirements ***R**** (***G****, ***Q****) and attitudes ***A***.

In logical notation, one can write ***P****, ***K**** $\vdash$ ***G****, ***Q****, ***A***, where the consequence relation $\vdash$ is replaced by a *defeasible* consequence relation $\mid\sim$ representing a greater dynamics of this new view on requirements engineering.

That work distinguishes soft-goals and quality constraints. Soft-goals are used to model ill-defined and abstract conditions while quality constraints are used to define well-defined and concrete constraints.

Adding alternatives to each element of requirement helps requirement engineers look at the requirement problem with a broader view in a more structured way. However this also implies a greater complexity for requirement analysis. The practicability of this newly proposed view and its related techniques should be demonstrated through its relevance to real applications.

2.3. Functional and Non-Functional Requirements

In the early stages of the history of software engineering, the essential concerns were about the functions that the software system is expected to perform. Then, gradually, with consistent reports about software development risks such as uncontrolled budgets and development times (ROPPONEN, J. and Lyytinen, K., 2000), as well as about plain failures (CNET, 2008)(PCWORLD, 2009)(LIONS, J.L. and al., et, 1996)(LEVESON, N.G., 2004), the focus has been shifting to non-functional or quality aspects that constrain the way a system should carry out its functions. It is widely accepted that requirements are basically divided into two categories, functional and non-functional. This division is sometimes referred to as behavioural versus non-behavioural requirements (DAVIS, A.M., 1993).

As introduced in Section 1.1, functional requirements describe "*what*" the future system has to provide to its users, i.e., functions and services. Non-functional or quality requirements focus on *"how well"* the future system has to carry out its functions and services, interms of some specific aspects like availability, security, or precision, etc. Non-functional requirements concern both functions of the future system and the development process of that system.

Among many definitions of functional and non-functional requirements, Definition 1 can be considered as a good starting point for this introductory section. More definitions of non-functional and quality requirements are given in Chapter 3.

Definition 1: (SOMMERVILLE, I., 2007)

> ***Functional requirements****: are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.*
>
> ***Non-functional requirements****: are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the systems as a whole. They do not usually just apply to individual system features or services.*

The following are some examples of functional requirements describing functions of an Automatic Teller Machine.

- (Money Withdrawal) Automatic Teller Machines (ATM) must verify that there is enough money left for a withdrawal request before asking the customer for a confirmation code.
- (Language Configuration) Customers have at least three language options (English, French and Chinese) and they can switch at any time.
- (Card Blocking) A banking card that has not been retaken by its holder 10 minutes after the last manipulation must be kept inside the ATM and a notification message must be sent to the central management system.

The following are some examples of non-functional requirements that could be desired for the ATM example:

- (Confidentiality) The numerical keypad for entering secret codes must be visible only to the current customer.
- (User Friendliness) The graphical interface must be clear, understandable and intuitive.
- (Security) Transactions between terminals and the central management system must be secure.

If all functional requirements are implemented, the system should be able to work in some basic mode of operation. Handling non-functional aspects will enhance the satisfaction of stakeholder expectations. For example, if "promptness" is an important non-functional requirement, then of two systems that do the same functional job, the one that accomplishes the job in less time, all other things being equal will normally be chosen.

In software engineering research, the absence of non-functional requirements or quality requirements (such as *security, accuracy, availability,* etc.) have made the Zave and Jackson's view on the fundamental question of requirements engineering insufficient for

the recent developments. In fact, many works, for example (CHUNG, L. et al., 2000), acknowledge the importance of non-functional requirements. However, the satisfaction criteria of non-functional requirements are often "soft", meaning that those requirements often cannot be formalized. Non-functional requirements are considered as similar to soft-goals whose satisfaction criteria cannot be described in a clear-cut way and that were left out of Zave and Jackson's formulation.

The role of non-functional requirements has been demonstrated in practice in many projects. An inadequate consideration of quality aspects can cause fatal failures to the system, especially to those that work under safety-critical conditions. A famous example is the failure that led to the explosion of the first prototype of the Ariane 5 rocket in 1996 (LIONS, J.L. and al., et, 1996). The inspecting board after the failure concluded that poor software engineering practice was the main cause. Much research, for example (CLELAND-HUANG, J. et al., 2005) and (LEVESON, N.G., 2004), claim that the cause of the failure was the incorrect implementation and management of non-functional requirements. The trade-off between cost and safety was not properly addressed during specification and design, which created fatal overflow exceptions, because of unwarranted reuse of existing code.

Functional and non-functional requirements are suitable for modelling simple systems in which requirements are simple enough to deal with. For larger systems with modern technology, finer approaches are needed, because:

i. Directly modelling stakeholder needs by descriptions of functions in a complex system is can be difficult. It is not trivial for developers to figure out all the functions of the system-to-be directly from business intentions. Transition steps are needed to avoid possible gaps between initial ideas and final product.

ii. Software systems have become more autonomous, interoperative and intelligent. In addition to performing functions, systems may be required, without human intervention, to search for the best service provider, to perceive and analyze the environment, to reason about the situation, and to select the best option for action. Functional requirements and non-functional requirements, which favour statically built systems, are no longer sufficient.

Hard-goals and soft-goals are now generally considered a more adequate approach to model stakeholder desires and system objectives.

2.4. Hard-goals and Soft-goals

A fruitful approach to elicit and to analyze requirements is that of goal-oriented requirements engineering where requirements are considered as *goals* (DARIMONT, R. et al., 1997). Broadly speaking, a goal is a state of affairs that the system-to-be is expected to achieve. A hard-goal is a goal whose desired state can be exactly defined and completely fulfilled. A soft-goal is a goal whose desired state cannot be defined in a clear-cut way and/or whose fulfilment can be subjective in the sense that it may be judged as satisfied or unsatisfied to different degrees by different people. Thus, hard-goals can be satisfied, while soft-goals can only be "satisficed" (CHUNG, L. et al., 2000). The term "satisficing" is believed to have been introduced in (SIMON, H.A., 1955). Satisficing a soft-goal is a weaker notion of satisfaction where goals are satisfied to a certain extent or whose satisfaction is subject to subjective judgement.

Goals were introduced to fill the gap between real-world objectives and operations of the system (CASTRO, J. et al., 2002). High-level goals are global objectives stated early by the stakeholders about what they expect the system to bring about. High-level goals are often those of the ongoing business. For example, for a retail shop, an important goal is to *maximize revenue*.

There are similarities between functional requirements and hard-goals, and also between non-functional requirements and soft-goals. Indeed, it is typically difficult to precisely define the satisfaction criteria of non-functional requirements, which makes them similar to soft-goals.

2.4.1. Existing definitions of goals

We review here a nonexhaustive list of existing definitions of goals, hard-goals, and soft-goals. To construct this list, we started from the study in (JURETA, I.J. et al., 2007) and added some more recent definitions with nuances between the three concepts.

i*, GRL (LIU, L. and Yu, E., 2001) TROPOS(CASTRO, J. et al., 2002)

> *"A hard-goal is a condition or state of affairs in the world that the stakeholders would like to achieve. In general, how the goal is to be achieved is not specified, allowing alternatives to be considered."*
>
> *"A soft-goal is a condition or state of affairs in the world that the actor would like to achieve. But unlike a hard-goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated soft-goal"*

(VAN LAMSWEERDE, A., 2009)

"A goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents."

"Behavioural goals describe intended system behaviours declaratively. A behavioural goal can be established in a clear-cut sense."

"Soft-goals prescribe preferences among alternative system behaviours. A Soft-goal cannot be established in a clear-cut sense."

(DARDENNE, A. et al., 1993)

"A goal is a nonoperational objective to be achieved by the composite system. Nonoperational means that the objective is not formulated in terms of objects and actions available to some agent in the system; in other words, a goal as it is formulated cannot be established through appropriate state transitions under control of one of the agents."

"Abstract/informal goals need to be refined into concrete/formal ones."

Non-Functional Requirement framework - NFR (CHUNG, L. et al., 2000)

"A Goal [refer to] non-functional requirements, design decisions and arguments in support or against other goals."

(DONZELLI, P., 2004)

"According to the nature of a goal, a distinction is made between hard goals and soft goals. A goal is classified as hard when its achievement criterion is sharply defined"

" For a soft goal, instead, it is up to the goal originator, or to an agreement between the involved agents, to decide when the goal is considered to have been achieved"

"In comparison to hard goals, soft goals can be highly subjective and strictly related to a particular context; they enable the analysts to highlight quality issues [...] from the outset [...]"

Goal-Driven Change – GDC (KAVAKLI, E., 1999)

"An enterprise goal is a desired state of affairs that needs to be attained."

Goal-Based Requirements Analysis Method – GBRAM (ANTÓN, A., 1996)

"Goals are high level objectives of the business, organization or system. They capture the reasons why a system is needed and guide decisions at various levels within the enterprise."

Lightswitch (REGEV, G. and Wegmann, A., 2005) *"A goal corresponds to a state of a system in the underlying principles that is specified by an agent"* *"A soft-goal is defined as a goal that has no clear-cut criteria for achievement."*
(SOFFER, P. and Wand, Y., 2005) *"The state of a thing is the set of values of all its attribute functions. A goal is a set of states that the domain can stay in indefinitely, unless, something in the environment causes it to change."* *"A soft-goal is an order relation on states."*
(JURETA, I. et al., 2009) *"Any instance of communicated information that is communicated via a directive speech act is an instance of goal."* *Quality constraint is a goal that refers to a quality whose quality type has an associated quality space with a structure that is shared among the participants in requirement engineering.* *Functional goal is a goal that refers to a perdurant (i.e., either event, state or process), and does not refer to a quality.* *Soft-goal is a goal that refers to a quality whose quality type has an associated quality space with a structure that is not shared among the participants in RE.*

Table 1: Definitions of goals, hard-goals and soft-goals

Goals are related to prescriptive statements of intent (VAN LAMSWEERDE, A., 2009), nonoperational objectives in (DARDENNE, A. et al., 1993), desired properties about quantities in the environment (LETIER, E., 2001), conditions or states of affairs in i*, GRL, TROPOS and GDC, high level objectives of business, organization or system GBRAM, communicated information via a directive speech act (JURETA, I. et al., 2009), etc.

The term "states of affairs" seems to cover all variants of goal definitions, except for the last one, which relies on knowledge in communication theory (SEARLE, J.R., 1969) and DOLCE ontology (MASOLO, C. et al., 2003).

In NFR and GDC frameworks, the emphasis is on soft-goals. In KAOS as in (DARDENNE, A. et al., 1993), hard-goals and soft-goals are not explicitly mentioned; abstract goals and concrete goals are used instead.

Soft-goals (explicitly defined as preference in KAOS and as order relation in (SOFFER, P. and Wand, Y., 2005) are usually used to compare design options and system implementations.

2.4.2. Goal decompositions

In goal-based requirements engineering, one of the objectives is to transform high-level requirements (stakeholder needs) into lower-level requirements (concrete system requirements). AND/OR decomposition trees in KAOS (DARIMONT, R. and Van Lamsweerde, A., 1996) have become a standard tool for such task. AND/OR decomposition in KAOS is best used for hard-goals since hard-goals can be defined as logical conditions and the satisfaction of hard-goals can be determined by AND/OR relations between corresponding logical conditions of sub-hard-goals.

In an AND decomposition, satisfying all sub-goals is a sufficient condition for satisfying the goal. In an OR decomposition, satisfying one of the refinements is a sufficient condition for satisfying the goal.

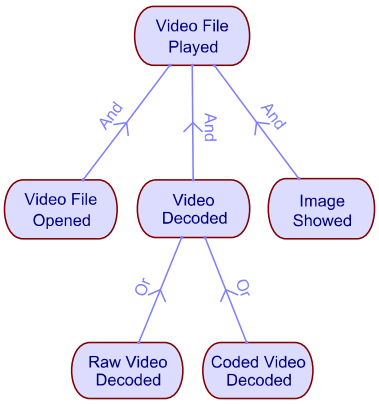


Figure 2-1: AND/OR decomposition tree

Figure 2-1 is an example of the decomposition tree of the hard-goal “Video File Played”. To satisfy that hard-goal, sub-hard-goals “Video File Opened”, “Video Decoded” and “Image Showed” must be satisfied. There are two alternatives to satisfy the hard-goal “Video-Decoded” in accordance to two types of video files: “Raw Video Decoded” and “Coded Video Decoded”.

For soft-goals, weaker types of goal transformation are introduced in (CHUNG, L. et al., 2000), namely AND/OR contributions. Derived soft-goals can contribute, negatively or positively, to fulfilling other parent soft-goals. Contribution links can take one of five contribution levels: ‘++’ (make), ‘+’ (help), ‘?’ (unknown), ‘-‘ (hurt) and ‘--‘ (break). These contributions types allow practitioners to determine the satisfaction level of each soft-goal, given the satisfaction level of the derived soft-goals and the type of the contributions (AND/OR).

The work in (JURETA, I. et al., 2009) argues that AND/OR decomposition is not sufficient to describe all possible refinements. It proposes a new type of decomposition, called j-refine (for justified-refine), in an interesting deeper ontology for requirements based on linguistic speech acts.

Other types of analysis links in TROPOS (CASTRO, J. et al., 2002) are described in Appendix A.

2.4.3. Why goals are attractive

Goals have been widely adopted in many different tasks of requirements engineering: acquiring requirements, relating requirements to organisational and business context, clarifying requirements, dealing with conflicts, driving design, etc. The reasons for their early popularity and current wide-spreading influence can be explained as follows.

First, according to their definition, goals can be used to describe both high-level business objectives and desired output of a particular function of the system-to-be. Moreover, they allow us to navigate through different levels of abstraction of requirements. Once an early set of goals has been attained and validated with stakeholders, WHY and HOW questions about them can be asked to identify other goals at higher-levels and lower-levels through abstraction and refinement (VAN LAMSWEERDE, A., 2002).

For example, why the soft-goal “Transfer Time Reduced” is interesting could be that it helps satisfy the soft-goal “Make Customers Happy”. Achieving “Transfer Time Reduced” could be addressed through a refined goal like “Data Compressed”.

Using goals can help reduce the gap between stakeholder desires and the outcomes of the system.

Since requirements are identified and analysed using goals, at different levels through a coherent and adequate goal decomposition of identified goals, it can be said that using goals helps bridge the gap between stakeholders desires and the outcome of system functions.

Second, goals can represent a wider range of requirements than functional or non-functional requirements. Every functional requirement, defining a function or a component of software system, such as “Users can turn on a printer” and “Users can print a document” can be reformulated as a hard-goal, i.e., as states “Printer turned on” and “Document printed”. Every non-functional requirement, as constraints on functions such as: “System should show intuitive messages” and “System should provide high quality printing” can be represented by means of soft-goals, i.e. “Intuitive

messages showed" and "High quality printing provided". However, soft-goals like "Market share increased", which are not related to constraints on functions, cannot be reformulated as non-functional requirements.

Third, conflicts between requirements can be negotiated and cleared by weighing the potential risks produced by each option. For example, the Non-Functional Requirement Framework (CHUNG, L. et al., 2000) provides a way to propagate the positive/negative effects upwards in the decomposition tree in order to evaluate the risk factor produced by an analysis option. The preferred option will be the one that presents the least risk.

Fourth, at the lowest level, all goals are operationalized (DARDENNE, A. et al., 1993). This results in the functional design of the system-to-be where hard-goals and soft-goals are re-expressed as operations to be performed by the future system.

The following are some examples of hard-goals in the Automatic Teller Machine (ATM) application.

- (Basic banking operations offered) Basic banking operations are offered to customers who hold a valid banking card.
- (ATMs controlled remotely) ATMs are controlled remotely by the central management system.
- (Logs recorded and transferred) All the operations at an ATM must be logged and log entries must be transferred to the central management system.

These examples prescribe some states that the future ATM system should bring about without details about how to realize them. High-level goals tend to formulate the *raison d'être* of the future system. This can be shown by the following examples of soft-goals for the same application:

- (Operating cost minimized) The operating cost of ATM system is minimized.
- (New customers attracted) New customers should be attracted by the new ATM system.
- (Fast, secure, accurate and reliable ATM) The ATM system should be easy to use and should respond promptly, while being secure, accurate, and reliable.

The third soft-goal is a prescription of qualities (usability, promptness, security, accuracy, and reliability) that ATMs will have to comply with when offering their services. This example illustrates the presence of qualities closely integrated in a soft-goal.

Some more examples of hard-goals and soft-goals in the Online Concert Ticket Booth are the following:

- (Seat reserved if available) Customers will be able to reserve seats for any available concert if the requested number of seats of the requested category is available.
- (Paid by money transfer) Payments can be made by bank money transfer.
- (Reservation status monitored) Reservation status can be monitored when a deferred payment mode is chosen.
- (Secure online payment) Online payments must be secure.
- (System availability) The system must be kept available for a very large number of accesses.
- (All available concerts listed) Any available concerts must be easily accessible from the interface.

The first three statements above are hard-goals and the last three are soft-goals. The previous examples of soft-goals are tightly coupled with some qualities (security, availability, accessibility).

We will come back to the relationships between qualities and soft-goals in Chapter 4, where we define quality requirements as an independent notion, alongside hard-goals and soft-goals, that can constrain hard-goals and soft-goals. This will help us to explore some implicit links between these three notions.

Chapter 3 QUALITIES AND NON-FUNCTIONAL REQUIREMENTS

This chapter reviews some definitions of non-functional requirements and quality requirements to set the stage for the introduction of our own concept of quality requirements in Chapter 4.

Section 3.1 is an overview of non-functional requirements that briefly recalls why this type of requirements is difficult to address. Among issues pointed out in this section, the *definition problem* is maybe the most important and it is further investigated in Section 3.2 through a literature study.

Before moving to practical issues about non-functional requirements, we narrow our discussion on our principal field of application, namely, multi-agent systems. In Section 3.3, questions addressed include why quality aspects are crucial to such type of systems and in which scope they must be addressed.

For multi-agent systems, we address the measurability of non-functional requirements in Section 3.4 and the fulfilment of non-functional requirements in Section 3.5.

3.1. Overview of Non-Functional Requirements

According to (CHUNG, L. and Leite, J.C.P., 2009), most existing treatments of requirements still lack a proper handling of quality characteristics.

The following are some reasons that make analyzing non-functional and quality requirements particularly difficult. Some of those are taken from (CHUNG, L. et al., 2000).

- Requirements can evolve during their analysis through interactions between developers and stakeholders. Important

non-functional requirements may remain hidden and missing from the final system. “Toxic” requirements (like the source of the Y2K problem according to (JONES, C., 2009), “a legitimate but inadequate requirement) and superfluous non-functional requirements may not be identified as such and may not be removed.

- Non-functional requirements are often abstract and not defined in a clear-cut manner.
- It is often difficult to verify the fulfilment of non-functional requirements.
- The definition and satisfaction of non-functional requirements can be subjective. Consensus may be hard to reach among developers and stakeholders, and even among developers themselves.
- Non-functional requirements sometimes conflict with one another. Tradeoffs are usually needed when considering two or more requirements at the same time.
- Non-functional requirements are not clearly related, in general, with specific components of the systems. Requiring the whole system to have a quality maybe very costly while requiring it from an insufficient set of parts may cause violations of the overall quality.

The pioneer work in (CHUNG, L. et al., 2000) provides a way to use non-functional requirements in the selection of design and implementation options in which contribution links are used to connect non-functional requirements to respective options. These links offer a graded measure to “compute” the fitness of each function with respect to its related non-functional requirements. Contribution links are graded through four different levels: *break, hurt, help* and *make.* The combination of all contribution links from each design alternative into non-functional requirements constitutes a basis for selecting the best option among the alternatives. However, there is no indicator of how to limit the number of design alternatives for an issue while it is usually impossible to list all such alternatives. Moreover, if there is only one alternative, it must be adopted irrespectively of the satisfaction or nonsatisfaction of non-functional requirements.

Impressively, while the literature essentially agrees on the definition of functional requirements, e.g., (SOMMERVILLE, I., 2007), see Definition 1, there is no consensus about a definition of non-functional requirements. A number of definitions are gathered in (GLINZ, M., 2007) that point out various divergences. This **definition problem** is addressed in Section 3.2 below. Other issues concern the representation and the classification of non-functional requirements.

As for the **representation problem**, the distinction between functional requirements and non-functional requirements is sometimes not sharp. Depending on the level of detail and on the progression through the development process, the same requirement can be represented as a functional requirement or as a non-functional requirement.

For example, the following requirement concerning the accuracy of exchange rates used by an automatic teller machine (ATM) can be formulated in various ways, like:

- The ATM should use the correct exchange rates.
- The ATM must check exchange rates from the central management system before each transaction takes place.

The second formulation is a functional requirement as an action to be performed by the system is mentioned, while the first one has a non-functional form as no action to satisfy the requirement is explicitly mentioned.

In fact, the second one can be considered as a correct operationalization (DARIMONT, R. et al., 1997) of the first one with concrete functions mentioned to do the work.

As for the **classification problem**, some classifications of quality requirements are presented in Appendix A. We did not attempt to compare those classifications, as it is a very difficult task that is out of the scope of this thesis.

3.2. Various Definitions of Non-Functional and of Quality Requirements

The reviews in (GLINZ, M., 2007) and (CHUNG, L. and Leite, J.C.P., 2009) show many variations in definitions of non-functional requirements.

Two basic ideas underlie most works about non-functional requirements:

- Non-functional requirement is used to refer to quality aspects such as “-ilities” (e.g. usability) or “-ities” (e.g., integrity) or some others (e.g., performance, user-friendliness, coherence).
- Non-functional requirement is used to refer to concerns not related to the functionality of the software.

In the context of functional/non-functional requirements, there are several confusions about those statements. Saying that functionality may be considered as a quality as pointed out in (CHUNG, L. and Leite, J.C.P., 2009) somehow contradicts the second statement. If functionality is not considered as a quality, then one can say that the

second statement describes a superset of that described by the first one, by possibly including architecture requirements, etc. However, this is rarely the case in the literature. Many authors relate non-functional requirements solely to quality aspects such as: usability, accuracy, security, etc. We do not enter further into the discussion about the relation between non-functional requirements and qualities in this chapter.

We give in the next table a list of various definitions of non-functional and quality requirements extracted from the literature, in particular from the two surveys in (GLINZ, M., 2007) and (CHUNG, L. and Leite, J.C.P., 2009) .

(ANTÓN, A., 1997) *"... describe the nonbehavioral aspects of a system, capturing the properties and constraints under which a system must operate."*
(DAVIS, A.M., 1993) *"The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability."*
(BOOCH, G. et al., 1999) *"A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement."*
(SOMMERVILLE, I., 2007) *"... constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Quality requirements often apply to the systems as a whole. They do not usually just apply to individual system features or services."*
(CHUNG, L. et al., 2000) *"... global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. (...) There is not a formal definition or a complete list of non-functional requirements."*
(NCUBE, C., 2000) *"... behavioral properties that the specified functions must have, such as performance, usability."*
(ROBERTSON, S. and Robertson, J., 1999) *"... property, or quality, that the product must have, such as an appearance, or a speed or accuracy property."*

(WIEGERS, K.E., 2003) *"… description of property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior."*
(GLINZ, M., 2007) *Requirement that " is an attribute of or a constraint on a system."*
(VAN LAMSWEERDE, A., 2002) *"… concerns associated with quality of service – such as safety, security, accuracy, performance, and so forth."*
(PAECH, B., and Kerkow, D., 2004) *"… requirements focusing on 'how good' software does something as opposed to the functional requirements, which focus on 'what' the software does."*
(LANDES, D. and Studer, R., 1995) *Requirements that "… constitute the justifications of design decisions and constrain the way in which the required functionality may be realized"*

Table 2: Definition of Non-Functional Requirement

These definitions demonstrate divergences about the supporting concepts for non-functional requirements: constraint, property, characteristic, attribute, quality, performance, etc. Worse, non-functional requirements can be defined as both non-behavioural aspects in (ANTÓN, A., 1997) and behavioral properties in (NCUBE, C., 2000). This reflects the subjectivity of non-functional requirements even in how to define the concept itself. Extended discussions in (CHUNG, L. and Leite, J.C.P., 2009) or (GLINZ, M., 2007) illustrate the difficulties involved.

Defining software quality

Quality has concerned many disciplines in management and engineering. According to ISO-9000 (ISO-9000, 2000), quality is the "degree to which a set of inherent characteristics fulfils requirements" and a requirement is a "need or expectation that is stated, generally implied or obligatory".

Most, if not all, researchers and practitioners in information systems and software engineering would agree that software quality is essential to successful software products and processes. But definitions are numerous in the literature, and a satisfactory general definition that would accommodate the various approaches is fairly difficult to formulate.

In his recent book, Capers Jones (JONES, C., 2009) highlights several popular definitions:

- "Quality means conformance to requirements".

  The major problem, for C. Jones, with that definition is that, in practice, requirements are never complete nor fully correct. Requirements do change with time, as stakeholders interact with developers. Quality thus defined is not predictable during development nor is it fully apparent to users.

- "Quality means reliability, portability and many other -ilities".

  According to C. Jones, the main problem with a list of -ility factors is that they are not predictable before they occur and often they are not easily measurable; another problem is that different factors concern different stakeholders: some, like reliability or testability, concern quality as viewed by users; other like portability or maintainability are of interest to vendors or development teams.

  This approach seems typical of standards organizations and of information-system researchers who take a normative position in front of the poor performance of the software industry in handling quality.

  Appendix A briefly presents some such classifications.

- "Quality is the absence of defects that would cause an application to stop working or to produce incorrect results, combined with topics like ease of customer support and maintenance speed"".

  This approach focuses on minimizing the defects to improve the adequacy and reliability of software. The following sentence by C. Jones summarizes the general idea: "Measuring defect volumes and defect severity levels, and then taking effective steps to reduce those volumes via a combination of defect prevention and defect removal activities is the key to successful engineering".

  The approach is typical of an active community producing an abundant literature on measurement and metrics for estimating quality aspects during both the development process and the operation of the final product.

  Techniques described in this quantitative direction can help software systems to satisfy non-functional requirements that cannot be satisfied or satificed by requirements analysis or

system design. These not-yet-fulfilled non-functional requirements can be measured and taken care of at runtime. This can be considered as part of the "operationalization" of non-functional requirements, at later stages after requirements engineering.

Our work is of course more related to the first definition (quality as conformance to requirements). The other two definitions shed light on the complexity and multi-dimensionality of quality: conformance to requirements like satisfying –ility factors of course correlate with the presence of defects and also with the absence of defects.

Related to this discussion of quality definition, Section 3.4 discusses the measurability of qualities, Chapter 5 proposes several design patterns for multi-agent systems to control quality at runtime and these patterns are applied to the detailed design stage of QTropos in Chapter 6 and partly implemented in our case study in Chapter 8.

3.3. Non-Functional Requirements in Multi-Agent Systems

Multi-agent systems are characterized by autonomous, proactive (and sometimes intelligent) agents that are loosely coupled (ODELL, J., 2000). They exchange messages encoded under commonly accepted *format* (structure of message) and *ontology* (taxonomy of communicated knowledge). This new structure offers both new capabilities and new quality concerns to software system. This section describes two issues about qualities in multi-agent systems that influence on the sequel of the thesis.

Based on the special characteristics of multi-agent systems, i.e. variable population scateredness, possible incompatibility of ontology, typical asynchronous collaboration and exposed vulnerability to attacks, Section 3.3.1 gives some reasons why non-functional requirements must receive more attention in multi-agent systems. Section 3.3.2 places non-functional requirements in components at different levels of multi-agent systems. Understanding those issues helps developers to predict possible consequences when some qualities are required.

3.3.1. Why non-functional requirements for agents

As pointed briefly in Chapter 1, a multi-agent system can be considered as a virtual society where software agents live. As in a human society, the overall behaviour depends on atomic behaviours of its individuals. To make any society evolve to a better state, individuals are not only constrained in what they can do but also in how they carry out their actions.

In a multi-agent society, agents are given the capabilities to collect information, to reason and to react to environmental changes to attain individual and social objectives. Unlike in traditional more static systems, where software components and connections between them are largely predefined at design time, in multi-agent systems, agents can be freely added and removed dynamically.

Moreover, connections between agents are often based on web protocols that are loose. Agents may not know whether a connection has been lost. Connections often exist only during a transaction and agents have to re-establish necessary connections before starting a transaction. It can happen that, when an agent needs to have a service done, it cannot find any service provider. This shows a risk of low availability.

Other pitfalls about software development with agents are discussed in (WOOLDRIDGE, M.J. and Jennings, N.R., 2002), which arise from difficulties about handling security, compatibility, availability, robustness, etc. Here we list some possible sources of problems that argue for the importance of carefully addressing non-functional requirements for agent systems.

i) Population Scatteredness

In multi-agent systems with a small population of agents, it is rather simple to keep track of all the active agents. This is even easier if all the agents operate inside a single environment.

In larger systems, agents usually spread across many hosting machines on many different platforms. A typical example is a peer-to-peer (P2P) multi-agent system with no structural distribution of agents. The system is composed of a very large number of agents that are totally decentralized on the internet. Several non-functional issues may arise due to the scattering of agents:

- *Availability*: the presence of agents may be undetected. Connections between agents may be lost. The population can be split into unconnected clusters if connecting links are lost.
- *Identifiability*: possible loss of agent identity, insufficient identity led to routing problems, etc.
- *Data integrity*: when data is stored in more than one source, data may become inaccessible, and become incomplete and inconsistent.

ii) Incompatibility of communicated contents

Incompatibility of ontology implies disagreement about knowledge representation between agents that take part in a communication.

Ontologies are used by agents to organize and coordinate exchanged contents.

Incompatibility of message format implies the disagreement of how communicated contents are structured in a message.

Without a commonly accepted ontology and structure, messages may not be correctly understood by receiving agents.

Since multi-agent systems depend largely on the collaboration of loosely coupled elements, communication plays a central role. Therefore, when exchanged messages are ignored or misunderstood by agents, the whole system can break down very easily.

iii) Asynchronous Collaboration

Intelligent agents can autonomously react to events that they perceive by reasoning about the actual situation using their internal states and measurements on their surroundings. However, each agent runs on its own thread whose internal states may not be observed by other agents.

When an agent perceives an event or receives a request to which it has to react, it may not be able to do so immediately. Instead, a waiting list of events or jobs is usually implemented to keep track of incoming information while the agent is busy. This may cause a problem of *responsiveness* of the system if an agent becomes a bottleneck for the whole system.

iv) Vulnerability to Attacks

In order to make multi-agent system highly extendable, interoperating and flexible, agents can accept different communication protocols. Moreover, unlike in traditional systems where components are more statically linked, in multi-agent systems, all inter-agent links dynamical. Such links between agents are more vulnerable to attacks than those in more static systems.

Relevant non-functional/quality aspects include trust, validity, encryption scheme, digital signatures and certificates, etc. in order to protect against infiltration attempts by malicious agents. *Security* problems in multi-agent systems are one of the major reasons why multi-agent schemes have not been more widely used in commercial products.

3.3.2. Scope of Non-functional Requirements

As we have seen, non-functional requirements define the users' expectation in terms of the quality of services offered by the system.

Non-functional requirements can be analyzed along the temporal scope:

- Development-time requirements like reusability, extensibility, scalability, usually influence the static structure of the system and the development methods.
- Run-time requirements, like availability, security, usability, influence the dynamics of the system during it operation.

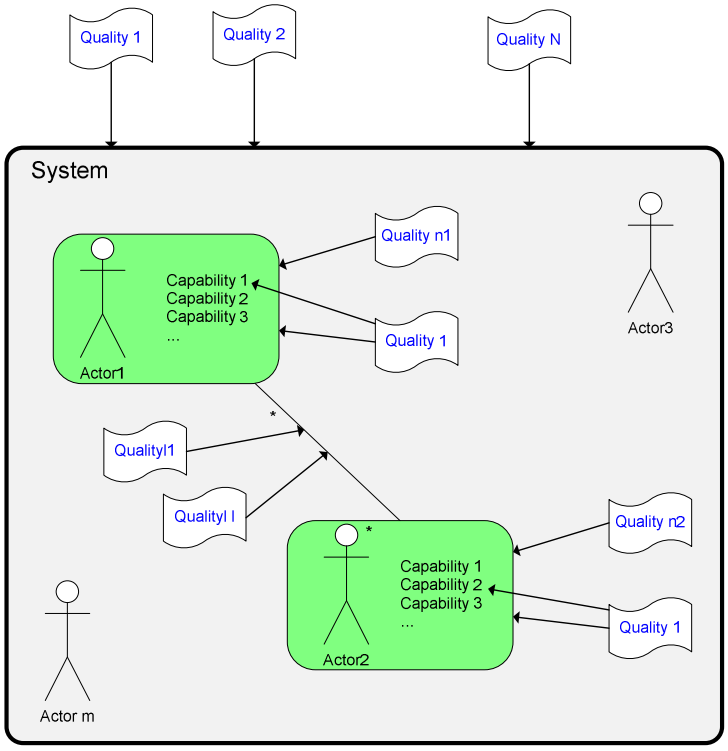


Figure 3-1: Quality requirement of multi-agent system

Non-functional requirements are usually referred to as system qualities or global constraints on the system. Nevertheless, one can identify all the parts of the system that concern a specific non-functional requirement. This defines the physical scope of a non-functional requirement.

Each agent has a set of capabilities and a system has a set of agents. But this does not mean that if all the agents (or capabilities) fulfil sufficiently a non-functional requirement, then the whole system (or the agent) will fulfil that requirement as well. A non-functional requirement, that an agent (or a system) is constrained to have, need not be necessarily fulfilled by all the capabilities of that agent (or by all the agents of that system). The fulfilment of a non-functional

requirement in a multi-agent system depends not only on the system composition but also on agent interactions and organization.

It may happen that a system can fulfil a non-functional requirement when only some of its agents sufficiently fulfil certain required qualities and when those agents have appropriate interactions inside the system and/or to the outside world. This is also true at agent level, where some required capabilities must be implemented to react in a suitable manner to a range of changes in its environment to help the agent fulfil a non-functional requirement.

Therefore, the scope of qualities must be carefully taken into account to determine the optimal level at which a non-functional requirement constrain: system level, agent level or capacity level (see Figure 3-1).

a. Non-functional requirements at system level

Some non-functional requirements are required for the whole system. They are those that the whole final product should sufficiently fulfil to guarantee that the system will work correctly.

For example, an online auction system has to satisfy the *portability*, that is, it must be compact and must executable on several platforms. A web-based application must have *consistent* behaviour given sufficiently similar environment and conditions. *Testability*, which is the possibility to easily check the operation of the system, is also required at the system level.

Another example of system quality is the *interoperability*, one of the most important qualities of an agent-based application. To easily accept the operation of new agents, the whole system must be highly *adaptable* to the newcomers, possibly written for several different agent platforms and deployed on different machines across the network.

Often, in the literature, non-functional requirements are often required only as global level of the system. However, this is not adequate, in general, for multi-agent systems, which are highly dynamic and loosely coupled. We should express non-functional requirements at finer levels: agent level or capability level.

b. Non-functional requirements at agent level

Non-functional requirements, required for agents, help developers to choose the right design parameters such that the newly created agent can operate harmoniously with already existing ones. At run time, required qualities influence how agents reason about their knowledge and react to changes in the environment to meet their goals, to help

others realize their goals and to contribute to the fulfilment of collective goals.

Each agent in the system has to fulfil some specific non-functional requirements to reach their individual and/or system goals more efficiently.

For example, in an online auction system, the Account Manager agent must be required to be *reliable* and *responsible* to the accounting system. In particular, the agent should not communicate any accounting information to agents that are not concerned.

c. Non-functional requirements at capability level

Each agent has a set of capabilities. Capabilities capture the intended behaviours of agents in the system. They can also be required to satisfy some non-functional requirements that influence how capabilities will be carried out.

For example, in an online auction system, it is required that when realizing the function "Describe item", the description of item must be *comprehensive*.

A non-functional requirement, initially analyzed as constraint on an agent, can be fulfilled by some of its capabilities.

The operation of the system is realized through its functions. In the view of multi-agent system, the operation of the whole system is broken down into capabilities of participating agents. When building such system, every non-functional requirement at the system level will be sooner or later decomposed and delegated or assigned to agents. Without the delegation and assignment process, non-functional requirements will never be sufficiently fulfilled. The delegation and assignment process should be also applied to agent level so that every non-functional requirement can be guaranteed by agent capabilities.

3.4. Measurability of Quality

Non-functional requirements are the demands about software quality. Reviewing on the quality for general products, a deep survey in (REEVES, C.A. and Bednar, D.A., 1994) compares several different visions about product quality, including: Quality is Excellence and Quality is Value. Being excellence, it is universally recognizable but has measurement difficulties. Being value, it allows for comparisons across disparate objects and experiences but it has difficulties in extracting individual components of value judgment. They also point out that quality and value are different constructs and that by defining quality as other than excellence, quality may lose its

meaning. But they also state that articulating precisely what excellence is may be impossible.

In software, both qualitative (as excellence) and quantitative (as value) approaches are applied for software quality.

Measurements of software quality are done using *metrics* (KAN, S.H., 2002). They provide analysts with quantified quality indicators that can be used to compare different systems or different system configurations. Examining the weaknesses of viewing quality as value, we argue that we cannot classify qualities into just two classes: *exactly* measureable and *not-at-all* measurable. Furthermore, as reconfirmed by (CHUNG, L. et al., 2000) that software qualities (as excellence) are usually abstract, relative and subjective, almost all of them cannot be exactly measurable.

In terms of project management, measurability maybe relative in the sense that a software quality sometimes depends on the implementation choices of a project, on the cost of measurement, on the tools available, etc. Given the available resources for the project, software designers must decide, for each quality, whether it will be measured completely, heuristically, or partly, or whether it is immeasurable at all. We base on this resource-dependant characteristic and on the intrinsic difficulty of quality to describe, in the followings, the four types of qualities in a project: fully measured, partly measured, heuristically measured and not unmeasured.

Note that measurements alone do not say whether a non-functional requirement is satisficed. Fuzzy satisfaction criteria of non-functional requirements (e.g., promptness) must be approximated by concrete and quantified criteria, (e.g. less than 5 seconds).

3.4.1. Fully measured quality

To be exactly measured, quality must be able to be exactly and completely represented by numerical values within the permitted limit of resources.

Examples:

- *Numerical accuracy* can be measured by the number of digits of precision, e.g. +/- 10^{-9}. The degree at which accuracy is considered as attained can still be fuzzy.
- *Compactness* (of data) can be measured by the size of data in a chosen unit, e.g. bytes.
- *Fast* (data transmission) can be measured by the size of data that is transmitted during a fixed period of time, e.g. bytes/second.

3.4.2. Partly measured quality

Partly measured qualities are those for which only a part of measurements can be carried out. This is usually the case when we know how carry out the complete measurements but are unable to so due to the resource restrictions.

Examples:

- *Availability* of a web site is a typical example where one can do a "ping" check at small intervals but fail to determine whether the site is available during the time between two consecutive checks.
- Any survey collected from a group of users about well-defined facts (like age, sex, language etc.) can be considered partly measurable.

For a partly measured quality, it is always possible to statistically project the partial measurement to have an approximation of the whole measurement. In some situation, the projected measurement is presented as the true overall measurement. For example, a web site can be said to have 99.999% uptime, which actually is a projection from "ping" checks.

3.4.3. Heuristically measured quality

Measurements that are partly or heuristically taken only give approximation of quality. However, all measurements in this case cannot be justified to be semantically correlated to the absolute quality, due to the loss of meaning cited above.

While this class is the most difficult to work with, most software qualities belong to it.

Example:

- One can say that a system is *understandable* because the number of lines of its source code is less than 1 million. However, others can say that the same system is not *understandable* since there are too many *if* clauses or *recursive* calls.
- To estimate how complex a web system interface is for users, one metric could count the number of elements and/or links inside web pages. Another one could compute the maximum, or minimum, or average number of links that a user has to follow to go from the homepage to each webpage of the website. However, the *complexity* may depend on other factors as well such as: the layout of elements, shortcut links …

3.4.4. Unmeasured quality

These qualities are those that cannot be measured or approximated at all. Constraints on cost, space, time, etc. can make measurements impossible.

For those qualities, the satisfaction of related non-functional requirements has to be guaranteed only at development time. No runtime control is possible if no measurement is taken.

3.5. Fulfilment of Non-functional Requirements

This last section of the chapter proposes several strategies that are useful for the fulfilment of non-functional requirements in multi-agent systems.

i) Make use of priority to resolve conflicts

Among intrinsic problems of non-functional requirements, disagreements between them happen quite frequently inside every design alternative. As already discussed in Section 4.2.5, the most probable scenario is when a design option favours some non-functional requirements and, at the same time, disfavours other requirements.

Here is another example of quality conflict. An online shop wants to store the buying history of its clients. There are more than one million clients every day. The system must access this history *rapidly* without violating user *privacy*. To take into account the *Rapidness* and *Privacy* requirements, when analyzing the *History Managed* hard-goal, the issue arises of whether (a) to store raw data or (b) to store encrypted data. Alternative (a) permits an instant access to the history but leaves open a possible leak of private user information. Alternative (b) takes better care of the *privacy* issue but lowers performance by encrypting and then decrypting the data. To make a decision, a priority among qualities must be decided as suggested in Section 4.2.5. If privacy is more valued, the second alternative (b) will clearly be a better choice than the first alternative (a).

ii) Prefer dynamical solutions to static solutions

In some cases, design solutions are only necessary conditions for the fulfilment of non-functional requirements.

In multi-agent systems, agents can be dynamically added and removed at run time. Potential problems identified in Section 3.3.1 suggest that there must exist a mechanism for quality control at run time. Furthermore, because non-functional requirements are usually considered subjective, relative and vague, design solutions, which

appear to sufficiently satisfice non-functional requirements, may become insufficient in some future situations. Therefore, design solutions must take into account the potential evolution of the system as well as of its environment. When possible, one should opt for *dynamical solutions* rather than static solutions.

For example, to insure the permanent availability of a service provider, consider two possible solutions: (a) create two (or more) agents of the same type, activate only one of them, and maintain the other(s) on hold until they are needed; (b) create service provider agents dynamically when needed. Solution (a) may be adequate and sufficient for static systems but it may fail in multi-agent solution when all agents are disconnected at the same time. Solution (b) requires a more complex framework to build such systems, but it can monitor the availability of service provider agents and dynamically create new ones when needed. Solution (b) is absolutely more suitable for multi-agent systems.

iii) Allow late operationalizations

It is sometimes impossible to design a system that can sufficiently fulfil all non-functional requirements, by just selecting the best design option among available alternatives. Some non-functional requirements, for example those of program code (reusability, modularity, understandability, etc.), cannot be fulfilled by changing design parameters. For those non-functional requirements, *Late Operationalizations* represent the possibility of leaving some of them for later considerations at the design stage and of postponing their treatment to later stages.

In multi-agent systems, agents can be designed and implemented separately. New agents can be developed after other agents have already been built and brought into operation. Since many non-functional requirements can only be satisfied by interactions between agents (especially between existing agents and newly introduced ones), late operationalizations are even more important than they are in traditional systems.

Late operationalizations also offer possibilities of fulfilling non-functional requirements by adjusting system run-time parameters. Indicators of their fulfilment can be measured and observed in order to detect any violation. When a violation is detected, an alert is raised and run-time parameters can be adjusted automatically or manually by system operators.

Chapter 5 describes design patterns that can help developers set up a sub-system for quality observation and control.

Chapter 4 GOALS AND QUALITY REQUIREMENTS

In the previous chapter, non-functional requirements are treated as constraints on quality aspects of system functions. In this chapter, we introduce a new notion, that we also call quality requirements, as constraints on hard-goals and soft-goals. Quality aspects are often described as non-functional requirements and treated as soft-goals to benefit from the techniques of goal analysis like AND/OR decomposition, conflict negotiations, operationalization, etc.

Our new definition of quality requirements separates them from soft-goals and treats them as an independent concept. We adapt the usual techniques for goal analysis so that they can take advantage of this additional notion.

Section 4.1 formulates basic definitions for quality requirements, hard-goals, and soft-goals. We explore links between goals and our new definition of quality requirements that suggest how to analyze soft-goals and quality requirements. Section 4.2 describes a refined goal analysis procedure that takes into account the new node type of quality requirements and three new types of link: elicitation, qualification, and contribution. Section 4.3 gives practical guidelines about how to efficiently carry out the new goal analysis. Section 4.4 illustrates the new material with an example that is meant to demonstrate its interest and usability

4.1. Definition of quality requirement

We first formulate the basic definitions, where the novelty essentially lies in the fact that quality requirements are defined as constraints on goals and not as non-functional requirements. Then, after some comments and examples, we complement the definitions by links between goals and quality requirements that express their

dependencies and that will lead to transformation operations in goal analysis (Section 4.2).

Definition 2:

> **Goal:** *describes a state that the system-to-be should be able to bring about.*
>
> **Hard-goal**: *is a goal for which satisfaction criteria can be precisely defined.*
>
> **Soft-goal**: *is a goal for which satisfaction criteria can not be defined in a clear-cut way.*
>
> **Quality requirement***: describes a constraint whose satisfaction or fulfilment ranges on a scale of possibilities and that can constrain hard-goals and soft-goals.*

Degrees of satisfaction or of fulfilment of quality requirements are typically expressed by abstract levels that range from not satisfied to fully satisfied (e.g., "high", "low", "average", "cheap", "expensive", "affordable" etc.).

Expressions like "Less Than 3 Euros" (for a price) and "Under 3 Seconds" (for a time delay) can be considered as concrete approximations of the degree of satisfaction of some quality requirements (e.g. "cheap" or "fast", respectively).

The following are examples using the proposed definitions:

- "Payment Sent" is a hard-goal since it describes a well-defined state (of having been sent) leading to defining payment functionalities.
- "Money Transferred with High Security" is a soft-goal since the level of security is not precisely stated.
- "Web Pages Served with High Availability" is also a soft-goal.
- "With High Security" and "With High Availability" are quality requirements.

We thus consider that the reason why it is difficult to define the satisfaction criterion of some soft-goals is the presence of one or more quality requirements closely integrated "inside the soft-goal".

As we said in Section 1.4, concerning the choice of term, using "quality requirement" for a constraint of a different nature than a non-functional requirement may not be the best idea. Just saying "quality" would not have been a better choice. We could maybe have chosen "quality constraint". We stayed with "quality requirement" and tried to be as clear as possible throughout the text to avoid ambiguities

Some soft-goals appear to be tightly integrated with a quality requirement, like "Web Pages Served with High Availability". In accordance to our definitions, one can alternatively represent such a

complex soft-goal as a quality requirement “High Availability” that constrains a hard-goal “Web Pages Served”. Similarly, the soft-goal “Money Transferred with High Security” can be understood as the combination of the hard-goal “Money Transferred” and of the quality requirement “High Security”.

Other soft-goals like “Make Customers Happy” do not explicitly “contain” a quality requirement, but the goals that could be derived from them in the goal analysis process (like , e.g., “High-Speed Connection Offered”) may make a quality more visible. In the example, the soft-goal “High-Speed Connection Offered” could be expressed as a combination of the hard-goal “Connection Offered” constrained by the quality requirement “High-Speed”.

Thus, the basic definition (Definition 2) is supplemented by the following definition of possible links between soft-goals and quality requirements: *some soft-goals can be viewed as the combination of a goal (hard-goal or soft-goal) and a quality requirement; such soft-goals can be alternatively re-expressed as a combination of the contained goal constrained by the quality requirement.*

4.2. Goal Analysis with Quality Requirements

We propose the following graphical notion for quality requirements in our analysis. Links from quality requirements to other elements to specify quality constraints will be called *qualification link*.

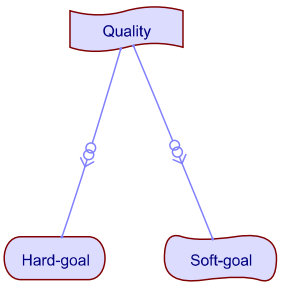


Figure 4-1: Quality on goals and soft-goals

A textual language could be easily defined to represent that information as follows:

```
QUALIFY([Quality], [Goal]).
```

in which goals and qualities are written inside a pair of square brackets [.]. The link types are in upper case. A set of elements are written inside a pair of curly brackets {.} and are used as abbreviations of individual links. In a link predicate, the order of parameters defines the direction of the arrow in the corresponding

graphical representation. Here we omit all the node predicates that specify the node type, i.e. QUALITY(.), HARDGOAL(.) and SOFTGOAL(.).

The next sub-sections present how to adapt goal analysis introduced in (VAN LAMSWEERDE, A., 2002) to cope with quality requirements. As usual, hard-goals and soft-goals are refined and decomposed with AND and OR decompositions. Quality requirements are propagated from a parent node to the child-nodes in the refinement tree according to the type of decomposition.

Tropos and NFR use only contribution links to decompose soft-goals. In our framework, we also allow the use of AND and OR decomposition for hard-goals and soft-goals since, with the presence of quality requirements, some soft-goals can also be exactly decomposed. Contribution links can also be used in the context in which a hard-goal or a soft-goal contributes to the satisfaction of a quality requirement.

4.2.1. OR decomposition

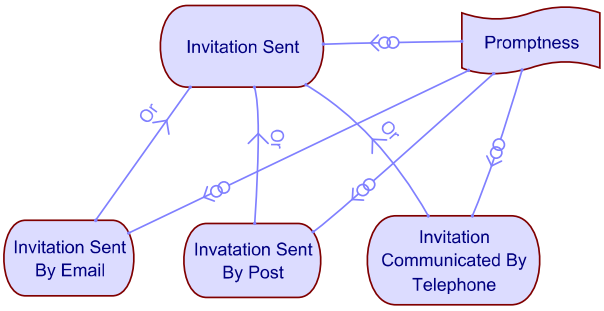


Figure 4-2: OR decomposition with quality requirement

OR decompositions represent alternatives to fulfil a goal. In Figure 4-2, the goal “Invitation Sent” is satisfied either by the goal “Invitation Sent By Email” or by the goal “Invitation Sent by Post” or by the goal “Invitation Communicated By Telephone”. Since the “Promptness” quality is required by the parent goal, “Promptness” should also be required of all the alternatives.

In textual form:

```
OR (
        {
                [Invitation Sent by Email],
                [Invitation Sent by Post],
                [Invitation Communicated by Telephone]
        },
        [Invitation Sent]
).
```

```
QUALIFY(
        [Promptness],
        {
                [Invitation Sent By Email],
                [Invitation Sent By Post],
                [Invitation Communicated by Telephone]
        }
).
```

Note that the above example can be analyzed differently if, for example, “Invitation Promptly Sent” is packaged as a soft-goal. Figure 4-3 shows a partial version of the example where the “Promptness” requirement has been only partly removed from the initial soft-goal “Invitation Promptly Sent”.

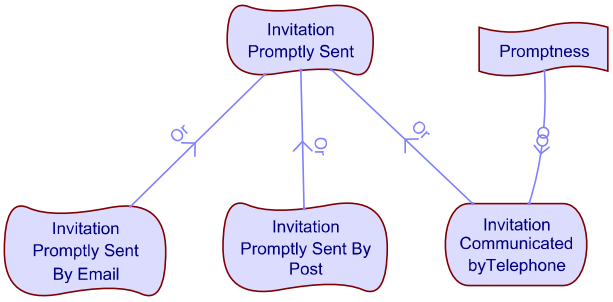


Figure 4-3: OR decomposition of Soft-goal

In textual form, this becomes:

```
OR(
        [Invitation Promptly Sent],
        {
                [Invitation Promptly Sent by Email],
                [Invitation Promptly Sent by Post],
                [Invitation Communicated by Telephone]
        }
).

QUALIFY(
        [Promptness],
        [Invitation Communicated by Telephone]
).
```

In fact, the first decomposition is preferable in our point of view, even if, in intermediate analyses, a partial decomposition of soft-goals is allowed. Figure 4-3 can also be described as produced by extracting “Promptness” from the soft-goal “Invitation Promptly Communicated by Telephone” to become a stand-alone quality requirement. We will come back to this point to adopt a new graphical representation making this relationship explicit.

4.2.2. AND decomposition

In AND decompositions, a goal is satisfied if and only if all leaf nodes of the decomposition are satisfied.

In Figure 4-4, the goal “Music Played” with the quality “Legality” can be satisfied if “Source File” is found and downloaded legally. Then the downloaded “Music File” is opened to send sound to speakers.

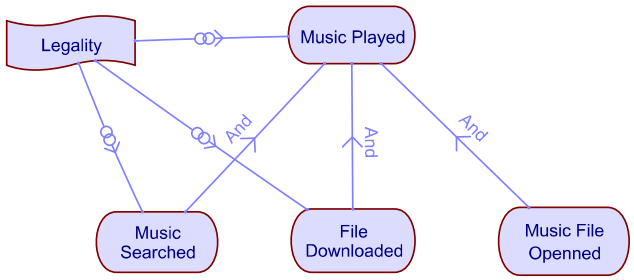


Figure 4-4: AND decomposition with quality requirement

Textually, we write:

```
AND(
        {
                [Music Searched],
                [File Downloaded],
                [Music File Opened]
        },
        [Music Played]
).

QUALIFY([Legality], [Music Played]).

QUALIFY(
        [Legality],
        {[Music Search], [File Downloaded]}
).
```

Notice that, in this example, the hard-goal “Music File Opened” is not concerned with the “Legality” requirement since “Legality” can be completely satisfied by the other two hard-goals. In the general case, the quality Q itself can be decomposed into several sub-qualities (i.e. Q1, Q2, ...), see, for example, (CHUNG, L. et al., 2000) for a taxonomy of quality requirements. The AND decomposition can be rewritten differently if all the sub-qualities are correctly distributed among the leaf nodes.

Figure 4-5 presents an extended version of AND decomposition. The quality “Economy” is decomposed into two sub-qualities: “Efficiency” and “Reusability”. To have “Software Developed Economically”, it must (i) be designed by using both existing and newly built reusable components, (ii) be provided with an efficient bug management system. The coding activities do not affect much the development

cost of the software. We then can rewrite the decomposition of the "Software Developed" hard-goal as follows:

```
AND(
        {[Efficiency], [Reusability},
        [Economic]
).

QUALIFY([Economic], [Software Developed]).

AND(
        {
                [Software Designed],
                [Software Coded],
                [Bug Managed]
        },
        [Software Developed]
).

QUALIFY([Efficiency], [Bug Managed]).
QUALITY([Reusability], [Software Designed]).
```

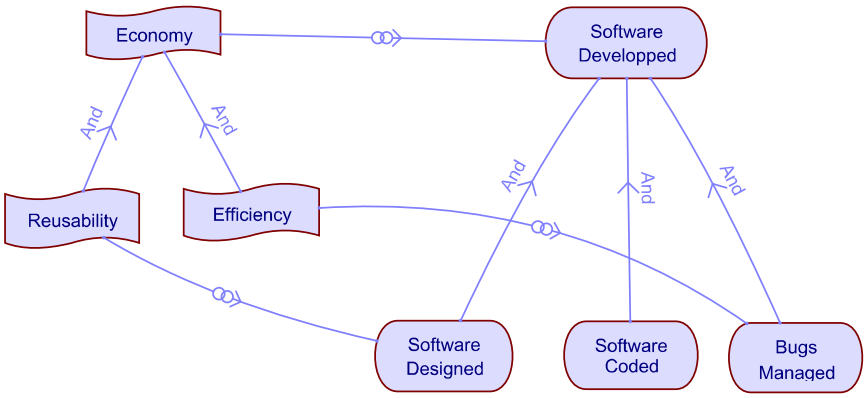


Figure 4-5: AND decomposition and extended quality requirement

AND decomposition can also be applied for soft-goals as done with OR decomposition. We omit such analyses here.

Note also that, when decompositions are partial or incomplete, contribution links can be used instead of AND or OR decompositions. This is helpful in cases where it is impossible to include the exact requirements, because of the special contexts or unknown structure of soft-goals.

4.2.3. Quality elicitation

We have seen that quality requirements can be packaged into high-level soft-goals. Those quality requirements can be elicited, that is, made explicit by extraction from a soft-goal.

Figure 4-6 shows how the soft-goal "Email Confidentially Sent" is decomposed into the hard-goal "Email Sent" with quality "Confidentiality". We thus say that quality "Confidentiality" is elicited from soft-goal "Email Confidentially Sent".

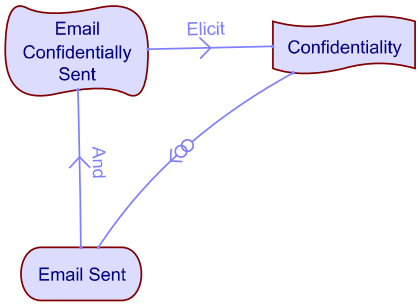


Figure 4-6: Quality requirement elicitation

Formally we write:

```
ELICIT(
        [Email Confidentially Sent],
        [Confidentiality]
).

AND([Email Sent], [Email Confidentially Sent]).
QUALIFY([Confidentiality], [Email Sent]).
```

In Figure 4-7 shows a typical soft-goal decomposition where soft-goal *S1* is satisficed by hard-goals *A*, *B* and *C*, soft-goal *S2* and elicited quality requirement *Q1* and *Q2*.

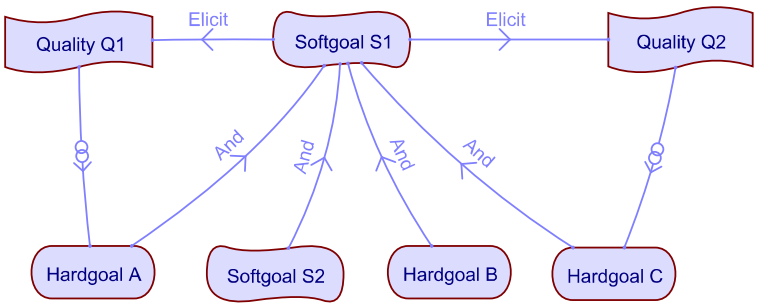


Figure 4-7: Soft-goal decomposition

Formally, we can write:

```
ELICIT(S1, Q1).
ELICIT(S1, Q2).
AND({A, B, C, S2}, S1).
QUALIFY(Q1, A).
QUALIFY(Q2, C).
```

4.2.4. Quality requirement fulfilment

Quality requirements are propagated through the decomposition of goals. At some stages, when there are possibilities to take care of quality requirements, contribution links can be used in combination with goal decomposition.

Contribution links, presented in A.1.2, were introduced by i* to help decompose soft-goals. Note that in i* and Tropos, these links are used exclusively to decompose soft-goals.

In our proposed approach, contribution links can also be used with quality requirements. The most common usage is to show how the satisfaction of a hard-goal can contribute to the satisfaction of a quality that the hard-goal is required to fulfil.

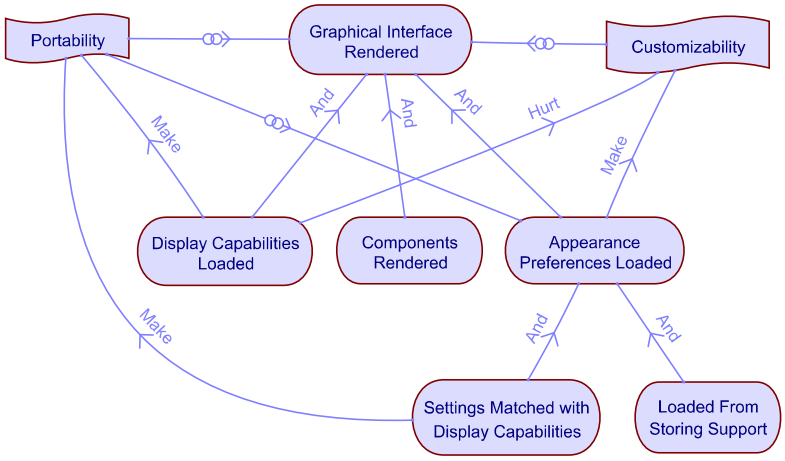


Figure 4-8: Quality requirement fulfilment

The example in Figure 4-8 shows a possible way to decompose the "Graphical Interface Rendered" hard-goal constrained by two quality requirements "Portability" and "Customizability". Functionally, the whole interface rendering can be simply split into the rendering of components. Considering quality requirements, to correctly render the graphical interface on different kinds of supports ("Portability" – PC monitor, Smartphone screen, Printer, etc.), we expect that all the parameters used in rendering on various systems must be available ("Display Capabilities Loaded"). In order to allow users to customize the graphical interface ("Customizability"), we expect that the rendering system takes into account pre-existing display settings (font face, colour, font size, etc.) by users.

To graphically show how these two additional hard-goals fulfil quality requirements, we add two *Make* contribution links from these hard-goals to their respective quality requirement in Figure 4-8. In textual form, we write:

```
QUALIFY([Portability], [Graphical Interface Rendered]).

QUALIFY(
        [Customizability],
        [Graphical Interface Rendered]
).

AND(
        {
                [Display Capabilities Managed],
                [Components Rendered],
                [Appearance Preferences Loaded]
        },
        [Graphical Interface Rendered]
).

QUALIFY([Portability], [Display Capabilities Managed]).

QUALIFY(
        [Customizability],
        [Appearance Preferences Loaded]
).

CONTRIB<MAKE>(
        [Display Capabilities Managed],
        [Portability]
).

CONTRIB<MAKE>(
        [Appearance Preferences Loaded],
        [Customizability]
).
```

Since the added hard-goals fulfil only their respective quality requirement, we must also consider their impacts on other quality requirements as well. In our example, the display capabilities of a specific platform may reduce the “Customizability” of the interface. Actually, to force the device with unsupported settings could lead to unexpectedly bad results. This is why we decide to further constrain “Appearance Preferences Loaded” with the “Portability” requirement. The new qualification link results in the decomposition of the “Appearance Preferences Loaded” hard-goal into hard-goals “Settings Matched with Display Capabilities” and “Loaded from Storing Support”. Matching to limit user settings to only allowable capabilities fulfils the “Portability” requirement. Additional analyses are written as:

```
QUALIFY([Portability], [Appearance Preferences Loaded]).

CONTRIB<Hurt>(
        [Display Capabilities Managed],
        [Customizability]
).

AND(
        {
                [Settings Matched
                        with Display Capabilities],
                [Loaded from Storing Support]
```

```
        },
        [Appearance Preferences Loaded]
).

QUALIFY(
        [Portability],
        [Settings Matched with Display Capabilities]
).

CONTRIB<Make>(
        [Settings Matched with Display Capabilities],
        [Portability]
).
```

The above example exhibits a conflict that is rather straightforward to resolve. In other cases, it is not so evident and needs special considerations introduced in the following section to select the best possible option.

4.2.5. Conflict negotiation

When analyzing requirements, it is possible that some entities are required to have multiple qualities whose fulfilments are contradictory. This is a frequent phenomenon, since a software product is always expected to have many qualities and some of them may be conflicting. If conflicting quality requirements constrain the same goal, it may become impossible to satisfy both the goal and all the quality requirements. For such situations, quality requirements usually suggest sets of design options. Typically, these sets do not intersect in the sense that a design option that meets one quality does not meet the others. Developers have to make the choice of which quality requirements should be fulfilled.

In order to ponder the available options, we adopt the evaluation technique introduced by Chung (CHUNG, L. et al., 2000) using contribution levels, i.e., Sufficient Negative (Break) < Partial Negative (Hurt) < Partial Positive (Help) < Sufficient Positive (Make). For each design option, we need to evaluate its contribution to the fulfilment of each quality requirement.

To illustrate this, we take the following example: an online store shop@shop that wants to allow its customers to make online payment. Store shop@shop wants its services to be both “Easy to Use” and “Secure”. To operate the store, it has to offer its customers an online payment service using credit cards. Two possible options are: (1) to develop and to deploy an in-house service; and (2) to use an external service offered by a reliable third-party company.

On the one hand, with the option “In-house Service Built”, shop@shop is free to design its own payment service including the interface to simplify and facilitate the customers’ payment. The accounting data is also kept and easily controlled. However, credit card process and management are very complex and often very

vulnerable to attacks. Building and maintaining a secure payment system solely for the store are very expensive.

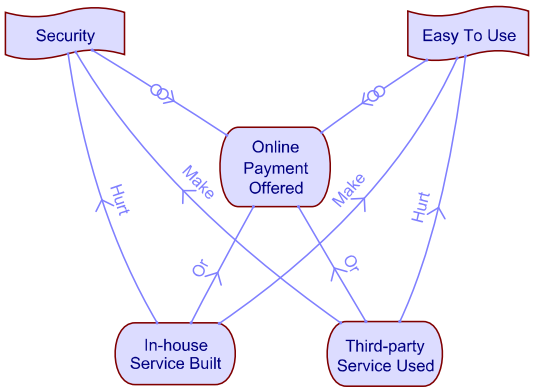


Figure 4-9 Quality conflict negotiation

On the other hand, existing third-party services are usually well built and maintained. Although they may still not provide a full guarantee for the security issue, they can often be considered a better choice than a self-developed service. We would choose the service offered by p@ym@te that has become a standard in online payment. The inconvenient side of this service is that it makes the payment process more complicated. The customers may need to jump back and forth between the site of shop@shop and p@um@te to complete a payment. For accountant service of shop@shop, using the external service also complicates its jobs with some additional transactions between p@yma@te and shop@shop.

To see which option is more suitable, we introduce the notion of *priority of quality requirement*. Requirement with higher priority should receive more attention than the lower priority one in making any design decision. This priority notion is local in the sense that a requirement can have very different priorities when acting on different goals. Concretely, the requirement A has a higher priority that the requirement B in fulfilling the goal G1 but B can have a higher priority than A in fulfilling the goal G2. We use an open scale for priority by putting a positive integer value next to the constraint link to represent the priority of the quality requirement on the corresponding node. This will helps the developers to select the right design parameters.

Reconsidering the above example, our analysis of “Online Payment Offered” hard-goal shows that the “Security” requirement should have a higher priority (e.g., 2) than the “Easy To Use” requirements should receive the same priority (e.g., 1). With this additional information, the right choice is clearly the third-party service.

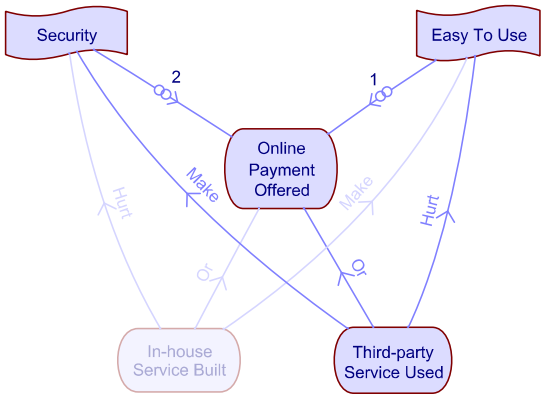


Figure 4-10: Priority of quality requirement

In textual representation, we can specify the priority label in the similar fashion as done with contribution links. Figure 4-10 can be represented as:

```
QUALIFY<2>(
        [Security],
        {
                [Online Payment Offered],
                [In-house Service Built],
                [Third-party Service Used]
        }
).

QUALIFY<1>(
        [Easy to Use],
        {
                [Online Payment Offered],
                [In-house Service Built],
                [Third-party Service Used]
        }
).

OR(
        {[In-house Service Built], [Third-Party Service Used]},
        [Online Payment Offered]
).

CONTRIB<Make>([In-house Service Built], [Easy to Use]).
CONTRIB<Hurt>([In-house Service Built], [Security]).

CONTRIB<Make>([Third-party Service Used], [Security]).
CONTRIB<Hurt>([Third-party Service Used], [Easy to Use]).
```

For shop@shop application, we might have to choose “Third-Party service” and drop out the possibility of an “In-House service” due to the significant additional cost. However, in other cases, if there is no other constraint on the implementation (cost, human resource, technical difficulty, etc.), one could also choose to keep all these alternatives. This would delay the choice to later phases or even implement them in the final application to let future users make

their own choices in accordance to their actual demand for quality. In the above example, "first-time" user would rather choose the service offered by shop@shop to complete their transaction than learn to use other services before being able to make a purchase. Note that if the "In-house Service" is implemented, it is still has the *security* requirement on it and major security measures should be implemented as well.

From the above analysis, we can summarize three important characteristics of our refined goal analysis:

- Quality requirements are typically implicit within high-level soft-goals at the start of the analysis process. They appear explicitly later from transformations of soft-goals.
- With elicitation links and qualification links, some soft-goals can be equivalently re-expressed as a combination of hard-goals constrained by quality requirements. The notion of soft-goal does not play a persistent role in the development process since it is possible to transform them into qualified goals. Those transformations should be done since hard-goals are clearer notions for describing the future system. Ideally, at some stages of the development process, there will be no soft-goal left.
- Quality requirements exist throughout the development process and play the role of qualifiers or constraints on goals.

The above characteristics strengthen our analysis that goals provide a richer dimension for requirements engineering than functional/non functional requirements. Moreover, they provide a higher degree of abstraction and expressiveness and can be used to capture early requirements only from the very early intentions of the system stakeholders.

4.3. Usage Guidelines of Goal Decomposition

We have argued that soft-goals cannot be used to describe the output of a specific system function because of their vagueness. Hard-goals cannot be used to describe vague business objectives.

In our point of view, if we take a soft-goal and try to decompose it into system services, we will, at some point, be able to elicit some quality requirements from soft-goals together with some hard-goals. This process will continue until all the leaf nodes are simple hard-goals that can be easily transformed into system services. It is possible that these leaf nodes are still constrained by some quality requirements. Such refinements suggest links between

functional/quality requirements and hard-goals/soft-goals, which can be summarized as follows:

- Lower-level hard-goals can be treated as classical functional requirements since they directly describe a function or a service of the future system.
- Having been operationalized, a hard-goal is satisfied by a system function and quality requirements can be considered as classical non-functional requirements constraining the derived function.
- A soft-goal is represented by a sub-tree of the refinement tree. Starting from a soft-goal and taking only the leaf nodes in the corresponding sub-tree, one can extract all the possible combinations of hard-goals and quality requirements that can satisfice the soft-goal. Quality requirements thus appear as sub-nodes in the decomposition tree of soft-goals.

Thus, roughly, soft-goals tend to appear near the root of refinement trees while functional and quality requirements tend to be at the leaf nodes. At the leaf nodes, there should not be any soft-goal left. Instead, there should be only simple hard-goals and quality requirements on them.

Here we formulate some rules that help the practitioners to obtain correct results when using our refined goal analysis. Applying these rules will prevent upward links in a top-down analysis, help soft-goals to be decomposed as soon as possible and help quality requirements to be correctly considered and satisfied.

- Hard-goals should not be decomposed into soft-goals. Actually, transforming a well-defined condition into a blurred condition does not give any benefits.
- Quality requirements should not be merged into goals to create soft-goals. Since the decomposition objective is to eliminate soft-goals but not to create more soft-goals.
- When a quality requirement is elicited, it should be immediately extracted from the original soft-goal to avoid redundancy.
- Contribution links should not be used to connect a soft-goal and a quality requirement. Because contribution links are used for operationalization of quality requirements but a soft-goal cannot be considered as an operationalization.

- Before adding a contribution link from a hard-goal to a quality requirement, the hard-goal should already be constrained by the same quality requirement.
- Justifications and descriptions of each element in analyses must be explicitly written down as documentation.

4.4. Illustrative example

To illustrate the usability of the above analysis techniques, we consider the example soft-goal "Payment Immediately and Securely Sent" in an online shop (Figure 4-11). Starting from that single soft-goal, we show how to use elicitation links to extract quality requirements from it, qualification links to constrain derived goals, and contribution links in the fulfilment of elicited quality requirements and in the selection of goal decomposition alternatives.

The analysis of this soft-goal is carried out through the following steps:

- The "Payment Immediately and Securely Sent" soft-goal is decomposed into the "Payment Sent" hard-goal and this elicits the "Security" and "Promptness" quality requirements. The "Payment Sent" hard-goal is constrained by the "Security" requirement and the "Promptness" requirement:

```
ELICIT(
        [Payment Securely and Immediately Sent],
        {[Security], [Promptness]}
).

AND([Payment Sent], [Payment Securely and Immediately Sent]).

QUALIFY({[Security], [Promptness]}, [Payment Sent]).
```

- The "Security" quality requirement is AND-decomposed into "Confidentiality" and "Integrity" quality requirements:

```
AND({[Confidentiality], [Integrity]}, Security).
```

- The "Payment Sent" hard-goal is AND-decomposed into "Authentication Sent", "Balance Checked", "Payment Ordered" and "Receipt Received":

```
AND(
        {
                [Authentication Sent],
                [Payment Issued],
                [Balance Checked],
```

```
                [Receipt Received]
        },
        [Payment Sent]
).

QUALIFY(
        {Promptness, Confidentiality},
        [Authentication Sent]
).

QUALIFY([Confidentiality], [Payment Issued]

QUALIFY([Integrity], [Balance Checked]).

QUALIFY([Security], [Receipt Received]).
```

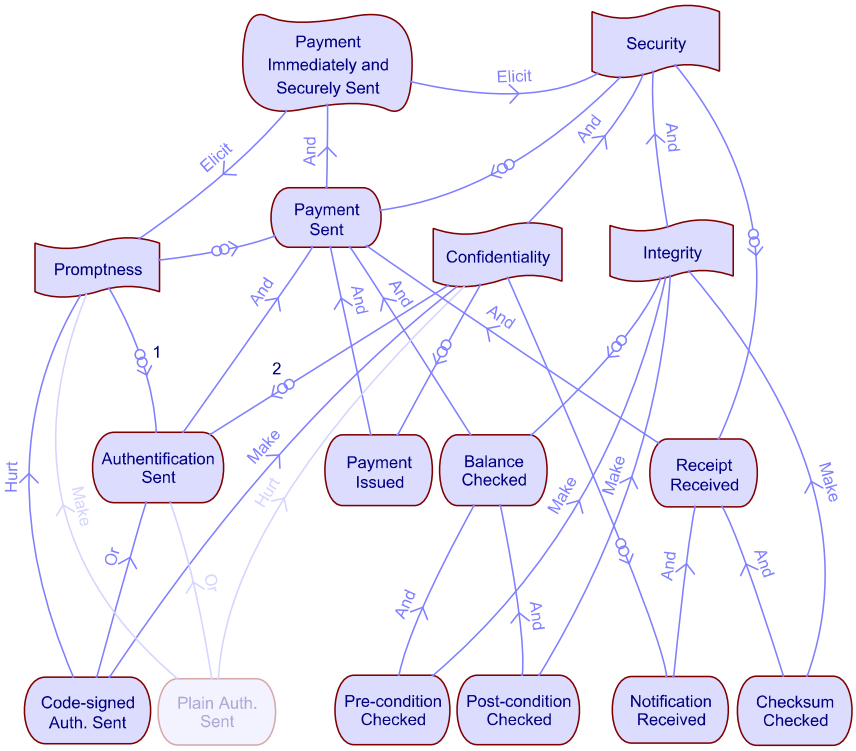


Figure 4-11: *Immediately and Securely Sent* soft-goal

- The “Authentication Sent” hard-goal is given two alternatives: the “Code-signed Auth. Sent” hard-goal and the “Plain Auth. Sent” hard-goal. The first alternative contributes positively (*Make*) to the “Confidentiality” requirement and negatively (*Hurt*) to the “Promptness” requirement since it is required with user-entered code to sign the authentication. The second alternative *Hurts* the “Confidentiality” requirement but *Makes*

the “Promptness” requirement satisficed. To break the tie in quality contribution between the two alternatives, we add priority to the “Promptness” and “Confidentiality” requirements (in the connection to the “Authentication Sent” hard-goal) by giving higher importance to the “Confidentiality” requirement.

```
QUALIFY<1>([Promptness], [Authentication Sent]).

QUALIFY<2>([Confidentiality], [Authentication Sent]).

OR(
        {[Code-signed Auth. Sent], [Plain Auth. Sent]},
        [Authentication Sent]
).

QUALIFY<1>(
        [Promptness],
        {[Code-signed Auth. Sent], [Plain Auth. Sent]}
).

QUALIFY<2>(
        [Confidentiality],
        {[Code-signed Auth. Sent], [Plain Auth. Sent]}
).


CONTRIB<Make>([Code-signed Auth. Sent], [Confidentiality]).
CONTRIB<Hurt>([Code-signed Auth. Sent], [Promptness]).


CONTRIB<Make>([Plain Auth. Sent], [Promptness]).
CONTRIB<Hurt>([Plain Auth. Sent], [Confidentiality]).
```

- The “Balance Checked” hard-goal is AND-decomposed into the “*Pre-condition Checked”* and the “*Post-condition Checked”*, hard-goals, i.e.

```
AND(
        {[Pre-condition Checked], [Post-condition Checked]},
        [Balance Checked]
).

QUALIFY(
        [Integrity],
        {[Pre-condition Checked], [Post-condition Checked]}
).

CONTRIB<Make>(
        {[Pre-condition Checked], [Post-condition Checked]},
        [Integrity]
).
```

- The “Receipt Received” hard-goal is AND-decomposed into “Notification Received” and “Checksum Checked”.

```
AND(
        {[Notification Received], [Checksum Checked]},
        [Receipt Received]
).

QUALIFY([Confidentiality], [Notification Received]).

QUALIFY([Integrity], [Checksum Checked]).

CONTRIB<Make>([Checksum Checked], [Integrity]).
```

At the end of the goal refinement process, there remain two hard-goals that the “Confidentiality” requirement constrains: “Payment Issued” and “Notification Received”. We decide to leave this quality requirement for latter consideration because they depend on payment services that the third-party offers. The remaining quality requirements can be reconsidered when additional information is available and/or during the implementation using special coding techniques. This represents the *late-operationalization* possibility offered by the proposed approach.

From the final analysis, one can create a summary of different possible solutions. This summary is created by identifying all the leaf nodes of the decomposition tree together with all quality requirements that constrain at least one leaf node. The objective of a summary is to produce only one AND-decomposition of each root goals together with some QUALIFY links whose satisfaction is left open. OR-decompositions at leaf level generate alternatives.

In our example, from Figure 4-11, satisficing the top-level soft-goal “Payment Immediately and Securely Sent” can be summarized as follows:

```
AND(
        {
                [Code-signed Auth. Sent],
                [Pre-condition Checked],
                [Post-condition Checked],
                [Notification Received],
                [Checksum Checked]
        },
        [Payment Securely and Immediately Sent]
).

QUALIFY(
        [Confidentiality],
        {[Payment Issued], [Notification Received]}
).
```

or by this alternative:

```
AND(
        {
                [Plain Auth. Sent],
                [Pre-condition Checked],
                [Post-condition Checked],
                [Notification Received],
```

```
                [Checksum Checked]
        },
        [Payment Securely and Immediately Sent]
).

QUALIFY(
        [Confidentiality],
        {[Payment Issued], [Notification Received]}
).
```

where the only difference lies in the format of authentication information. Alternatives are created by the decomposition:

```
OR(
        {[Code-signed Auth. Sent], [Plain Auth. Sent]},
        [Authentication Sent]
).
```

Authentication information is encrypted in the first alternative and is in plain text in the second alternative. Given the higher priority given to the "Confidentiality" requirement over the "Promptness" requirement, the first alternative is the preferred solution.

Note that the above two possibilities of satisficing the "Payment Securely and Immediately Sent" soft-goal do not contain any OR-combination since all the OR-decompositions in the refinement tree are translated into alternative possibilities.

One can argue that the bottom part of Figure 4-11 is somewhat similar to the diagram of goals/soft-goals integration presented in (MYLOPOULOS, J. et al., 2001). However, in their approach, soft-goals are mainly quality requirements, while hard-goals and quality requirements are analyzed separately and are correlated late in the analysis process. In comparison, our approach provides developers with an opportunity to analyze hard-goals, soft-goals, and quality requirements in an integrated scheme from the highest level and most abstract goals. If we remove ELICIT and QUALIFY links from our analysis, it becomes similar to the goal decomposition in (MYLOPOULOS, J. et al., 2001). However, the price to be paid for this simplification is the suppression of traceability of quality requirements, analysis rationale, and late-operationalization of quality requirements.

Compared to two-concept (hard-goals and soft-goals) approaches, the approach proposed in this chapter is semantically richer. Quality requirements are decoupled, as the analysis proceeds, from the functional part using three additional links: ELICIT, QUALIFY and CONTRIB. Our modelling notions are designed in such a way that we can separate two important parts: the quality part and the functional part.

Chapter 5 SOCIAL PATTERNS FOR QUALITY CONTROL

Chapter 4 has presented a classification or requirements in terms of hard-goals, soft-goals, and quality requirements. It has presented a goal analysis process adapted with additional tools to deal with the newly introduced class of quality requirements. The central idea of this thesis is to argue in favour of addressing some aspects of software quality through conformance to relevant requirements.

At the leaf nodes of the goal decomposition tree, there could be hard-goals constrained by quality requirements that are still left untreated. The satisfaction of some quality requirements depends on conditions that cannot be fully identified and made explicit during requirement analysis. As a consequence, some quality control is often needed during system implementation, testing, deployment, and runtime for those constrained hard-goals.

We do not discuss implementation and testing techniques. They cover a large scope that does not fit the main themes of this thesis. By explicitly indicating which quality requirements constrain which specific functions (defined by hard-goals at leaf nodes of the resulting goal tree), programmers and testers should have enough information to carry out necessary actions to satisfy constraining quality requirements.

This chapter focuses on design options that favour the control of software quality during the execution of the system. Section 5.1 is devoted to a brief description of quality control and social patterns. We describe five social dimensions that underlie the proposed patterns. Then Section 5.2 proposes a catalogue of several patterns that can assist developers in designing a quality control sub-system. Section 5.3 illustrates the use of this catalogue in a concrete example.

A preliminary version of the catalogue of social patterns presented in this chapter was published in (HOANG, T.T.H. and Kolp, M., 2009).

5.1. Quality Control and Social Patterns

Consider, for example, an online store featuring a Banking Agent that monitors the arrival of payments. Any received payment must be reported immediately to the Order Manager. If fast shipping is an important quality for gaining the confidence of customers, requirement analysis could impose that an order must be shipped as soon as the corresponding payment has been received. It is thus important to detect any suspicious delay between the deposit time mentioned on payment receipts and the notification time to the Order Manager, and to adjust, at run time, the interval between two consecutive checks of such delays.

This is a simple example where the satisfaction of quality requirements can be measured and taken care of at run time. In real systems, quality requirements may interact positively or negatively with each other in elaborate ways. To control their fulfilment, software developers have to design and implement a suitable quality control mechanism that can measure software qualities and react to risks of quality violations.

Social patterns are design patterns (GAMMA, E. et al., 1995) to which *s*ocial, intentional, structural, communication, and dynamic dimensions have been added. Since design patterns have proved useful to facilitate the design of good object-oriented systems, it is natural to extend patterns to social patterns in multi-agent systems (GAMMA, E. et al., 1995), (KOLP, M. et al., 2005) and (ARIDOR, Y. and Lange, D.B., 1998).

We describe a conceptual framework based on five complementary dimensions introduced in (DO, T.T. et al., 2003) and (DO, T., 2005) to investigate social patterns. Each dimension reflects a particular aspect of multi-agent system architecture.

- The *social dimension* identifies relevant agents in the system and their intentional interdependencies represented in the i* style.
- The *intentional dimension* identifies and formalizes services provided by agents to realize the intentions identified by the social dimension, independently of the plans that implement those services. This dimension answers the question: "What does each service do?"

- The *structural dimension* operationalizes the services identified by the intentional dimension in terms of agent-oriented concepts like beliefs, events, plans, and their relationships. This dimension answers the question: "How is each service operationalized?"
- The *communicational dimension* models the temporal exchange of events between agents.
- The *dynamic dimension* models the synchronization mechanisms between events and plans.

The social and the intentional dimensions are specific to multi-agent systems. The last three dimensions (structural, communicational, and dynamic) are relevant, in general, for traditional (non-agent) systems as well.

5.2. Social Patterns for Quality Control

In this chapter, we address only product quality requirements (constraining the runtime of the system) and ignore quality requirements on process such as time and cost constraints constraining system development.

The standard ISO/IEC 9126-1 (ISO/IEC, 9126-1, 2001) defines software product quality by means of general characteristics of software, which are further decomposed into several sub-characteristics, which are in turn refined into measurable attributes. This hierarchical structure depends on a set of metrics that measure and aggregate measurable attributes to compute sub-characteristics and then characteristics of software. Inspired by this three-level structure, we derive our multi-level structure by using two elements: quality metrics and signal. A quality metrics uses values of some signal types to compute a value representing the satisfaction of a quality requirement. The output of a metrics can be, in turn, considered as a value of another signal type and can be used as input for other quality metrics, and so on. This recursive structure creates also a hierarchical structure similar to that defined in the standard ISO/IEC 9126-1, but with a more flexible organization.

This structure of quality metrics helps create a corresponding structure of agents for our quality control subsystem. This subsystem should be able to monitor the fulfilment of quality requirements (measured or computed using metrics) at runtime and to react dynamically whenever a quality requirement is violated. We identify the following issues that must be taken into account when building such a subsystem.

First, according to the described hierarchical structure of quality metrics, a quality metrics can use more than one signal types. As a consequence, agents that are responsible for the quality measurement should have access to the source of those signal types. Every change of value of those signals results in a change in the satisfaction of the quality requirement.

Second, according to the typical structure of large systems, the satisfaction of quality requirements can involve several components distributed on different host machines and on different geographic sites. Therefore, to keep track of a quality requirement, we may need to deploy several agents that are responsible for quality management. There should also be a communication mechanism by which they exchange information about satisfaction status.

Third, in practice, the cost for applying quality control must be justified by benefits resulting from quality improvement (SLAUGHTER, S.A. et al., 1998). In particular, when deploying quality control, one has to decide which quality requirements should be measured and which information flows and management will be involved in order to control the cost of quality control. Strategies for measurement acquisition may depend on specific quality requirements. For example, less important signals or continuous signals could be acquired by a pulling mechanism (sampling), while concrete signals or system events should be collected by a pushing mechanism. A nice discussion about the use of those two data delivery mechanisms is presented in (ACHARYA, S. et al., 1997).

To deal with these three issues, we define the following agent roles that should be implemented in our proposed quality control sub-system:

- Signal Source: emits signals used in the measurement of some quality requirements on the system. Signal source must support pulling and/or pushing mechanism of data delivery.
- Signal Manager: maintains an up-to-date list of available signal sources together with the type of signal and possible mechanisms of data delivery (pull and/or push).
- Quality Meter: combines several signals to compute informative values reflecting the fulfilment degree of quality requirements.
- Quality Manager: is responsible for detecting quality violations using the indications given by quality meters as well as for checking required conditions before doing some actions.

- Quality Assurer: involves agents that are required to fulfil some quality requirements for their attributes, plans or operations. They trigger the procedures of quality control and assurance.

The following subsections detail the proposed patterns that help developers design the quality management subsystem and integrate this subsystem into the system-to-be.

5.2.1. Signal pushing pattern

This pattern specifies interactions between a "Signal Source", *a* "Signal Manager", and a "Quality Meter". It applies in cases where every change in the signal is pushed to the "Quality Meter". The main interactions are summarized in Figure 5-1.

i) Social dimension

To receive signals from a "Signal Source", every "Quality Meter" needs to subscribe to it. Each "Signal Source" maintains a list of current subscribers to which it sends any change in signal values. "Signal Sources" are discovered by "Quality Meters" with the help of "Signal Manager" that maintains a signal directory.

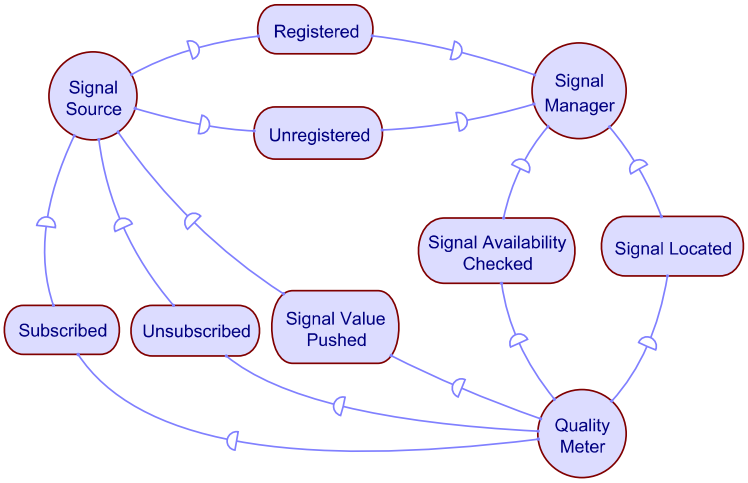


Figure 5-1: Signal Pushing Pattern

Dependencies: "Registered", "Unregistered", "Subscribed", "Unsubscribed" are described in the Subscription pattern (KOLP, M. et al., 2005) and (DO, T., 2005) using a formal pattern description language based on Formal Tropos (FUXMAN, A. et al., 2001) and

first-order temporal logics. The following definitions describe other dependencies in that language for which some predefined predicates can be found in (DO, T., 2005).

Dependency [Signal Value Pushed] (st: [Signal Type])
Mode Achieve
Type Goal
Depender [Quality Meter] qm
Dependee [Signal Source] ss
Fulfilment

$$\forall val\colon [Signal\ Value].\begin{bmatrix}(val.type = st) \wedge provide(ss, st) \wedge\ subscribed(qm, ss)\\ \wedge\, produce(ss, val)\\ \rightarrow \diamond\, known(qm, val)\end{bmatrix}$$

[Quality Meter qm wishes Signal Source ss, to which qm has subscribed, send all its signal values to it]

Dependency [Signal Availability Checked] (ss: [Signal Source])
Mode Achieve
Type Goal
Depender [Quality Meter] qm
Dependee [Signal Manager] smgr
Fulfilment

$$registered(ss, smgr) \wedge pinged(smgr, ss, pingMsg, pongMsg) \rightarrow available(ss)$$

[Quality Meter qm wishes Signal Manager smgr to keep track of the availability of Signal Source ss by pinging ss with pingMsg and verifying if received message is pongMsg]

Dependency [Signal Located] (st: [Signal Type])
Mode Achieve
Type Goal
Depender [Quality Meter] qm
Depender [Signal Manager] smgr
Fulfilment

$$\forall ss\colon [Signal\ Source].\begin{bmatrix}ofType(ss, st) \wedge registered(ss, smgr) \wedge provide(ss, st)\\ \rightarrow \diamond\, known(qm, ss)\end{bmatrix}$$

[Quality Meter qm wishes that Signal Manager smgr let it know about all registered Signal Source ss that can emit Signal Type st]

The following define the missing predicates in the above descriptions:

$$Pinged(from, to, pingMsg, pongMsg) :=\ \bullet\, request(from, to, pingMsg) \rightarrow recieved(to, from, pongMsg)$$

ii) Intentional dimension

The following table lists services offered by agents participating in the Signal Pushing pattern. Services described in the Subscription pattern are omitted.

Service Names	**Informal description**	**Agent**
SendRegisterRequest	Send a Request to a Signal Manager to be registered	Signal Source
SendOKDecision	Send a OK decision as reply, some specific data can be included with the decision	*
SendKODecision	Send a KO (not OK) decision as reply, some specific data can be included with the decision	*
....	Other services from Subscription pattern	
QueryAvailState	Send a query for current state of a signal source	Quality Meter
ReturnAvailState	Return the current state of a signal source.	Signal Manager
SendSignalLocateQuery	Send a request for locating signal sources based on the signal type	Quality Meter
ReturnSignalList	Return a list of matched signal sources in response to a SendSignalLocateRequest	Signal Manager
ReturnNotFoundMsg	Return a "Not Found" message in response to a SendSignalLocateRequest	Signal Manager
SelectSignalSource	If SendSignalLocateRequest gives a list of potential signal sources, QualityMeter select one of the signal source based on its owned criteria.	Quality Meter
PushSignalValue	Push Signal Values to all agents that are registered.	Signal Source
RecordSignalRecvState	Record the state of receivers of signal values	Signal Source

Table 3: Agent services in Signal Pushing pattern

We do not enter in all the details of those services which can be described in the same way as done for the above dependencies.

iii) Communication dimension

The details about how "Quality Meters" discover the "Signal Sources" with the help of "Quality Manager" are omitted here.

Messages that can be exchanged between "Signal Sources" and "Quality Meter" are summarized in Figure 5-2.

The interactions between “Signal Sources” and “Quality Meters” in the “Pushing Signal” pattern are started when a “Quality Meter” sends a subscription request to a “Signal Source”. If the request is accepted, the “Signal Source” adds the “Quality Meter” to its list of subscribers.

At fixed intervals or at timestamps specified by the “Signal Source”, it takes the current value of the Signal and sends a change notification to all its subscribers.

If no errors are encountered, the “Quality Meter” receives notification messages until an “Unsubscribe” request sent to the “Signal Source” is accepted.

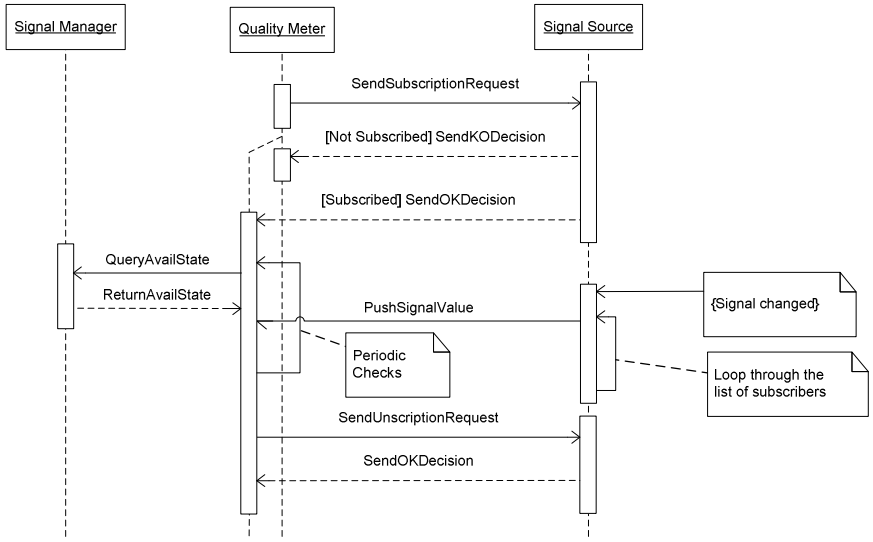


Figure 5-2: Sequence diagram - Signal Pushing pattern

The sequence diagram in Figure 5-2 can also be detailed using the dynamic diagram and services described. The dynamic diagram is used to describe the dynamic dimension of a pattern.

Agents, plans, events, and beliefs used to detail the pattern can be described by extended class diagrams for agent-oriented software. Such descriptions are presented in the structural dimension.

Both dynamic and structural dimensions are omitted here for brevity.

It could be argued that the Signal Pushing Pattern just presented could be expressed in terms of the *Observer* pattern (GAMMA, E. et

al., 1995) and the *Matchmaker* pattern (DO, T.T. et al., 2003) and (KOLP, M. et al., 2005). However, in the context of multi-agent systems and quality control subsystems, some reasons for not doing so are the following:

- The presence of a "Signal Manager" is essential, since it guarantees and verifies the availability of signal sources. Without it, 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 any changes in the signal.

The presence of all three agents "Signal Source", "Signal Manager", and "Quality Meter" represents the *existence* constraint of these agents inside the system-to-be.

5.2.2. Signal pulling pattern

Contrary to the pushing pattern presented above, "Quality Meters" plays an active role in the Signal Pulling Pattern. The signal is acquired only on demand determined by the monitoring strategy of the signal monitor. The social diagram is shown in Figure 5-3.

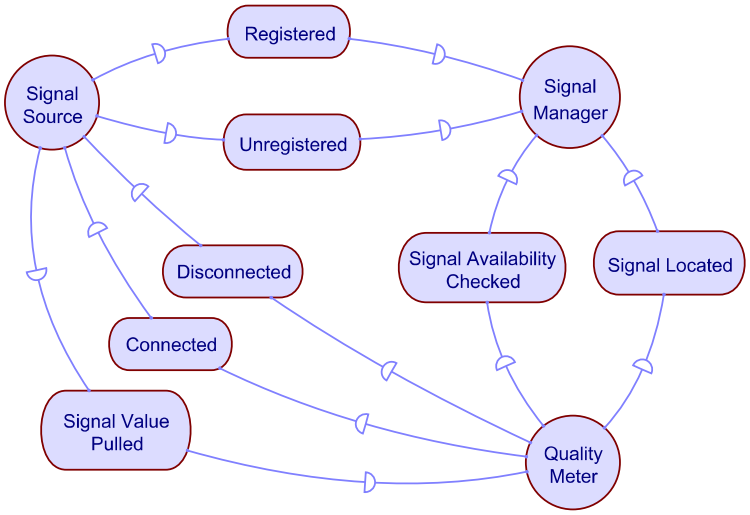


Figure 5-3: Signal pulling pattern

The “Signal Value Pulled” dependency can be described as follows:

Dependency [Signal Value Pulled] (st: [Signal Type])
Mode Achieve
Type Goal
Depender [Signal Source] ss
Dependee [Quality Meter] qm
Fulfilment

$$\forall val\colon [Signal\ Value].\begin{bmatrix} ofType(val, st) \wedge provide(ss, st) \wedge (now\ \in qm.pullTS) \\ \wedge\ connected(qm, ss) \wedge produce(ss, val) \\ \rightarrow \Diamond\ known(qm, val) \end{bmatrix}$$

[Quality Meter qm wishes that Signal Value val can be pulled from Signal Source ss at a predefined timestamp. Variable now store the system clock and the set qm.pullTS contains all predefined timestamps at which Quality qm Meter wants to pull values]

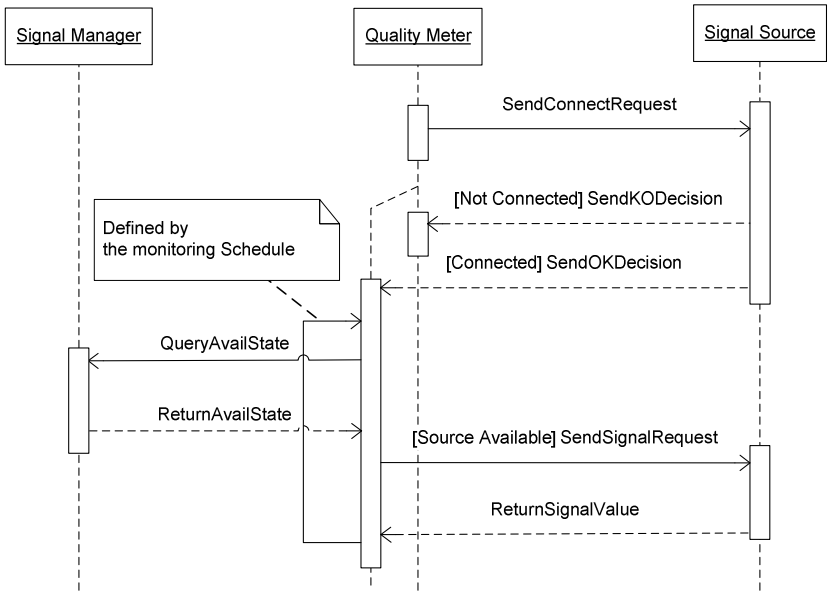


Figure 5-4: Sequence diagram – Signal Pulling pattern

The “Quality Meter” first has to connect to the “Signal Source” and maintain that 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. The “Signal Manager” helps “Quality Meters” to discover and to keep track of the availability of “Signal Sources”. The pulling sequence is described in Figure 5-4. The description of other dimensions is similar to those in the Signal Pushing pattern. The disconnection request from “Quality

Meter" to "Signal Source" can be ignored since the active agent is "Quality Meter".

Again, it could be argued that the Signal Pulling Pattern just presented could be expressed in terms of the *Observer* pattern (GAMMA, E. et al., 1995) and the *Matchmaker* pattern (DO, T.T. et al., 2003) and (KOLP, M. et al., 2005). However, in the context of multi-agent systems and quality control subsystem, the existence and availability of participating agents is crucial to assure good application of patterns, they must be coupled.

5.2.3. Quality assurance pattern

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

Precondition check: when an agent is about to carry out a plan, 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 every possible plan has been tried without success, the plan executer must be notified to cancel the requested plan.

Intermediate check: during the execution of a plan, it is possible that the fulfilment of a quality requirement is violated. This may be caused by the plan itself but also by other agents. It is also possible that quality requirements differ at each stage of the action. In either case, intermediate checks are very important. These checks can be carried out by predefining a list of several check points, usually points vulnerable to the fulfilment of the quality at stake, 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 it may decide to continue with the plan, or to start 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 are usually known before the plan is started, while the intermediate conditions depend on the operations actually carried out by the plan.

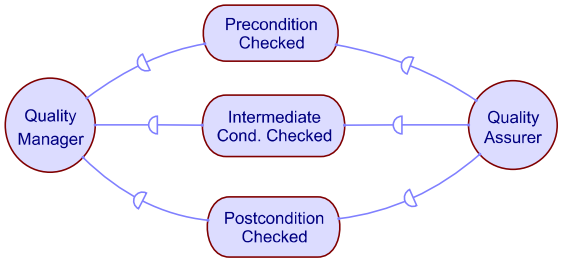


Figure 5-5: Quality assurance pattern

The social diagram is presented in Figure 5-5. For each type of check, five actions (services) are defined as in the following table.

Service Names	**Informal description**	**Agent**
SendCheckRequest	Send a Request to a Quality Manager for checking the fulfilment state of a quality requirement.	Quality Assurer
SendCheckFailedRes	At each request, Quality Manager do a check for the fulfilment of quality requirements. He can reuse old check result if the check is not too old. This service is used when the fulfilment is not met.	Quality Manager
SendCheckOkRes	Send an OK message if the check of quality fulfilment is sufficiently satisfied.	Quality Manager
RemoveViolation	Try to remove violations of quality requirements.	Quality Manager
ResolveConflict	Try to resolve a conflict with other quality requirements	Quality Manager

Table 4: Services for quality check

The sequence diagram depicting when checks are requested by quality assurers is shown in Figure 5-6. As presented above, at least two checks are required at the beginning and at the end of the execution of a plan. The number of intermediate checks may not be known until the actual execution takes place. During the plan, if one check returns a failed message, than the plan should be stopped and cancelled.

One of the most important features that the “Quality Manager” must implement is how it handles violations of quality requirements and conflicts in the fulfilment of quality requirements. Two services are provided for this purpose: “RemoveViolation” and “ResolveConflict”. However, the implementation of those services can only be detailed by taking into account the actual list of quality requirements and the actual situations in which the plan is carried out.

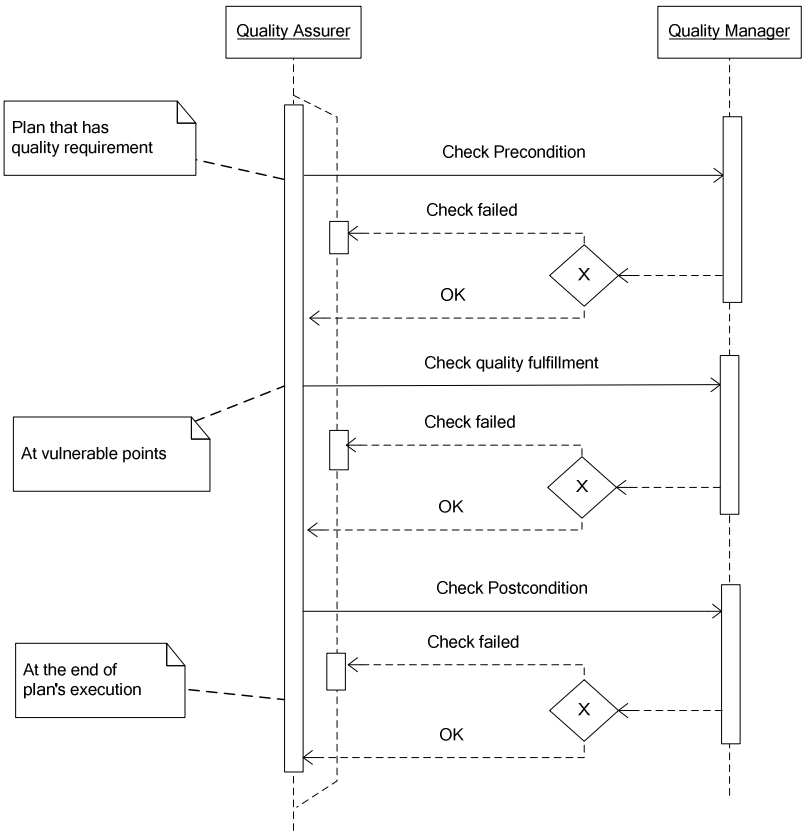


Figure 5-6: Sequence diagram - Quality Assurance pattern

5.2.4. Total quality manager pattern

In a complex system, when the number of quality requirements becomes large, responsibility needs to be distributed among several "Quality Managers". This partition can be made based on:

- Topology setup: a system may operate on different physical 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 into a hierarchy where lower-rank managers have to report to their direct higher manager.

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

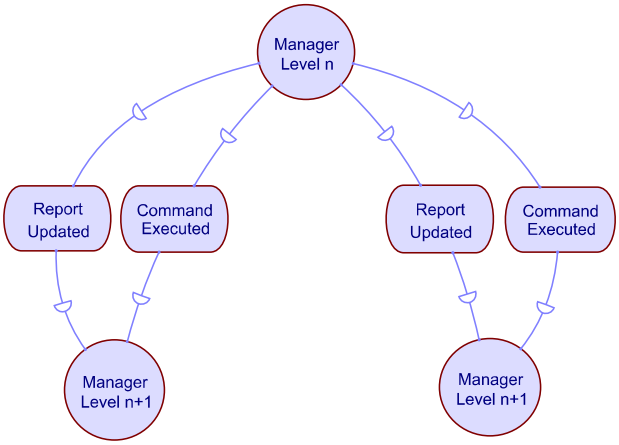


Figure 5-7: Total quality manager

This pattern provides a total control of the Quality Management Subsystem even for complex systems. Quality managers can be organized in several levels depending on the organizational structure of the system. However, designers 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 a heavy organization. Furthermore, system designers 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.

The social diagram in Figure 5-7 lists relevant services (only relations between two consecutive levels of managers are shown).

Service Names	Informal description	Agent
SendReportRequest	Send a request for report about the fulfilment of a quality requirement.	Higher Lever Manager
ReturnNoReport	After having received a request for report, the manager verifies if there is any report left. If not, it will notify that no available report is found.	Lower Level Manager
ReturnLatestReports	If there are several reports that are not sent, it will send all the pending reports. Only reports that were issued not so long ago are returned.	Lower Level Manager
SendCommandRequest	Ask a lower level manager to do something.	Higher Level Manager
CarryOutCommand	Execute commands that are asked by higher level manager.	Lower Level Manager
ReturnCommandResult	Return results of execution of command to higher level manager.	Lower Level Manager

Table 5: Services for total quality management

The description of other dimensions for this pattern is rather simple and is omitted.

We have introduced a list of social patterns that can be used to design a subsystem that can control and carry out action to assure quality requirements. This list is not exhaustive. More useful patterns could certainly be identified.

5.3. Example: Quality Control for Web Server

This section illustrates the use of the proposed patterns in a concrete example to control the Availability and Integrity of web servers.

Figure 5-8 presents a solution for a farm of web servers that are used to serve a large number of web pages. The scenario of usage is as follows:

- In a web browser (“Client Browser actor”), a user enters the URL (“Universal Resource Locator”) of a webpage into the address box.
- A request is created and relayed across the internet and is received by our master server (“Master Server actor”).
- The “Master Server” does not serve the page itself. Instead, it searches, in the list of “Web Servers” which are available and can serve the requested page, the most suitable server. Then the master server redirects the request to the chosen server.
- The requested page is sent back to the “Client Browser”.

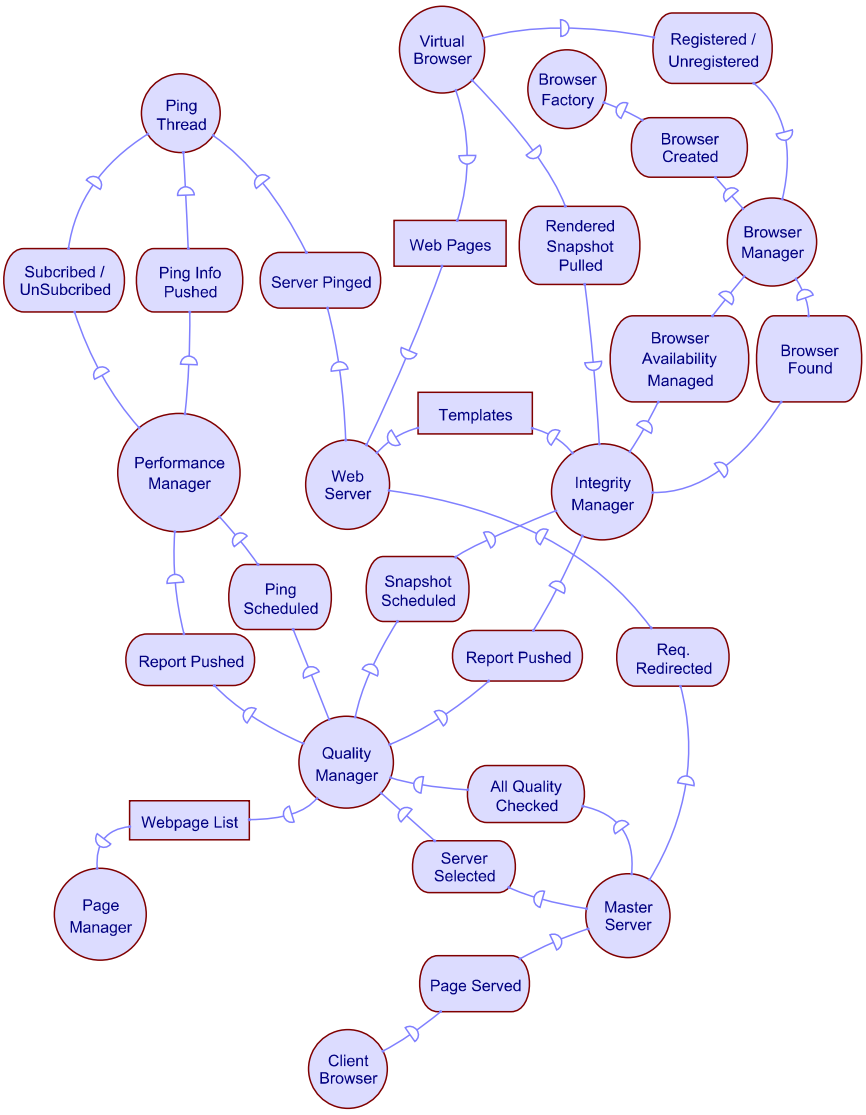


Figure 5-8: Quality Control for Web Servers

We concentrate on two quality requirements: the “Performance” of servers and the “Integrity” of web pages. By “Performance”, we mean that the request loads are distributed equally to all available “Web Servers” in the system such that the average response time is minimized for incoming requests. The “Integrity” of web pages means that each web page is controlled to be correctly displayed in the “Client Browser”. The following two sections present a solution that uses our patterns for the assurance of these two quality requirements.

5.3.1. Webpage Integrity

From our definition, the "Integrity" of a web page can be measured by the difference between the rendered webpage in the "Client Browser" and the intended result presented in a template.

A template is automatically generated when a web page is physically created or modified on a "Web Server". Each template is a picture taken in a window of a standard browser together with some meta-data that can help to compare different pictures of the same web page. These templates can be served using a special internal protocol.

We define a snapshot of a web page as a picture taken in a browser window that displays that web page.

To organize agents to do this control, we create an "Integrity Manager" that is responsible for creating templates and snapshots of web pages and for comparing snapshots to the corresponding template.

To take snapshots, we create several "Virtual Browsers" to simulate exactly the behaviours of real browsers on different platforms. Each "Virtual Browser" is responsible for a specific browser on a specific platform. The agent that manages all the "Virtual Browsers" in the system is called "Browser Manager".

We apply the Signal Pulling pattern to the composition of the three agents: "Integrity Manager", "Browser Manager" and "Virtual Browser", in which the "Integrity Manager" agent plays the role of "Quality Meter", the "Browser Manager" agent plays the role of "Quality Manager" and "Virtual Browser" agents play the role of Signal Source. The "signal values" that is pulled from the "Virtual Browser" by the "Integrity Manager" are snapshots of web pages. An additional agent named "Browser Factory" is also added to help the "Browser Manager" to guarantee the "Availability" of the "Virtual Browsers".

The "Integrity Manager" of the system will carry out comparisons between the snapshots taken by "Virtual Servers" and the corresponding template taken from the "Web Server". Similarity degrees between a snapshot and a template picture, computed thanks to the meta-data in the template, help the "Integrity Manager" to determine the "Integrity" of a web page.

5.3.2. Web Server Performance

To control the performance of each server in the system, we run several independent threads at different locations on the internet. Each thread contains a list of web servers that must be measured by simply sending ping message and receiving the result.

The most important information that can be collected from a ping is the *response time*. It a server is overloaded, it will not be able to quickly answer the ping message until it is freed by the pending jobs.

A ping thread is scheduled to send a ping message at fixed intervals and to send a report to its subscribers at larger intervals. In Figure 5-8, we use the Signal Pushing pattern between "Performance Manager", "Ping Thread" and "Ping Thread Manager". We omit in the diagram the "Ping Thread Manager" for brevity.

5.3.3. Management of Quality Requirements

In order to use the reports sent by the "Integrity Manager" and "Performance Manager" to assist the "Master Server" in serving "Web Pages" to client browser. We use the hierarchical structure of the "Total Quality Manager" pattern to create a top-level "Quality Manager" agent.

The "Quality Manager" has "Integrity" report and "Performance" report updated by the respective managers. It uses those reports to help the "Master Server" to select the right server. If a requested web page is missing from the "Integrity" report, the "Quality Manager" will ask the "Integrity Manager" to schedule "Virtual Browser" to take snapshots of those web pages. Likewise, when a new web server is introduced or there is missing from the "Performance Report", the "Quality Server" can send a request to the "Performance Manager" to reschedule its "Ping Threads".

We can also use the Quality Assurance pattern between the "Quality Manager" and the "Master Server" because when a web page is requested, it has to follow several stages before the page is actually served.

This chapter has presented a way to help multi-agent systems cope efficiently and systematically with quality requirements. The proposed social patterns help control the fulfilment of quality requirements at run time. The proposed control scheme is based on collecting and combining signals to create indicators of the fulfilment of quality requirements.

These patterns are independent from the development methodology. In the following chapter (Chapter 6), the Tropos methodology is revised to better take into account the quality requirements, in particular by applying the proposed patterns in the Architectural Design stage.

Chapter 6 QTROPOS – QUALITY AWARE TROPOS

Chapter 4 proposed a new definition of quality requirements, and then a refined goal analysis for soft-goals, hard-goals, and quality requirements. Chapter 5 complemented this analysis with a catalogue of social patterns for quality control in multi-agent systems. These two chapters present our main propositions for software in particular for multi-agent systems, with a high conformance to *relevant* requirements, hence high quality. The material is described in a sufficiently generic manner that it can be applied to several software development methodologies, in which goals play the central role, like those listed in Appendix C.

Tropos, presented in Appendix A, is a good candidate methodology that could benefit from the application of our proposed material. Tropos is considered as one of the most complete methodology in terms of coverage of workflow. Therefore, taking Tropos as an example helps us to better validate our approach and to open the possibility to other methodologies as well.

The objective of this chapter is to integrate our goal analysis and social patterns into Tropos to create Quality-aware Tropos or QTropos. We analyze the consequences of this integration in all phases of the Tropos development process, from Early Requirements and Late Requirements, where quality requirements are elicited; to Architectural Design, where quality requirements help define the system structure; and to Detailed Design, where quality requirements influence internal design of agents. Then, information about quality requirements and the constrained components in the resulting system design will guide programmers to adjust their coding of system functions to the identified quality requirements.

The application of quality requirements in Tropos, to create QTropos, covers the central sections of this chapter (Sections 6.2 to 6.6). Section 6.1 describes some limitations of Tropos in the treatment of

non-functional requirements to the problem of requirements traceability described in (GOTEL, O. and Finkelstein, A, 1994). This analysis helps us to identify five criteria that a multi-agent system should meet in order to ensure a high traceability of requirements related to software qualities.

6.1. Limitations of Tropos in the Analysis of Non-Functional Requirements

As presented in (CASTRO, J. et al., 2002) and in Appendix A, non-functional requirements in the original Tropos are represented as soft-goals. A soft-goal cannot be refined. Instead, other components, including other soft-goals, hard-goals, tasks, and resources can contribute negatively, positively, or not at all to fulfilling it. This allows developers using Tropos to evaluate design alternatives and implementation choices in order to maximize the fulfilment of related non-functional requirements. More precisely, given a list of possibilities for technical solutions or architectural structures, the strength of contribution links that connect those possibilities and non-functional requirements provide a basis for comparing options and choosing the most suitable one. This particular way of treating non-functional requirements has the following limitations:

- Non-functional requirements can be taken into account as early as in the early requirement phase. Their propagation from one analysis stage to the next one is hidden. In fact, there is no propagation at all, since the list of identified non-functional requirements is only used in the late requirement phase and the architecture design phase to compare alternatives in means-ends analysis and in the selection of organizational styles, see Appendix A for more detail.
- Non-functional requirements are set to influence all of the elements inside an actor. If the system-to-be is represented by an actor, which is the case in most analyses found in the literature, non-functional requirements do influence all elements of the system-to-be. Questions that we frequently asked ourselves when examining a Tropos analysis are: *why is a particular element chosen to fulfil non-functional requirements rather than others? Or why do we have to compare refinement alternatives for a particular element but not for the others?* Relevant answers are usually not obvious.
- The use of social patterns at the architectural design stage is not quality-aware, that is, no pattern is available to help taking care of and controlling non-functional requirements. Existing social patterns, e.g., those in (DO, T., 2005), may

themselves cause risks of violation of some non-functional requirements.

The problems of social patterns must be treated by their designers. Quality aspects must be considered during the design process of patterns. Or at least, a list of quality violation risks must be provided with each pattern. We do not address those issues for the existing patterns proposed in the literature. Instead, we have proposed in Chapter 5 a catalogue of other social patterns that can be used to develop a pluggable sub-system for quality control for multi-agent systems.

Two other problems of Tropos concerning non-functional requirements can be related to question of requirements traceability analyzed in (GOTEL, O. and Finkelstein, A, 1994). They refer to requirements traceability as "*the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases).*"

Applying this definition to quality-related requirements (i.e., non-functional requirements in Tropos), quality requirements (in our approach) or other terminologies in multi-agent systems, it can be interpreted as the ability to describe and follow quality-related requirements from the first development phase to the last one (or all the way backwards, horizontally), from the highest level, i.e. the whole system downwards to the lowest level, i.e., agent plans (all the way upwards, vertically) and during the specific treatment for quality-related requirements, i.e., elicitation, fulfilment and operationalization. From this, we formulate five concrete criteria for assessing any development methodology of multi-agent system in terms of the traceability of quality-related requirements, as follows:

C1. Horizontal traceability: quality-related requirements must be analyzed throughout the development process, from the first requirement stage to the final product. Moreover, the treatment of quality requirements must be coherent from one stage to the next one.

C2. Vertical traceability: quality-related requirements must be analyzed at all the decomposition levels of the system. When an item is broken down into smaller parts, there should be a mechanism to derive quality-related requirements for the parts from those of the original item.

C3. Traceability in elicitation: software developers should be able to elicit new quality-related requirements throughout the development process, since it is difficult to foresee all

of them from the beginning and since they are usually elicited by the analysis of stakeholder intentions.

C4. Traceability in fulfilment: software developers should be able to keep track of the fulfilment status of every quality-related requirement. It should be possible to automatically prevent requirements about quality from being lost during the development.

C5. Traceability in operationalization: it should be possible to operationalize a quality-related requirement or leave it untreated, at one stage, for further consideration at later stages. Quality-related requirements can be left untreated as far as the implementation stage during which they can be taken care of by the choice of programming language, coding types, interface design, etc., and even as far as the deployment stage where they can be taken care of by adjusting operational parameters.

In the original Tropos, as presented in Appendix A, criterion C1 is partly satisfied since non-functional requirements are considered only during the phases of early requirements, late requirements, and architecture design.

Criterion C2 is not well taken care of, since non-functional requirements defined as soft-goals are not flexible enough to adjust to every decomposition level of the system. Besides, contribution links in Tropos maybe confusedly understood as decompositions.

Tropos fails to meet Criterion C3, since all non-functional requirements are explicitly introduced in the strategic dependency analysis, during early and late requirements, as dependencies between actors.

Criterion C4 is essentially supported by Tropos. Still, quality assessment is only carried out down to the architectural design phase and only in options selection. This prevents Tropos from taking advantage of more analysis to fulfil non-functional requirements. The absence of non-functional requirements in the later phases is also the reason why Tropos fails to meet Criterion C5.

According to these five criteria, we can see that the traceability of non-functional requirements in Tropos can still be improved. And this motivates us to bring about some improvements in the remaining sections of this chapter.

6.2. Overview of QTropos Process

The key tool used to modify the original Tropos methodology is based on the discussion presented in Chapter 4. We use the new definitions of hard-goal, soft-goal, and quality requirement (Definition 2) to redefine how to analyze elements in i*. Changes in i* imply that the whole Tropos methodology must be revised.

The proposed approach helps to add another dimension to i* and to Tropos: the quality dimension. Decoupling quality requirements from other notions helps not only to simplify analysis tasks but also to enrich Tropos. The most important gain lies in the use of quality requirements in the Tropos process at all levels, which helps to capture and to fulfil all quality requirements in the target system.

In our revised framework, quality requirements influence both the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. The propagation of quality requirements from the former to the latter naturally follows the goal refinement process. A quality requirement, which acts on the system-to-be actor, is related to goals and/or tasks and/or resources inside the system that are responsible for that quality fulfilment. Moreover, quality requirements are propagated from the earliest phase to latest phase of the development process in a natural way. This strengthens the consistency of our approach.

Section 6.3 begins the integration of the quality dimension in Tropos by introducing now graphical notations for quality requirements and for related analysis links.

6.3. New Notations of QTropos

We first introduce extensions to *i**. In order to distinguish quality requirements from soft-goals, we use the graphical notion introduced in Chapter 4 to depict quality requirements.

One can use the catalogue of non-functional requirements proposed in (CHUNG, L. et al., 2000) in the analysis. Non-functional requirements can be decomposed using AND and OR operators, offering a hierarchical view of how a non-functional requirement can be decomposed into and satisficed by sub-requirements. The approach is largely independent of the development context. It can thus be used as a dictionary of quality requirements.

This section introduces some new notations for i* to cope with quality requirements. This section focuses in the notation part. The application part is presented in Section 6.3.1 where new analyses are added to Tropos to deal with quality requirements as a separate modelling class.

6.3.1. Quality Requirements and Qualification Links

As introduced in Chapter 4, our quality requirements constrain hard-goals and soft-goals. To be able to apply quality requirements to Tropos, we have to extend quality requirements so that they can constrain other intentional elements of Tropos, namely Tasks and Resources.

The *Qualification Link* defined in Chapter 4 is used to constrain hard-goals and soft-goals to fulfil quality requirements. In this chapter, we extend this link to other elements of i*, as illustrated in Figure 6-1.

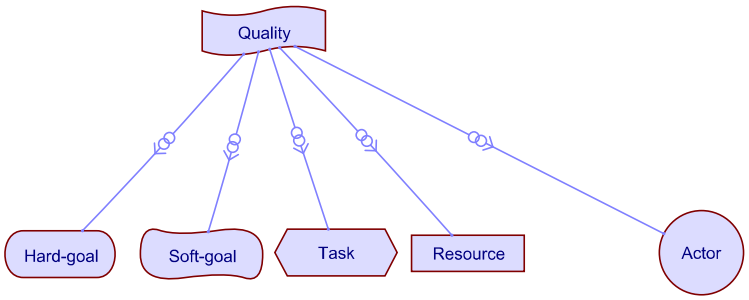


Figure 6-1: Quality in i*

We also include the possibility of qualification links between quality requirements and actors. Such links imply qualification links from a quality requirement to all the components inside the corresponding actor when they are made explicit in the strategic rationale model. However, this type of "abbreviation" should be avoided in general because, as argued in Section 6.1, this reduces the traceability of quality requirements at different levels of decomposition of the system.

Qualification links are also extended to constrain dependencies, or more precisely dependums. In this case, the depender depends on the dependee to carry out or to obtain the dependum constrained by some quality requirements. In i*, dependencies are used to exchange intentions between agents. Therefore, adding quality requirements on dependencies guarantees that quality requirements will be correctly exchanged together with their respective dependum.

Figure 6-3 shows all possibilities of using quality requirements to constrain dependencies. Quality requirements are no longer considered as dependum. Instead, they become separate elements that constrain those dependums.

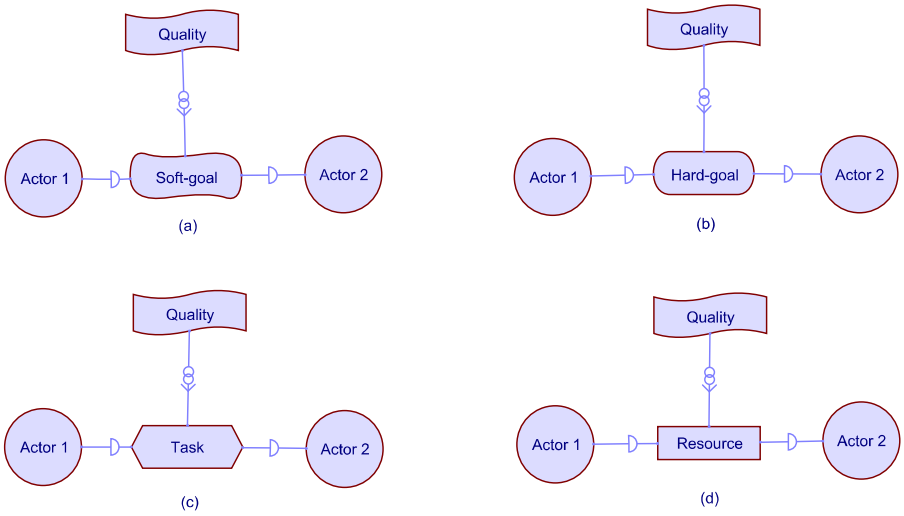


Figure 6-3: Dependency quality

The example in Figure 6-2 illustrates how to use quality requirements to constrain dependums. It shows simple dependencies between actors "Patient" and "Doctor". "Patients" depend on "Doctors" to be "Examined", to "Receive Treatment" if needed, and to be sent a "Report" about their health condition. Furthermore, Patients expect to receive "efficient" examinations and "efficient" treatments. "Reports" must be handled and handed to patients with privacy.

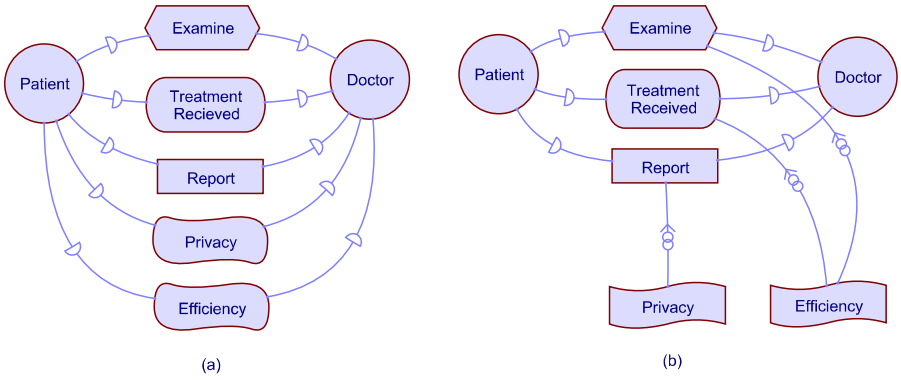


Figure 6-2: Strategic Dependency model: original (a) vs. quality-aware (b)

Figure 6-2(a) is a typical strategic dependency model in the original i*. Figure 6-2(b) shows the new analysis with our treatment of quality

requirements, where quality requirements are separated from soft-goals rather than integrated within soft-goals.

As we can see in Figure 6-2, the revised version (b) captures much better the relevant intentions than the original Tropos can do (a), since it shows that the task "Examine" and the hard-goal "Treatment Received" are constrained by the quality requirement "Efficiency". Likewise, the quality requirement "Privacy" constrains only the "Report" that the actor "Doctor" sends to the actor "Patient".

6.3.2. Elicitation Link

We use the generic quality elicitation presented in Section 4.2.3 to introduce the quality elicitation mechanism in i*. This new link describes the action of identifying the presence of a quality requirement inside a soft-goal and taking it out of the soft-goal.

Remember that, in the original i*, non-functional requirements are represented as soft-goals and introduced mainly during the strategic dependency analysis. In our revised approach, quality requirements can be elicited not only from the strategic dependency model but also in the strategic rationale phase when goals are refined and decomposed. This strategy offers to analysts a possibility of adding new quality requirements on children nodes in order to better fulfil requirements on the parent node.

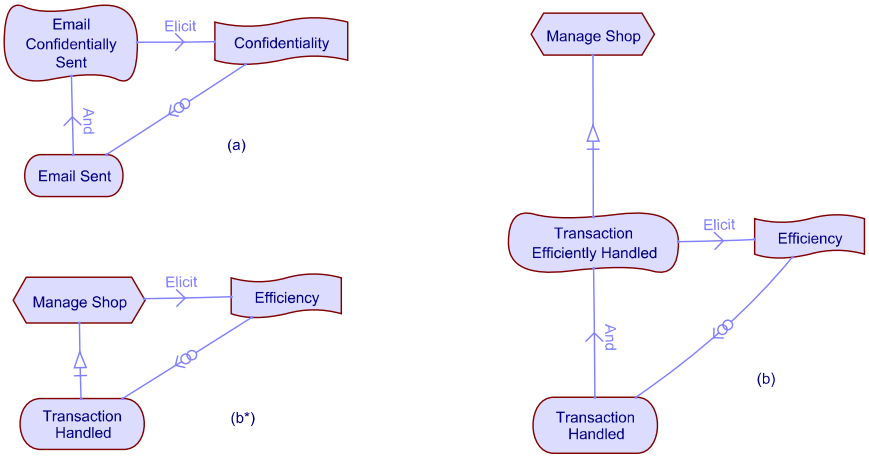


Figure 6-4: Quality elicitation

Modifications to goal decomposition in i* are presented in the next section (Section 6.3.3). Now we consider examples of how elicitation links can be integrated into i*. Figure 6-4(a) is Figure 4-6 of Section 4.2.3 Soft-goal "Email Confidentially Sent" elicits quality requirement

"Confidentiality" that constrains hard-goal "Email Sent", which is a child node of soft-goal "Email Confidentially Sent".

Essentially, the elicitation process described here helps to make explicit quality requirements that were integrated in the parent node.

In Figure 6-4(b), task "Manage Shop" is decomposed into several child nodes including soft-goal "Transaction Efficiently Handled". This soft-goal elicits quality "Efficiency" that constrains child hard-goal "Transaction Handled", which is the single child node of soft-goal "Transaction Efficiently Handled".

We do not allow elements other than soft-goals to elicit a quality requirement. Figure 6-4(b*) suggests that task "Manage Shop" could elicit quality "Efficiency". Figure 6-4(b*) is thus equivalent to Figure 6-4(b). However, we argue that the abbreviation in Figure 6-4(b*) reduces the traceability of the "Efficiency" quality requirement (Criterion C4 of Section 6.1) that we consider important.

6.3.3. Goal Decomposition with Quality Requirements

In the original i*, only Means-Ends links are available to decompose a hard-goal and create alternative tasks to fulfil the hard-goal. Contribution links are used in analyzing soft-goals

Our analysis in Chapter 4, as well as a goal-based methodology like KAOS (DARIMONT, R. et al., 1997), allow AND links and OR links for both hard-goals and soft-goals.

i) AND and OR link

We summarize briefly Section 4.2.1 and 4.2.2 in the following rules that must be taken into account to use correctly AND and OR links in our proposed extension of i*:

- AND and OR links are used only for hard-goals, soft-goals, and quality requirements. They cannot be used for tasks and resources.
- We allow AND links and OR links exclusively for goals (i.e., hard-goals and soft-goals). In Sections 4.2.1 and 4.2.2, AND and OR links are presented in a general context. Here we focus on their use in extending i* and in a comparison with the original i*.
- The fulfilment relation between a parent node and its children nodes is *exact* in contrast to the contribution links where the fulfilment maybe only approximate.

ii) Contribution link

In the original i*, contribution links can be used to connect any element to a soft-goal. The fulfilment of the children node contributes positively or negatively to the fulfilment of the parent node.

In Chapter 4, we briefly described the use of contribution links when it is impossible to decompose exactly and completely soft-goals. In the original i*, contribution links are exclusively used for decomposing soft-goals.

In our extension, contributions links can be used when:

- decomposing soft-goals whose structure is unknown or whose complete decomposition is impossible (or unaffordable).
- connecting hard-goals, soft-goals, tasks, or resources to quality requirements to represent the fulfilment of quality requirements.

The first usage must be avoided in cases where soft-goals can be correctly decomposed into hard-goals constrained by quality requirement, as shown in the following example.

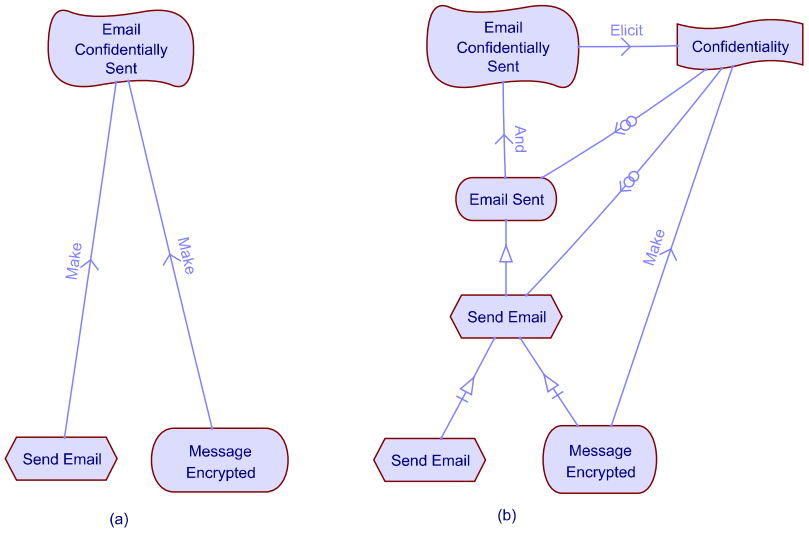


Figure 6-5: Contribution links

Figure 6-5 presents two decompositions of the soft-goal “Email Confidentially Sent”. Figure 6-5(a) shows a typical analysis in the original i*, where task “Send Email” and hard-goal “Message Encrypted” contribute to the fulfilment of the parent soft-goal. Figure 6-5(b) follows our new analysis without contribution links to soft-goals.

The semantics of Figure 6-5(b) is much clearer. It exhibits a complete and exact decomposition of the soft-goal. Contributions links in Figure 6-5(a) do not make explicit whether the soft-goal is decomposed or approximated. Syntactically, two contribution links can be understood to approximate a soft-goal while they could semantically provide an exact decomposition if their contribution message is correctly captured.

This section has mostly introduced graphical notations that can enrich i*. In the next section, these notations will be used in different phases of the QTropos methodology.

6.4. Quality Requirements in QTropos

In this section, we go through all the phases of the Tropos development process: Early Requirement, Late Requirement, Architectural Design and Detailed Design. Because the analyses used in Early and Late Requirements are similar, the next subsection is reserved for these first two phases.

6.4.1. Early and Late Requirements

As presented in Section A.2.1 and Section A.2.2, the Early and Late Requirement phases use the strategic dependency and strategic rationale models to analyze the system requirements.

This section proposes several new analysis techniques focusing on quality requirements, in addition to the standard ones in the original Tropos methodology.

i) Elimination of quality-only soft-goal

Figure 6-2 illustrates a comparison between our new strategic dependency model and that of the original i*. According to our Definition 2 in Chapter 4, the presence of quality requirements prohibits the usage of the soft-goal notation to describe only a pure non-functional requirement.

The process of transforming an original i* strategic dependency model into the new one consists in:

- matching every soft-goal dependency in the model with the new definitions and transforming every *quality-only soft-goal* into a quality requirement.
- searching, for each newly added quality requirement, dependencies in the same context (connecting the same pair

of actors) that should be constrained and creating qualification links from the quality requirement to the corresponding dependums.

This procedure is illustrated in Figure 6-2.

As a special case, in the strategic dependency model of the Late Requirement phase, most quality requirements can be inferred from the strategic rationale model of the Early Requirement phase.

ii) Quality requirement delegation and assignment

We detail the use of the basic operations presented in Section 4.2 to deal with the analysis of quality requirements in Tropos. This section adapts the AND and OR decompositions of *qualified* goal to fit the Tropos notions.

In the strategic rationale model, we focus on the internal rationale of an actor. Internal soft-goals, goals, tasks and resources of an actor are added to satisfy the dependum of which the actor is the dependee.

When qualities are added to the external dependency, they are also propagated inside the actor. We show how the two main analysis tasks, namely *means-ends analysis* and *task decomposition analysis*, of the strategic rationale model can be modified in order to handle quality requirements.

Means-ends:

Older practice of i* used to define four principal means-ends links: hard-goal – task, resource – task, soft-goal – task and soft-goal – soft-goal. To support the current practice of i*, here we develop only the hard-goal – task link, which proposes alternatives to satisfy hard-goals.

Each link in Figure 6-6 is described as a “means” to satisfy the corresponding “end”. Each “means” is an alternative to meet the “end”, this is why any quality required for the “end” will be required for each “means”. When an “end” can be satisfied by some alternatives, each alternative (means) will have at least the same quality requirements as that “end”, as shown in Figure 6-7. This makes sense since every quality requirement must be satisfied no matter which alternative is chosen.

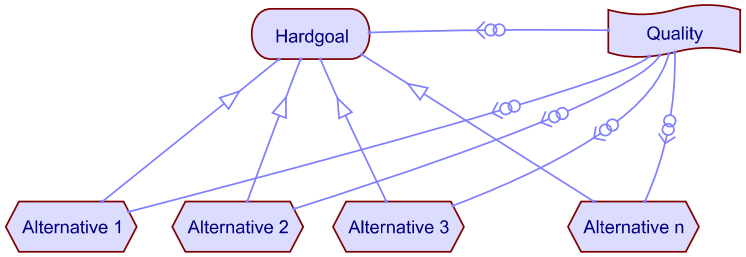


Figure 6-6: Means-ends quality propagation

In the example of Figure 6-7, the idea is to satisfy soft-goal "Attractive to new users" by one of three alternatives:

- "Flyer Sent": potential users are sent information about the services offered
- "Advertised on Media": services offered are advertised on the media
- "User Service Improved": improve its reputation of good service to attract new users

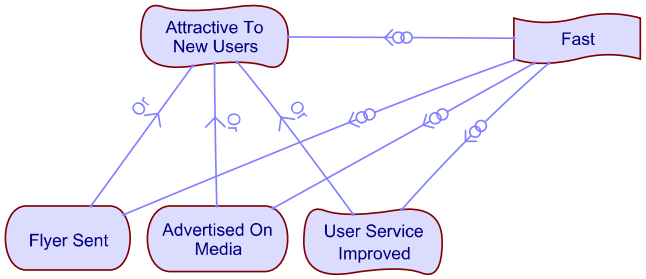


Figure 6-7: Quality delegation to means

If the service offered is new to the market and there are many competitors, then not only new users should be attracted, but they should be attracted as fast as possible. This is described by the quality requirement "Fast" on soft-goal "Attractive to New users". This quality requirement must be propagated to the alternative "means" to create a "Fast Flyer Sent" task, a "Fast Advertised on Media" task, and a "Fast User Service Improved" soft-goal. Then, with this propagation, "Fast Flyer Sent" could be realized for example by "Use Postal Direct Mail" offered by the post office.

Thus the idea is to find a means to satisfy both the task and the quality requirement. If a means used to realize the "ends" helps to satisfy the quality requirements as well, then the qualification link can be replaced by a make link from the means to the quality requirement, as shown in Figure 6-8.

Still, in general, tasks and soft-goals with quality requirements may not be fulfilled by simple tasks. Tasks with quality requirements could be further analyzed in a task decomposition analysis. In Figure 6-8, task "Use Postal Direct Mail" can satisfy soft-goal "Attractive to New Users" with quality "Fast", while tasks "Advertise on TV" and "Advertise on Street Panels" can have to face delays linked to the availability of the advertising spots. The latter two tasks are still constrained by the "Fast" quality requirement and that will have to be considered in subsequent steps.

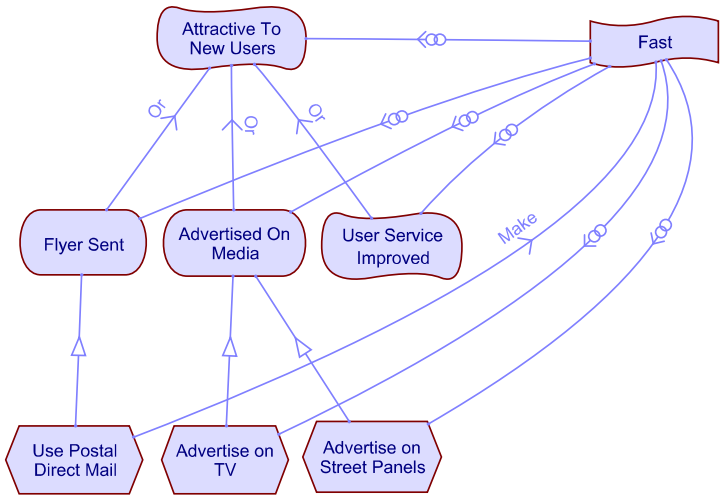


Figure 6-8: Extended Means-Ends Analysis

Task Decomposition:

In Task Decomposition links, tasks are decomposed into sub-tasks, goals, resources and/or soft-goals. One can achieve a task being decomposed only by implementing all the sub-tasks, goals, resources, and soft-goals into which the original task is decomposed. This gives a possibility to delegate a/some children the responsibility to satisfy the quality requirement.

However, quality delegation could become a vulnerable spot of the process. A too pessimistic approach could constrain all the derived items to have at least the same quality requirements as the parent task, while a too optimistic approach might leave some derived items without required quality requirements. This leaves to the modeller the responsibility to make some right judgment to guarantee an optimal choice of decomposition.

In the original i*, there are four principal ways to decompose a task corresponding to soft-goals, goals, tasks, and resources, as shown in Figure A-5.

Figure 6-9 shows an example of a music player who wishes to extend his dictionary from the Internet with the help of an information system. One of the quality requirements for that system is that it must retrieve only legal contents from the Internet. Figure 6-9 suggests a main task "Get Music" on which a "Legality" quality requirement is imposed. "Get Music" is decomposed into subtasks "Download Video" and "Get Author Consent", where "Get Author Consent" is constrained by the "Legality" requirement.

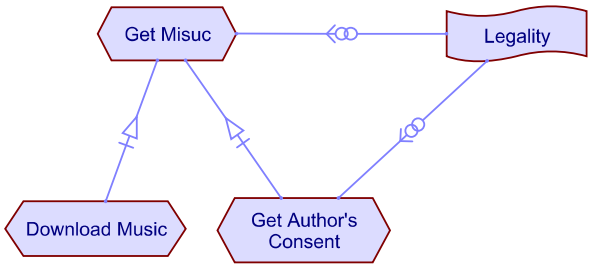


Figure 6-9: Quality delegation in action

6.4.2. Quality requirements refinement

This section applies the discussion in Section 4.2.2 to Tropos.

In the non-functional requirement framework (CHUNG, L. et al., 2000), requirements (soft-goals in NFR framework) can be decomposed using AND or OR operators. In the AND decomposition, a quality requirement is fulfilled if all its sub-requirements are fulfilled, while, in the OR decomposition, a requirement is fulfilled if at least one of its sub-requirement is fulfilled. This suggests a strong correlation with the means-ends and task decomposition analysis presented in the preceding section. This correlation allows us to incorporate the refinement of quality requirements into these analyses in a very natural way.

The previous section (Section 6.4.1) presented a way to propagate quality requirements from a parent node to its children nodes without making any refinement. Here we try to refine the quality requirements in the strategic rationale analysis with the help of Non-Functional Requirement framework. While there are two types of decompositions (i.e AND and OR) for quality requirements, the AND decomposition is usually used in the refinement of requirement and the OR decomposition is usually used in the operationalization process to represent alternative implementations. Since the OR decomposition only gives the alternative operationalizations to fulfil the target quality, there is no quality refinement at all. The means-ends analysis in Tropos includes already this OR decomposition. We only examine the AND decomposition of quality requirement.

The example in Figure 6-10 shows a “Security” quality requirement decomposed into “Integrity”, “Confidentiality”, and “Availability” requirements.

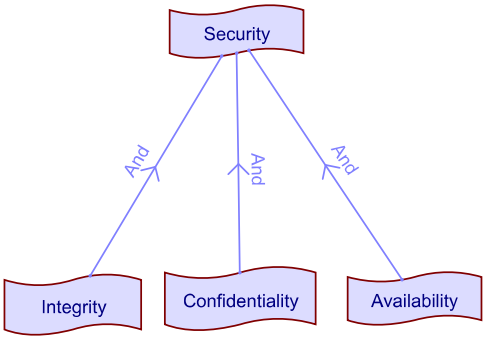


Figure 6-10: AND decomposition of quality requirement

Such a refinement can be applied to the strategic rationale analysis to detail the fulfilment plan of each quality requirement. Concretely, using the AND decomposition of quality requirement together with the task decompositions in the strategic rationale analysis can provide much more detail about how the target task could be carried out while trying to fulfil as many quality requirements as possible.

A condition in which the AND decomposition of quality can be useful is that during the task decomposition, one of the children node cannot fulfil or need not fulfil the whole quality requirement but only a part of it. The remaining part of the quality requirement will be fulfilled by other siblings. In other cases, the quality requirement should be propagated as a whole through the siblings of the target node. This will leave the possibility to fulfil the whole quality requirement later.

We consider an internet phone operator called Phone@Net. In order to offer some discount plan to the most active customers, it depends on an expert to analyze customer behaviour during the last three months. This operation must be carried out in “Security” in the sense that all the private information about its customers must be “Secure”.

Figure 6-11 shows a decomposition of the task “Transfer Customer Calling Statistics” to the expert. It is decomposed into three child nodes: the “Number Scrambled” hard-goal, the “Send Data by Email” task, and the “Calling History” resource. The target task is required to be carried out in “Security”. Knowing the decomposition of the “Security” requirement in Figure 6-10, the goal “Number Scrambled” must be responsible only for making the information “confidential”, the “Calling History log must be complete and

accurate ("integrity") and "available" for sending upon requests from the expert, while the task "Send Data By Email" must still guarantee the whole *security* requirement.

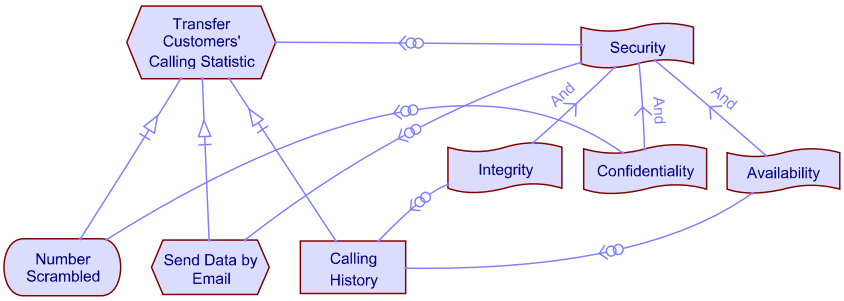


Figure 6-11: Quality requirement decomposition in task decomposition

The above example is a very simple situation where there is only one quality requirement on the target task. When there are two or more quality requirements that contain some conflicts in their fulfilment, one must be careful not to make any decomposition with some conflicts that can be avoided by a different decomposition. The impossibility to have a contradiction-free decomposition means that the system-to-be will not be able to fulfil all the quality requirements. One should leave the contradictory node that cannot be further decomposed until a latter means-ends analysis to choose the best alternatives. The conflict negotiation of quality requirements will be the subject of the Section 6.4.4.

6.4.3. Quality requirement elicitation

In the previous section, quality requirements are propagated and delegated from the parent nodes to the child nodes in means-ends analysis and task decomposition. As a consequence, all quality requirements on the child nodes are, in fact, requirements on the parent nodes. No new qualities are elicited though these analyses.

For an experienced software designer, this process may not provide any help, since he can probably foresee all the quality requirements, apply them to strategic dependencies, and propagate them to derived tasks, goals and resources. Even though this analysis is legal, it does not reflect well the intentional evolution during the software development process.

In Tropos, foreseeing all the quality requirements on the system-to-be in the early requirement phase is unrealistic, since the system-to-be has not been explicitly introduced yet. The objective of this early

step is to emphasize the need of the system-to-be to satisfy intentions of all social actors. Moreover, quality requirements, in the form of soft-goals, are introduced only from strategic dependencies between social actors of the system. The subsequent strategic rationale analysis only refines all the dependencies identified during the strategic dependency analysis, and thus contains only the existing quality requirements. Adding additional quality requirements to support existing goals, tasks and resources using means-ends and/or task decomposition analyses is possible but not enough and not natural, since:

- Quality requirements are neither a possibility for satificing the “end” in the means-ends analysis nor a sub-task in the task decomposition.
- Child nodes may be required to have different quality requirements to contribute to the fulfilment of the parent node.
- Responsibility of quality assurance is not well assigned. This could result in over-spreading quality requirements, hence insufficient treatment to achieve quality objectives.

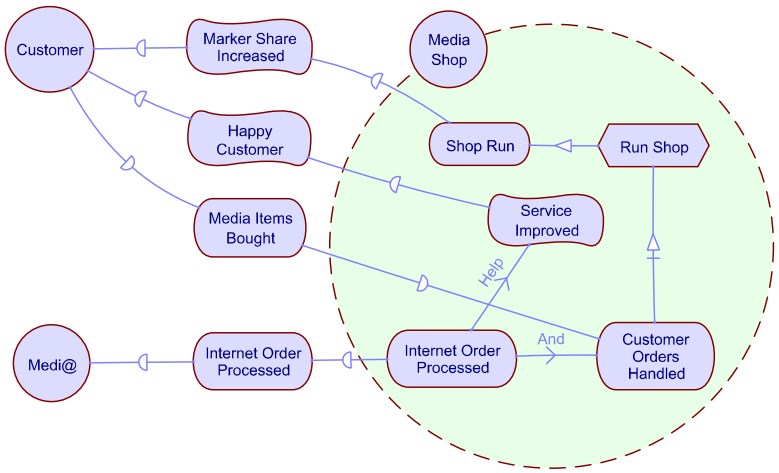


Figure 6-12: Strategic rationale analysis (Early requirement) of Media Shop using Tropos

There is also a tendency in the literature to introduce quality soft-goal dependencies in the strategic dependency analysis of the late requirement phase. This is a very *forcing* treatment to the lack of some quality requirements on the system-to-be during the analysis process. Taking a closer look at these quality requirements, it seems clear that they could have been elicited by the means-ends analysis

or the decomposition of tasks in the strategic rationale model of the early requirement phase.

The mechanics presented in this section allows us to add quality requirements where they arise effectively and to enhance coherence between successive phases of the development process.

To see the improvement from the Tropos analysis, we compare the analyses of a Media Shop by (CASTRO, J. et al., 2002) to our proposed analysis. So far, this shop only sells media items in its shops or by telephone. To compete with other shops in the market, it intends to open an online store, Media@. A portion of its strategic rationale in the early requirement taken from (CASTRO, J. et al., 2002) is in Figure 6-12.

In the next step, the strategic dependency is carried out for the later requirement phase, where the actor “Media@” plays the central role. This is shown in Figure 6-14 where three additional quality requirements (soft-goals in Tropos) are introduced to the model: “Security”, “Availability” and “Adaptability”. These quality requirements are added here as dependencies between the actors. See (CASTRO, J. et al., 2002) for more details about the original analyses.

However, as we discussed earlier, quality requirements should be elicited from the refinements of goals, tasks or resources, except for very special cases where quality requirements are implied and can be added directly to the dependencies such as legal or environmental requirements. The additional quality requirements must, nevertheless, be compatible with other refinement processes. And if they are left undiscovered, other refinement processes would help to elicit them correctly. To illustrate this, we revise the preceding two analyses (using Tropos) in Figure 6-12 and Figure 6-14 in our quality-aware framework (using QTropos).

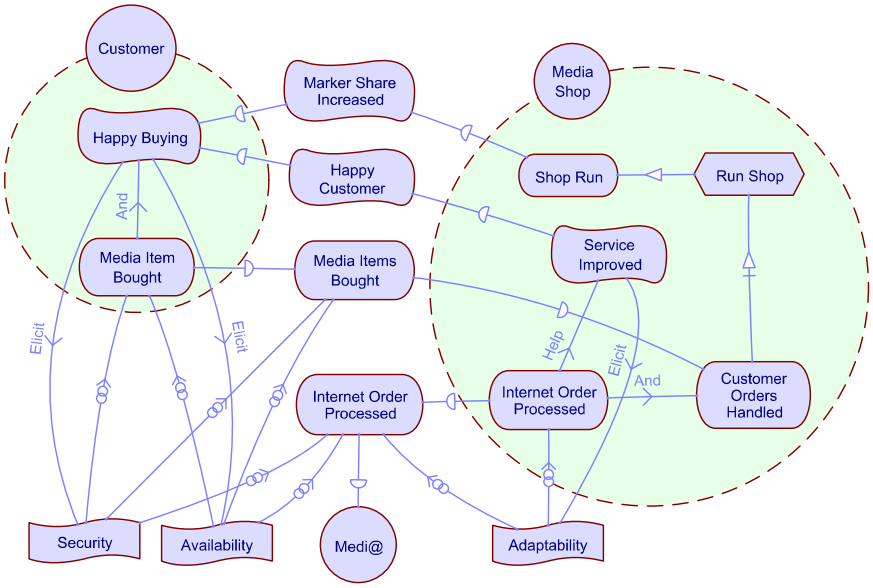


Figure 6-13: Quality-aware Strategic rationale analysis (Early requirement) of Media Shop using QTropos

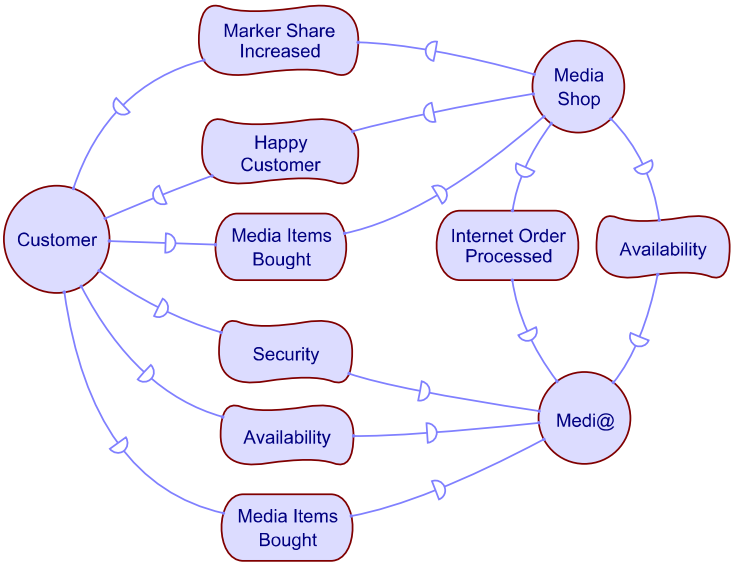


Figure 6-14: Strategic dependency analysis (Late requirement) of Media shop using Tropos

In Figure 6-13, we detail the “Customer” actor to reveal the “Happy buying” soft-goal on which, soft-goals “Market Share Increased” and “Happy Customers” are dependent. The soft-goal “Happy Buying” is then supported by the task “Buy Media Item”, which are required to be “Secure” and “Available” whenever the customer wants to make a purchase. The two quality requirements “Security” and “Availability”, elicited to support the “Happy Buying” soft-goal, are then propagated through the dependencies and refinements of goals and tasks and are finally required by the goal dependency “Internet Orders Processed” from “Media Shop” to “Medi@”. All the nodes used in the propagation are required to have these two requirements as well.

Furthermore, in the actor Media Shop, the “Internet Order Processed” hard-goal depends on the actor Medi@ through the goal dependency “Internet Order Processed”. Since, Medi@ does not exist yet, the “Internet Order Processed” hard-goal help to have “Service Improved”. And it is even further improved if the management of the internet store is *Adaptable* with the evolution of the market. This is why the “Adaptability” requirement is elicited from the soft-goal “Service Improve” and constrains “Internet Order Processed” hard-goal.

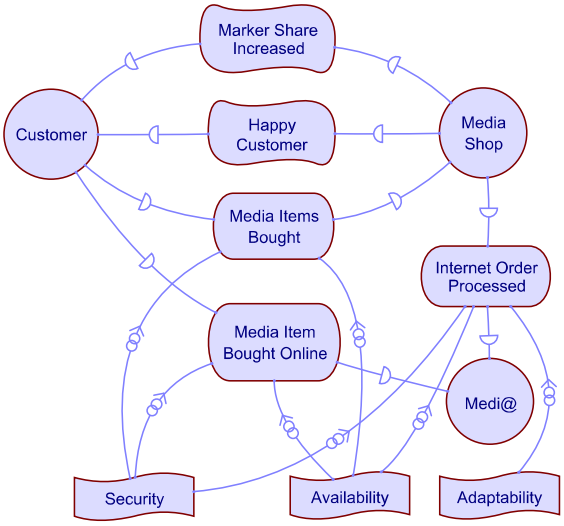


Figure 6-15: Quality-aware Strategic rationale analysis (Late requirement) of Media Shop using QTropos

The above developments explain the presence of some quality requirements in the subsequent strategic rationale analysis at the late requirement phase shown in Figure 6-15. Given the development of quality requirements in the previous step, all the quality requirements are inserted into the strategic rationale analysis without any ambiguity. Note that the quality requirements “Security”

and “Availability” constraining the goal dependency “Media Items Bought Online” from the “Customer” to the actor Medi@ are “trivially” made in accordance to the goal dependency Media Items Bought to the actor Media Shop. However, if they are not added here, they will be correctly added by through the task “Buy Media Items” in actor “Customer” later in the subsequent strategic rationale analysis, as done earlier.

The above example shows a substantial improvement of QTropos over Tropos in dealing with quality requirements. We are now able to fill all the life cycle of quality requirements: from being elicited, evolving (by propagation and assignment) to begin fulfilled. To have a full account of a quality requirement, we can extract the Quality analysis view from all the analysis by keeping only the nodes and links constrained by the quality requirement together with all corresponding qualification links and quality contribution links. This can also be done with respect to each quality requirement for a better follow-up, as done in Figure 6-16 for the “Availability” requirement.

Figure 6-16 is extracted from Figure 6-13. It focuses only on “Availability”, which begins by its elicitation from the “Happy Buying” soft-goal of the “Customer” actor. It shows also the transfer of that quality requirement from the “Customer” actor to “Media Shop” actor and to the “Medi@” actor and from the “Media Item Bought” hard-goal to “Customer Orders Handled” hard-goal and to the “Internet Orders Processed” hard-goal.

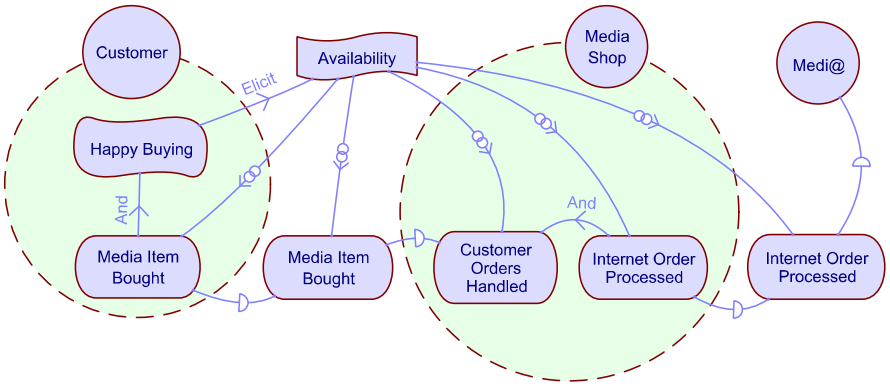


Figure 6-16: View on availability requirement

The views on quality requirements can be created anytime during the development process.

6.4.4. Quality conflict negotiation

Section 4.2.5 proposed a way to negotiate conflicts that can arise in the fulfilment of various quality requirements. The proposed method consists in defining numerical priority levels for the order in which conflicting quality requirements are taken into account. This method can be adapted to the analyses of QTropos by allowing alternatives in the Means-Ends analyses to be compared by the same technique.

We consider a slightly more complicated example than that in Section 4.2.5 where there are four quality requirements and two alternatives from the Means-Ends analysis of the "Customer Support Offered" hard-goal of a Computer Shop that sells computer devices in a city. The shop owner wants to offer his own support services to his clients to solve computer problems related to their purchased products.

There are two options for offering support to customers: "Send Technical" support to the customer site to solve the problem or "Offer Online Support" in the shop website with a knowledge base continuously updated. For each option, a software module will be integrated into the existing software system.

The owner wants to offer both options but decides to start with only one, to reduce costs. Developers help him to select the option to be implemented first by examining both options under four aspects: efficiency, ease of use, cost, and availability.

Figure 6-17 shows a Means-Ends analysis that resolves the "Customer Support Offered" hard-goal using two alternative tasks: "Send Technical" and "Offer Online Support". The hard-goal and its two alternative tasks are constrained by four quality requirements: "High Efficiency", "Easy for Customer", "Low Operational Cost", and "High Availability".

The contribution links are drawn and assessed to assign contribution levels as shown in Figure 6-17. Without priorities, there is a tie between the two alternatives with one *Make link*, one *Help Link,* and two *Hurt Links.* With priorities for quality requirements as indicated in the figure, "Easiness for Customer" is preferred over the "Low Operational Cost". As a result, "Send Technical" is preferred than "Offer Online Support".

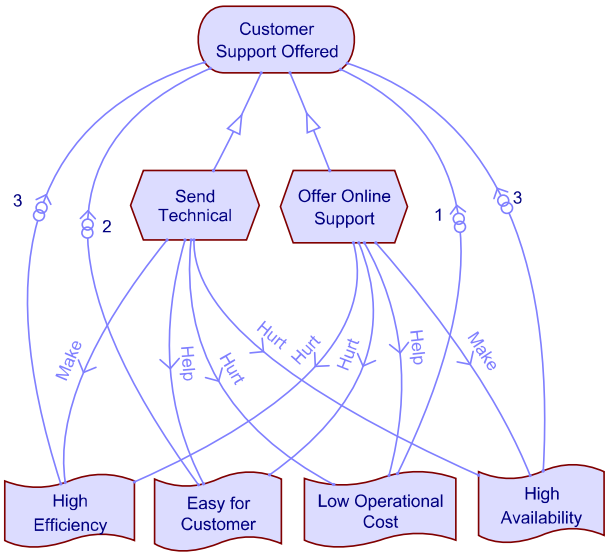


Figure 6-17: Conflict negotiation

One can also think of other labels on contribution links than those of i* and Tropos (*Break/Hurt/Help/Make)* to better cope better with quality conflicts. A finer numerical analysis can also be devised for example. A more detailed discussion about this could be interesting but this is out of the scope of this thesis.

6.5. Architectural Design of QTropos

In QTropos, the Architecture Design phase is built upon that of Tropos. Information about quality requirements from the previous phases can also be propagated to this phase in greater details.

In the original Tropos, quality requirements are present at this phase only in a list that is used to choose among organizational style for grouping elements, as presented in Section A.2.3.

In QTropos, much more information is retained since most of the analyses from earlier phases can be included. Because quality requirements are modelled in a separate dimension and by a separate set of notations than those of other elements, their influence is increased on the selection of design options without disturbing other analyses.

The choice of the organizational style for the system-to-be is based solely on the list of quality requirements identified during the earlier phases. This was already addressed in original Tropos, as described in Section A.2.3. Quality requirements raise other issues in Architectural Design. We give some details about how to incorporate

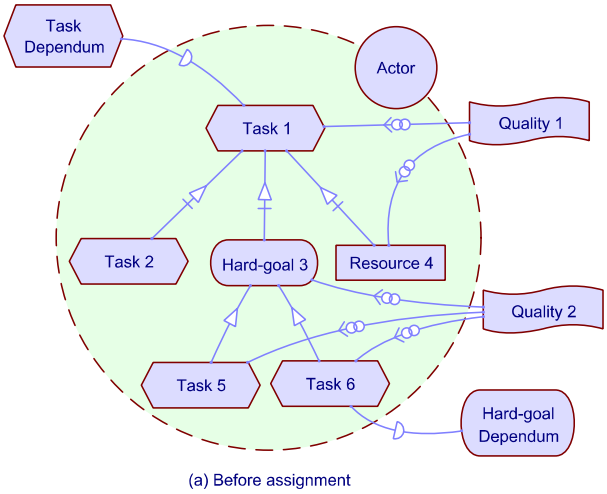
quality requirements in the decomposition into subsystems and also in the decomposition of each subsystem using social patterns.

6.5.1. Sub-system decomposition

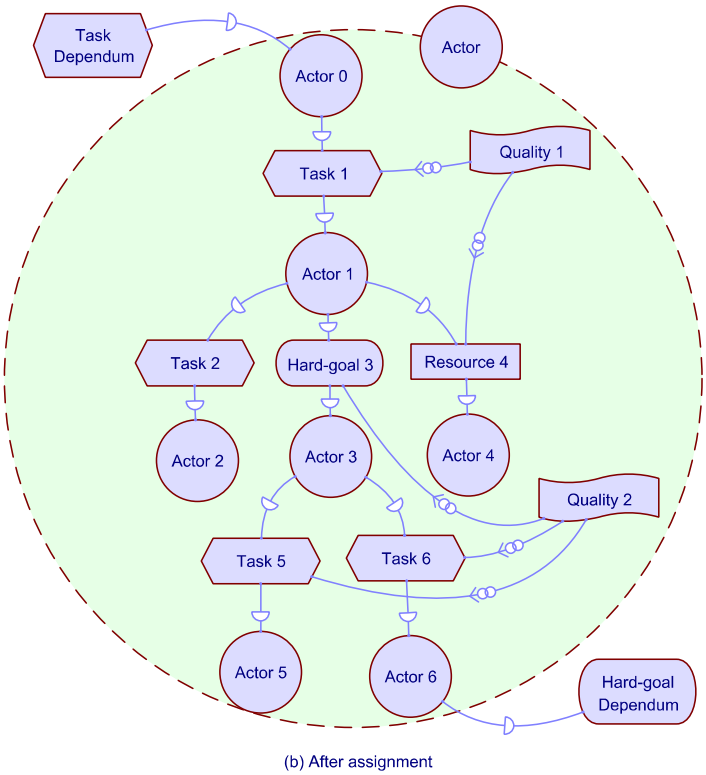
From the Late Requirement phase, we have identified all the goals, tasks and resources. In order to divide the system into sub-systems, one can use a bottom-up approach to first transform each node of the strategic rationale analysis into a dependency with the corresponding type. Additional sub-actors are also added to complete the newly created dependencies. This procedure is described in Section A.2.3.

Suppose that a part of the strategic rationale analysis of the late requirement is as presented in Figure 6-18(a). The dependencies to task *Task 1* and from task *Task 6* involve participants outside of the system. A coarse decomposition of the system is shown in Figure 6-18(b). Note that all the quality requirements have been retained in place At this stage, we can consider that each resulting actor is a subsystem. In QTropos, we identify two possible operations that the developers can sequentially carry out:

- Merging: if two or more actors play the same role or possess the same capability or resource, they can be merged into one actor. The merging is carried out only if the implementation of resulting actor is not more difficult than that of the original actors. This condition has to be respected in order to guarantee the feasibility of the resulting actor.
- Grouping: the selection of organizational style results in some predefined subsystems (actors). The developer can group actors obtained after the merging step into subsystems defined by the organizational style.



(a) Before assignment



(b) After assignment

Figure 6-18: Coarse decomposition of system-to-be

Figure 6-19 depicts the results of the merging and grouping operations by which the actors “Actor 3” and “Actors 4” from Figure 6-18 are merged into “Actor 34”. Then “Actor 1”, “Actor 2” and “Actor 34” are grouped into “Subsystem 1”. Likewise, “Actor 5” and “Actor 6” are merged into “Subsystem 2”. Note that, these operations are carried out inside the boundary of the system-to-be actor.

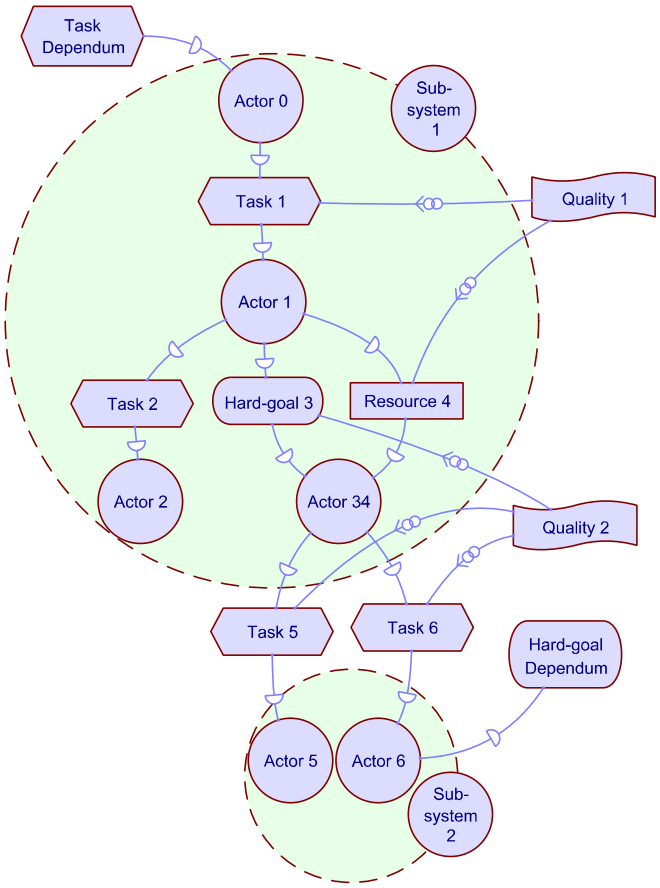


Figure 6-19: Archtectural design - Subsystem decomposition

It is obvious that the above decomposition and operations do not split or merge nodes (hard-goals, tasks, and resources) from the strategic rationale analysis to create new nodes. Therefore, the quality requirements on these nodes are kept unchanged.

An important remark is that soft-goals should not appear at this stage, the old notion of quality soft-goal is replaced by that of quality requirement. The existence of a soft-goal at this stage would imply that the early and late requirement analyses were not carried out properly.

6.5.2. Sub-actor design using social patterns

The next step of the architectural design is to detail all the sub-actors using a catalogue of existing social patterns specially designed for multi-agent system, for examples (DO, T., 2005) and those introduced in Chapter 5. However, with additional quality requirements, one should review the result of the application of patterns in order to propagate and assign necessary quality requirements for the interactions (services) between added agents.

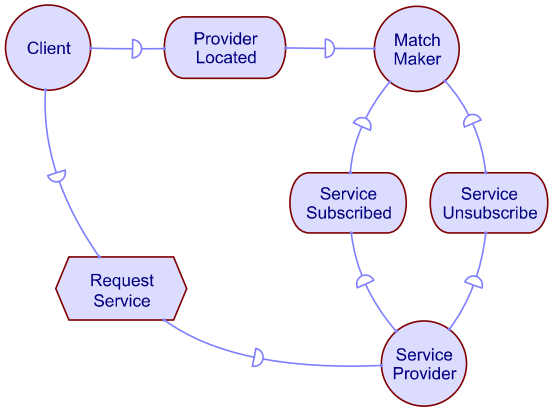


Figure 6-20: Match Maker pattern

Figure 6-20 presents the Match Maker pattern designed for multi-agent system (DO, T., 2005). The main agent that controls all the activities is agent "Match Maker". The "Client" depends on the "Match Maker" to find a "Provider" for a specific service. When an appropriate "Service Provider" is located, the "Client" can request the desired service from that "Provider". "Service Providers" can also make themselves available to the "Match Maker" by subscribing their services. They can unsubscribe their services when needed.

To illustrate the use of this pattern, especially when there is a quality requirement, we consider the following example. In a document management system, there are some "PCs", a "Print Manager" and several "Printers". "PCs" depend on the "Print Manager" to print documents with "High Availability" (i.e., small delay) and "High Customizability" (i.e., with desired printing quality). The "Print Manager" depends on the "Printers" to queue printing tasks. The "Print Manager" does not know when a "Printer" becomes available and which resolution a "Printer" can offer. Therefore, we use the "Match Maker" to build the actor "Print Manager".

In Figure 6-21, the top diagram is a part of the system decomposition of the document management system with three sub-actors: "PC", "Print Manager" and "Printer". The bottom diagram is the revised

version of the top diagram where “Print Manager” is decomposed using the pattern “Match Maker”.

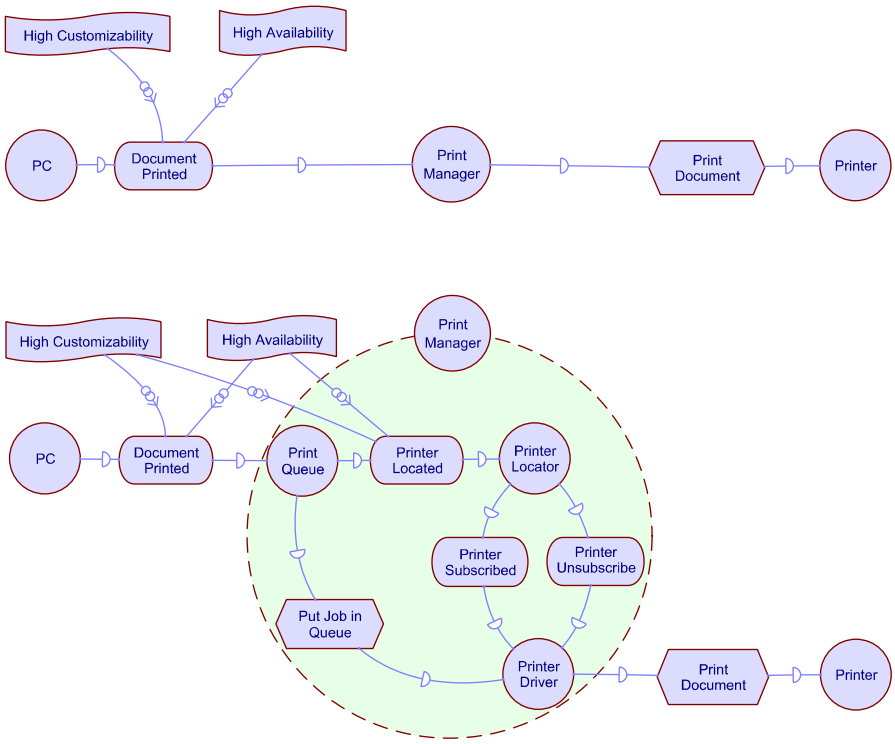


Figure 6-21: Match Maker pattern with quality requirement

In the revised diagram, each “Printer” has a “Printer Driver” that plays the role of a software interface to the physical “Printer”. This “Driver” is responsible for subscribing/unsubscribing the “Printer” to the “Printer Locator”. Possessing a list of subscribed “Printers” together with their service description, the “Printer Locator” chooses the “Printer” with the highest resolution among all free and working Printers and forwards its identity to the requesting “PC”. The editor can now request the printer driver to put a printing task to the queue of the “Printer Driver”.

The quality requirement “High Availability” and “High Customizability” constrain the hard-goal dependency “Document Printed” from actor “PC” to actor “Print Manager”. Dependee actor “Print Manager” must be responsible for fulfilling “High Availability” and “High Customizability”. This is why the requirements are propagated into the new “Printer Manager”. When a sub-actor is detailed, e.g. “Printer Manager”, quality requirements are propagated inside that sub-actor. Responsibility is assigned to dependencies that can fulfil the requirement. Here, the “Printer Located” hard-goal is

made responsible for both quality requirements. The "Printer Locator" can choose the "Printer" that can offer the desired printing quality and the shortest delay among available ones to fulfil the qualified hard-goal dependency.

This example demonstrates some advantages of QTropos over Tropos in the treatment of quality requirements. In Tropos, quality requirements in Architectural Design are used only to select organizational styles and are excluded from later phases. In QTropos, quality requirements continue to influence the choice of social patterns (e.g. *Match Maker pattern*) and even the detailed design of services (e.g. *High Resolution Printer Located*). This "Print Manager" example is taken from the case study that is further elaborated in Chapter 7.

6.6. Detailed Design of QTropos

The architecture design contains the last decomposition into the finest level of the system. The system-to-be is introduced as an actor at the beginning of the early requirement phase. This system actor is decomposed into subsystems (sub-actors) and then to agents during the architectural design. In Tropos, agents are the smallest containers that contain functionalities, plans of actions, and capabilities. Now, the detailed design phase is for describing these components of every agent inside the system and specifying protocols for agent interaction. Like Tropos, we adopt the Agent UML language (AUML) (CABAC, L. and Moldt, D., 2005) to model agent components and interactions.

The presence of soft-goals at this stage would show that the earlier analyses have not been completed. But hard-goals are allowed, for example those in predefined patterns for which the means to achieve are already described in the pattern document. Other examples are hard-goals that represent intermediate dependences whose dependee knows how to achieve it. For example, the "Document Printed" hard-goal in Figure 6-21 is materialized by the task "Print Document" executed by "Printer".

One important point that influences the outcome of this phase is the target agent model. The chosen model defines the necessary components that must be implemented for agents. One of the most used is the *Belief-Desire-Intention (BDI)* software model supported by many agent-oriented programming paradigms such as JACK (JACK, Agent Oriented Software Pty. Ltd., 2002), JADEX (POKAHR, A. et al., 2005). A fourth notion in the *BDI* model is that of *Plan*. Plans are alternatives for agents to achieve their goals (*desires*) and they are left for the designer or programmer to define. Consequently, in the

class diagram, a class of agents has an additional part that contains *plans*.

In QTropos, the architectural design results in a set of agents, with their responsibility described by dependencies and quality requirements. For each agent, there is a set of elements (hard-goals, tasks, and resources) that it has to offer or realize and another set of elements that it depends on other agents to obtain. Each dependum can be modified by some quality requirements, as shown in Figure 6-22.

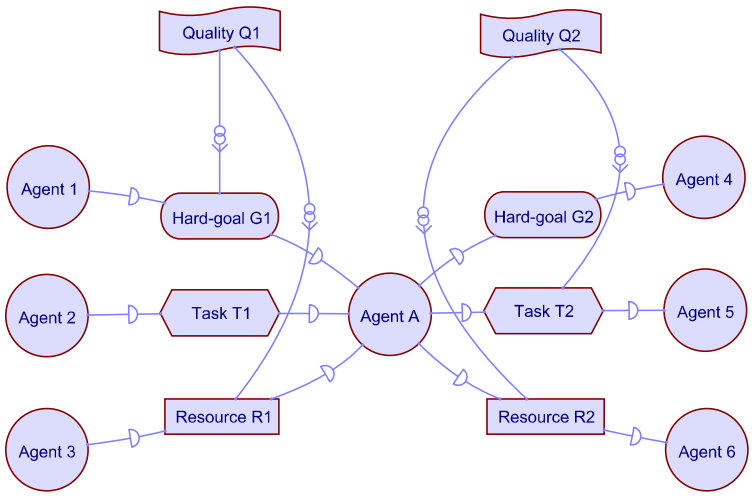


Figure 6-22: Typical agent with quality requirement

The corresponding class diagram of "Agent A" in AUML should reflect the above agent dependency diagram. The most natural mapping is done by the following rules:

- Goals to be realized by the agent (e.g. "Hard-goal G1") are implemented by one of the plans in the class diagram. Chosen plans inherit all quality requirements (e.g. "Quality Q1") from the realized goal.
- Tasks to be carried out by "Agent A" (e.g. "Task T1") are transformed in to operations in the class diagram together with their quality requirements.
- Resources to be offered by the agent (e.g., "Resource R1") are stored as agent attributes. The corresponding attributes must satisfy the quality required by the resource (e.g., "Quality Q1").
- If an operation of the agent' class is used in a plan that is required to have some qualities, these qualities are also required for the operation.

- To complete the quality coverage, one should take into account some quality requirements that may be required for the agent when requesting other agents to carry out a task (e.g., “Task T2”), to obtain a goal (e.g., “Hard-goal G2”) and/or to acquire a resource (e.g., “resource R2”). The main responsibility, in those cases, remains at the agent partners.

Suppose that “Agent A” uses “Plan1ForG1” or “Plan2ForG1” to obtain “Hard-goal G1”. The plan “Plan1ForG1” uses “PlanP” and “PlanP” requests “Agent 5” to carry out “Task T2”. “Agent A” uses “Operation1” to carry out “Task T1”. The resulting class of the “Agent A” is presented in Figure 6-23.

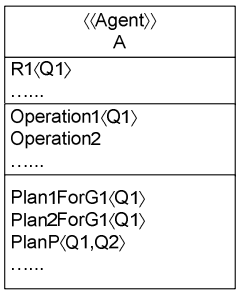


Figure 6-23: Class diagram with quality requirements

We use angle brackets to specify qualities required for elements of a class. In Figure 6-23, R1⟨Q1⟩ means that Resource R1 is required to have quality requirement Q1, PlanP⟨Q1, Q2⟩ means plan PlanP is required to have qualities Q1 and Q2. These notations help the developers to design each element using the best option to fulfil the required qualities.

In each agent, plans are very important to satisfy a goal. To specify a plan, one can use AUML templates and packages as well as sequence and collaborations diagrams. The most intuitive is the sequence diagram used at different levels to reveal the interaction context for each plan. At the lowest level, particular processes in the sequence diagram can be detailed through plan diagrams. In QTropos, each plan has a set of quality requirements to be taken into account when it is detailed in the sequence and plan diagrams. In general, how agents interact with each other is left for the designer to decide to insure the best way to fulfil quality requirements. Social patterns, like those described in Chapter 5, can also be used to monitor and control the fulfilment of quality requirements.

Chapter 7 QCASE – TOOLS FOR QUALITY REQUIREMENTS

Earlier chapters have elaborated on how to better exploit the notion of quality requirements in the goal-based paradigm in general and in the Tropos methodology in particular.

A common thread of some of the developments in this thesis is based on the clear separation of quality requirements from soft-goals. Since this has not been considered in other research, there are no editing and analyzing tools for it.

In order to be able to bring these new analyses into real projects, we developed a CASE (Computer-Aided Software Engineering) tool that we named QCase, for analyzing qualified goals, i.e., goals constrained by quality requirements. An early version of this tool was described in (HOANG, T.T.H and Kolp, M., 2010). QCase can be considered as a prototype of a more complete and sophisticated tool or of an add-on to an existing development environment. Although developing a complete product was not a main objective of our thesis project, QCase is implemented with almost all the principal functions of a normal CASE tool. Our final implementation aims at the following targets:

- To support all relevant developments in this thesis, mainly of Chapter 4 and Chapter 6, especially the relations between quality requirements, hard-goals, and soft-goals. Quality requirements are given the constraint (or qualifier) role alongside hard-goals and soft-goals and the central role inside QCase.
- To provide a proof of concept for the goal-oriented analysis of quality requirements. The design is extendible to other top-down goal-based methodologies. Two chosen examples of model, which were also conceptually developed in this thesis,

are Goal Analysis with Quality Requirements (Chapter 4) and QTropos (Chapter 6). The refined goal analysis in Chapter 4 builds upon some basic ideas of the well-known KAOS methodology.

- To provide a graphical interface for editing and analyzing quality requirements. The final models or diagrams can be printed or exported into text editors for high quality printing.

The implementation of QCase allowed us to replace all the figures in this text by vector figures that are generated to be scalable to higher resolution for a better printing.

Since the concept of constraints on functions already exists in some methodologies, we use "qualifiers" as the superset of quality requirements and functional constraints. Section 7.1 describes the general concepts and analysis which is used to specify the functions of QCase. We do not distinguish quality requirements and other types of constraints: we treat all of them as qualifiers. Still, the tool is largely inspired by our main proposals in this thesis. Section 7.2 applies the general concepts and analysis to design the implementation of the refined goal analysis. Section 7.3 shows how QTropos can be realized in QCase. Finally Section 7.4 summarizes the architecture and implementation of QCase.

7.1. General concepts

We base our tool on more general concepts than agent, actor, goal, soft-goal, etc. After studying the methodologies listed in Appendix C and given our proposed analysis of quality requirements in Chapter 4 and Chapter 6, we decide to build the prototype upon the following two generic concepts:

- *Nodes*: are all self-defined items that are used for modelling different kinds of requirements and specifications, for example, actors, agents, soft-goals and hard-goals, tasks, resources, and qualities in Q-Tropos.
- *Links*: are meaningful connections between nodes. They are the four types of dependency, decomposition links, contribution links and qualification links in Q-Tropos.

Nodes can be further classified into the following behavioural classes:

- *Containers*: are items that can contain other items. In Tropos, examples are actors and agents that can contain tasks, resources, soft-goals, hard-goals, sub-agents.

- *Intentional items*: are items that represent the behaviour of actors and agents. They can be contained inside containers such as tasks, resources, soft-goals, hard-goals in Tropos and Q-Tropos.
- *Qualifiers*: are items that constrain other items. Usually, these items cannot be contained in other items. For Q-Tropos, they are quality requirements. In other methodologies, qualifiers can be numerical constraints, functional constraints, etc. on other item as well.

Our graphical diagrams are represented in the form of directed graphs in which there are nodes and directed links between them. A *model* $M = (N, L)$ is a graph that is composed by two sets: N is the set of typed *nodes* and L is its set of typed *links*. This graph contains subtrees that represent top-down analyses of some components of the future system.

A diagram of model M is a sub-graph of M together with some layout properties for rendering its graphical layout. From a model, one can have as many diagrams as all possible sub-graphs of it.

A typed *node* is characterized by the following attributes:

- $node.type$: is the type of the node. $node.type$ lies in a fixed set (namely T_N) determined by the chosen methodology;
- $node.behavior$: is one of three values $container$, $intentional$, or $qualifier$;
- $node.name$: is a displayed textual definition of the node;
- $node.notes$: is a explaining text of the node, usually used to clarify the meanings of $node.name$.
- $node.container$: node that contains the current node;
- $node.labels$: is a set of additional labels that are assigned to the node. The set of all possible labels that can be assigned to a node is determined by the chosen methodology and by the type of that node;
- $node.parent$: is the node from which *node* is derived. A *node* with a null parent is called free node;
- $node.children$: is the set of other nodes that are derived from *node*;
- $node.inlinks$: is a subset of L containing links that enter the *node*;

- $node.outlinks$: is a subset of L containing links l that go out of the $node$.

A typed $link$ is a characterized by the following attributes:

- $link.type$: is the type of the link. $link.type$ lies in a fixed set (namely T_L) determined by the chosen methodology;
- $link.name$: is a (displayed) textual definition of the link;
- $link.notes$: is a clarification of $link.name$, when needed;
- $link.labels$: is a set of additional labels that are assigned to the link. The set of all possible labels that can be assigned to a link is defined by the chosen methodology and by the type of that link;
- $link.from$: is the node which link comes;
- $link.to$: is the node to which the link goes.

From the above definitions, we can derive some simple relations:

$$\forall n \in N: n.inlinks = \{l \in L | l.to = n\}$$

$$\forall n \in N: n.outlinks = \{l \in L | l.from = n\}$$

We will use $l(a, b)$ as a constructor of the link named l from the node a to the node b.

For each methodology, the following logical predicates must be defined:

- $linkable(linktype, from, to)$: has truth value TRUE only if it is legal to create a link l such that $l.type = linktype$, $l.from = from$ and $l.to = to$. This predicate is one of conditions that must be satisfied before a link of type $linktype$ is about to be created between two nodes: $from$ and to. This is also used in the validation of the model.
- $QForwardable(link)$: has truth value TRUE only if every qualifier constraining the $link.from$ of the current node can be transferred to the $link.to$. This predicate is used to automatically propagate and trace quality requirements.
- $QBackwardable(link)$: has truth value TRUE only if every qualifier constraining the $link.to$ of the current node can be transferred to the $link.from$ node. This predicate is used to automatically propagate and trace quality requirements.

One of the reasons why we have to allow qualifiers to be transferred backward or forward through a link is that, in methodologies like Tropos or QTropos, the directions of link symbols in the graphical

representation do not indicate the qualifier transfer direction. On some links like the dependency, quality requirements can only be transferred along the link direction while on the means-ends link, quality requirements can only be transferred in the opposite direction of the link direction. It is useful to notice that $QForwardable(l)$ and $QBackwardable(l)$ cannot have truth value $TRUE$ simultaneously, since this would create a cycle in the satisfaction of quality requirements.

In general, the above predicates are evaluated according to the actual situations in which the existence of some elements can make those predicates TRUE or FALSE. Concerning the relation of parent node and child nodes in the fulfilment of qualifiers, predicates $QBackwardable(\ )$ and $QForwardable(\ )$ define a tree of transfers of each qualifier, which is independent of the node decomposition tree.

Based on those qualifier transfer trees, we say that qualifier q is *fully considered* at node n, i.e. $FullyConsidered(q,n)$, if at least one of the following conditions is fulfilled:

- n is not constrained by q.
- q is correctly transferred to its child nodes of n such that the fulfilment degree of q at constrained child nodes determines the fulfilment degree of q at n. The satisfaction of this condition depends on the way n is decomposed into child nodes. For example, OR decomposition (cfr. Chapter 4) requires that all child nodes are constrained by the same qualifiers of the parent node.
- It can be fully justified that q is satisficed or dissatisfied at n. The satisfaction of this condition is determined by the contents of n. For example, Encrypting Data node makes Confidentiality qualifier satisficed.

We say that qualifier q is fully considered in model $M = (N, L)$, i.e. $FullyConsidered(q)$, if and only if q is fully considered at every node $n \in N$, $FullyConsidered(q,n)$. This can be represented by the following formal relation:

$$FullyConsidered(q) = \bigwedge_{n \in N} FullyConsidered(q,n)$$

Again, the $FullyConsidered$ predicate is defined with respect to specific methodologies. It is very important to note that, if a qualifier is fully considered (in the whole model or at a node), this does not mean that the qualifier is satisfied. In fact, this says that we have enough evidence to determine the satisfaction level (either positive or negative) of the qualifier.

To help analysts keep track of qualifiers, we also define the following working sets (qualifier status sets) of each $qualifier$ in a model:

- C_q (Constrained Set): set of nodes that are constrained by qualifier q;
- A_q (Active Set): is a subset of C_q which is not fully considered;
- P_q (Pre-constrained Set): set of nodes that have not been constrained by qualifier q yet, but it is mandatory to consider whether they are constrained by qualifier q or not;
- S_q (Suggestion Set): set of nodes that have not been constrained by qualifier q yet and that should be constrained by qualifier q. However, there is no consequence if these nodes are not constrained by qualifier q.

A qualifier q that is fully considered must have empty A_q and empty P_q. The set C_q is usually specified by the analyst while the three others can be deducted from the qualifier transfer tree.

Indeed, we extract all the nodes $n \in C_q$ for which $FullyConsidered(n) = FALSE$ to construct A_q. To construct S_q, for each node $n \in C_q \backslash A_q$, we add all the nodes $n_1 \in N \backslash C_q$ such that $l(n, n_1) \in L$ (or $l(n_1, n) \in L$) and $QForwardable(l)$ (or $QBackwardable(l)$). And to construct P_q, for each node $n \in A_q$, we add all the nodes $n_1 \in N \backslash C_q$ such that $l(n, n_1) \in L$ (or $l(n_1, n) \in L$) and $QForwardable(l)$ (or $QBackwardable(l)$).

These sets allow the tool to automatically check if a qualifier is fully considered or not in a model when $FullyConsidered$ are defined in specific situations.

The last issue that must be elaborated here is the check for the existence of cycles. As pointed out earlier, this prototype is designed for top-down analysis. Therefore the additional concept of qualifiers must also follow this scheme. Since we can extract for each qualifier a sub-graph, determined by C_q, $QForwardable(\ )$ and $QBackwardable(\ )$, we must ensure also that this sub-graph contains no cycle and that this sub-graph is actually a tree. Formally, we cannot have a list of nodes $n_1, n_2, \dots, n_k \in C_q$ such that $n_k \equiv n_1$ and for all $i = 1, \dots, k-1$ one of the two following conditions hold:

1. $[l(n_i, n_{i+1}) \in L] \wedge QForwardable(l)$
2. $[l(n_{i+1}, n_i) \in L] \wedge QBackwardable(l)$

One can modify standard methods for cycle detection in directed graphs from graph theory to carry out this check, see for example (KNUTH, D.E., 1997). If no cycle is found, there will be no circular dependence in the satisfaction of qualifiers.

In Section 7.2, we define the above notions for the refined goal analysis with quality requirements presented in Chapter 4.

7.2. Implementation of the Refined Goal Analysis

In our general goal analysis, presented in Chapter 4, we focus only on three types of nodes: Hard-Goal (HG), Soft-Goal (SG) and Quality Requirement (QLY) and five types of links: And (AND), Or (OR), Elicitation (ELICIT), Qualification (QUALIFY) and Contribution (CONTRIB) links. The accepted labels for CONTRIB links are: Make, Help, Unknown, Hurt and Break. QUALIFY links define the set of constrained nodes C_q.

$$T_N := \{SG, HG, QLY\}$$

$$T_L := \{AND, OR, ELICIT, QUALIFY, CONTRIB\}$$

$$GOAL := \{SG, HG\}$$

$$C_q := \{n \in N | \exists l \in L: (l.type = QUALIFY) \wedge (l.to = n)\}$$

$$\forall l \in L: (l.type = CONTRIB) \rightarrow (l.label \in \{Make, Help, Unknown, Hurst, Break\})$$

We define the *linkable* predicate to determine if a link of a specified type can be created between two specified nodes. The following *linkable* predicates summarize what are defined in Chapter 4. We also know that the direction of AND and OR links indicate the contribution from the child node (from) to the parent node (to). However, the Quality Requirements can only be transferred from the parent node to the child nodes, hence in the opposite direction to the link direction. Since quality requirements can only be transferred through AND and OR link, the predicates *QForwardable* and *QBackwardable* are rather simple to be defined. The details of these three predicates *linkable*, *QForwardable* and *QBackwardable* are given in the box below. The description of those predicates describes only what was said in Chapter 4: AND and OR links are used to refine GOAL nodes or QLY nodes; QUALIFY links connect QLY nodes to GOAL nodes; ELICIT links connect a SG node to its derived QLY nodes; and CONTRIB links connect GOAL nodes to QLY nodes whose satisfaction is contributed (negatively or positively) by the GOAL nodes.

It is natural to forbid that a node be decomposed by using both AND links and OR links. This should be controlled by the tool every time

$$linkable(AND, from, to) \coloneqq [(from.type = QLY) \wedge (to.type = QLY)] \vee [(from.type \in GOAL) \wedge (to.type \in GOAL)]$$

$$linkable(OR, from, to) \coloneqq [(from.type = QLY) \wedge (to.type = QLY)] \vee [(from.type \in GOAL) \wedge (to.type \in GOAL)]$$

$$linkable(ELICIT, from, to) \coloneqq (from.type = SG) \wedge (to.type = QLY)$$

$$linkable(CONTRIB, from, to) \coloneqq (from.type = HG) \wedge (to.type = QLY)$$

$$linkable(QUALIFY, from, to) \coloneqq (from.type = QLY) \wedge (to.type \in GOAL)$$

an AND link (or an OR link) is added.

As we pointed out in the previous subsection, to correctly trace the consideration state of quality requirement, we must also define the predicate $FullyConsidered(q, n)$.

In this general goal analysis, since CONTRIB links are used to show how hard-goals contribute to the fulfilment of quality requirements and since we only have two types of decomposition links (i.e. AND and OR links), $FullyConsidered(q, n) = TRUE$ if and only if one of the following is TRUE:

- $T_0(q, n) \coloneqq n \notin C_q \wedge n \notin P_q$
- $T_{AND}(q, n) \coloneqq \exists l(n_1, n) \in L: (l.type = AND) \wedge (n_1 \in C_q)$
- $T_{OR}(q, n) \coloneqq (\exists l(n_1, n) \in L: (l.type = OR)) \wedge (\forall l(n_1, n) \in L: (l.type = OR) \Longrightarrow (n_1 \in C_q))$
- $T_{CONTRIB}(q, n) \coloneqq \exists l(n, q) \in L: (l.type = CONTRIB)$

where $T_0(q, n) = TRUE$ if and only if n is not and has not to be constrained by q; $T_{AND}(q, n) = TRUE$ if and only if there exists one AND link from a node n_1 to n and n_1 is constrained by q; $T_{OR}(q, n) = TRUE$ if and only if n is OR-decomposed and all the child nodes of the decomposition are constrained by q; and $T_{CONTRIB}(q, n) = TRUE$ if and

only if there is a CONTRIB link from n to q. We can now create the

$$FullyConsidered(q,n) = T_0(q,n) \vee T_{AND}(q,n) \vee T_{OR}(q,n) \vee T_{CONTRIB}(q,n)$$

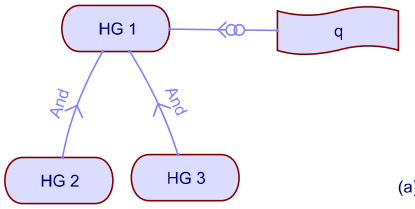
final formula of $FullyConsidered(q,n)$:

We are now able to automatically create all the sets: A_q, P_q and S_q, introduced in the previous sub-section, to keep track of all the quality requirements. The creation procedure was described in the previous section.

Figure 7-1 shows examples of AND and OR decomposition in which qualifier status sets are calculated based on the diagram on the left.

- In figure (a), there is only one qualification link from q to HG 1. This represents an intermediate state of an analysis in which q is not fully considered since HG 1 is still active and both HG 2 and HP 3 are pre-constrained by q.
- Figure (b) is an improved version of Figure (a), in which HG 3 is also constrained by q. By definition, HG 1 is fully considered but HG 3 is not and HG 2 can be (but not necessarily) constrained by q (according to the semantics). If an additional CONTRIB link is added between HG 2 and q then q will be fully considered.
- Figure (c) presents an incomplete analysis in which q is not fully considered. Only HG 1 is in C_q and both HG 2 and HG 3 and in P_q.
- Figure (d) is a slightly improved version of Figure (c), in which HG 3 is further constrained by q. Unlink in Figure (b), HG 1 is not fully considered since HG 1 is OR-decomposed. To make q fully considered, one must add a QUALIFY link from q to HG 2 and CONTRIB links from HG 2 and HG 3 to q.

We have presented the principal treatment of quality requirements in the refined goal analysis with quality requirements. In the next section, we continue to specify the implementation of QTropos with the main focus on quality requirements.



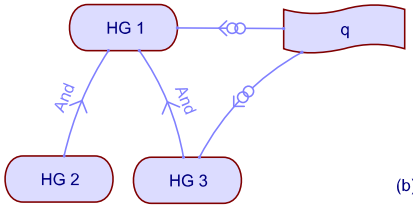
(a)

$C_q = \{HG\ 1\}$

$A_q = \{HG\ 1\}$

$P_q = \{HG\ 2, HG\ 3\}$

$S_q = \{\ \ \}$



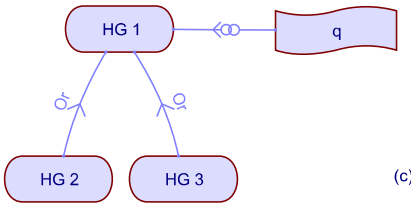
(b)

$C_q = \{HG\ 1, HG\ 3\}$

$A_q = \{HG\ 3\}$

$P_q = \{\ \ \}$

$S_q = \{HG\ 2\}$



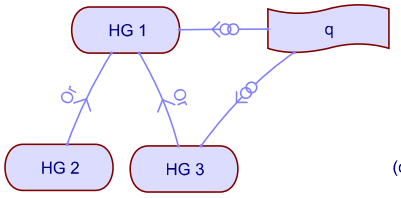
(c)

$C_q = \{HG\ 1\}$

$A_q = \{HG\ 1\}$

$P_q = \{HG\ 2, HG\ 3\}$

$S_q = \{\ \ \}$



(d)

$C_q = \{HG\ 1, HG\ 3\}$

$A_q = \{HG\ 1, HG\ 3\}$

$P_q = \{HG\ 2\}$

$S_q = \{\ \ \}$

Figure 7-1: Quality Requirement with AND/OR links

7.3. Implementation of QTropos

In this subsection, we extend the refined goal analysis presented in the previous subsection to the case of QTropos. All the notions for create Tropos diagrams are included in QTropos, which means that one can create Tropos model using what is implemented in this section.

The following two sets describe all possible types of *node* and *link* in QTropos methodology, described in Chapter 6.

$$T_N := \{ ACTOR, AGENT, QLY, SG, HG, TASK, RES \}$$

$$T_L := \{ DEPEND, ME, DECOMP, CONTRIB, AND, OR, ELICIT, QUALIFY, IMPL \}$$

$$C_q := \{n \in N | \exists l \in L: (l.type = QUALIFY) \wedge (l.to = n)\}$$

$$\forall l \in L: (l.type = CONTRIB) \rightarrow (l.label \in \{Make, Help, Unknown, Hurst, Break\})$$

Where QLY stands for Quality, SG for Soft-goal, HG for Hard-goal, RES for Resource, DEPEND for Dependency, ME for Means-Ends, DECOMP for Decomposition, CONTRIB for Contribution, ELICIT for Elicitation and IMPL for IMPLEMENTATION. Again, the set C_q and possible labels for CONTRIB links are the same as those defined in the previous section.

In order to simplify the formulae, we define the following additional sets:

$$GOAL = \{SG, HG\}$$

$$INTENT = \{SG, HG, TASK, RES\}$$

$$PROG = \{HG, TASK, RES\}$$

The $GOAL$ set contains only Goals including Hard-Goal and Soft-Goal. The $INTENT$ set contains all intentional element defined in i* framework. The $PROG$ set contains elements that can be *programmatically* implemented. However, to determine an element can be really implemented by computer code or not depends on its complexity, its semantics and the available programming resources. Usually, the analyses of the first four phases of Tropos and QTropos terminate only when all the leaf nodes are really implemented.

In QTropos, link duplicate is *not allowed*. It means that a *model* should not contain two *different* links l_1 and l_2 such that ($l_1.type$ =

$l_2.type) \wedge (l_1.from = l_2.from) \wedge (l_1.to = l_2.to)$. From now on, we suppose that there is not duplicated link in the treated $model$.

In order to define easily the $linkable(.)$ predicate for QTropos, we split the predicate in accordance to the type of link. The AND, OR, ELICIT, CONTRIB and QUALIFY links are the same as those in the General Goal Analysis.

$$linkable(ME, from, to) \coloneqq (from.type = TASK) \wedge (to.type = HG)$$

$$linkable(DECOMP, from, to) \coloneqq (from.type \in INTENT) \wedge (to.type = TASK)$$

$$linkable(AND, from, to) \coloneqq [(from.type = QLY) \wedge (to.type = QLY)] \vee [(from.type \in \{SG, HG\}) \wedge (to.type \in \{SG, HG\})]$$

$$linkable(OR, from, to) \coloneqq [(from.type = QLY) \wedge (to.type = QLY)] \vee [(from.type \in \{SG, HG\}) \wedge (to.type \in \{SG, HG\})]$$

$$linkable(ELICIT, from, to) \coloneqq (from.type = SG) \wedge (to.type = QLY)$$

$$linkable(CONTRIB, from, to) \coloneqq (from.type \in INTENT) \wedge (to.type = QLY)$$

$$linkable(QUALIFY, from, to) \coloneqq (from.type = QLY) \wedge (to.type \in T_N \backslash \{QLY\})$$

We introduce a new link type called implementation link (IMPL) into Tropos and QTropos. This link is, in fact, a dependency link between sub-actor or agent inside actors. IMPL links are used exclusively in the design phases, i.e., architecture design and detailed design. An IMPL link is created by connecting an AGENT node (that stands for sub-actors or computer agents) and a PROG node.

For Dependency links, in the requirement phases, the assertion for the *linkable* predicate is a more complicated. It has to cover two different cases:

- Dependency in a strategic dependency model: links an Actor to a Dependum and vice versa.

- Dependency in a strategic rationale model: is a derived link of a Dependency link in the corresponding strategic dependency model.

In Tropos and QTropos, it is not allowed to have a node that is decomposed by using two different types of decomposition link. Decomposition links include: AND, OR, ME, DECOMP, DEPEND and IMPL links. For example, we cannot have a node that is AND-decomposed into some child nodes and depends on other actor or agent at the same time.

For the treatment of quality requirement, $QForwardable(\ )$ and $QBackwardable(\ )$ predicates have to be defined, as done in the previous section. We use the following Table 6 to define those predicates:

Link type	$QForwardable$	$QBackwardable$
AND	FALSE	TRUE
OR	FALSE	TRUE
ME	FALSE	TRUE
DECOMP	FALSE	TRUE
QUALIFY	FALSE	FALSE
ELICIT	FALSE	FALSE
CONTRIB	FALSE	FALSE
DEPEND	TRUE	FALSE
IMPL	TRUE	FALSE

Table 6: Definition of $QForwardable(\ )$ and $QBackwardable(\ )$

To be able to keep track of the quality requirements, we also define the predicate $FullyConsidered(\ )$.

- $T_0(q,n) \coloneqq n \notin C_q \wedge n \notin P_q$
- $T_{AND}(q,n) \coloneqq \exists l(n_1,n) \in L: (l.type = AND) \wedge (n_1 \in C_q)$
- $T_{OR}(q,n) \coloneqq \big(\exists l(n_1,n) \in L: (l.type = OR)\big)$
$$\wedge \Big(\forall l(n_1,n) \in L: (l.type = OR) \Longrightarrow (n_1 \in C_q)\Big)$$
- $T_{DECOMP}(q,n) \coloneqq \exists l(n_1,n) \in L: (l.type = DECOMP) \wedge (n_1 \in C_q)$

- $T_{ME}(q,n) \coloneqq \left(\exists l(n_1,n) \in L\colon (l.type = ME)\right)$
  $\wedge \left(\forall l(n_1,n) \in L\colon (l.type = ME) \Longrightarrow \left(n_1 \in C_q\right)\right)$
- $T_{DEPEND}(q,n) \coloneqq \exists l(n,n_1) \in L\colon (l.type = DEPEND) \wedge \left(n_1 \in C_q\right)$
- $T_{IMPL}(q,n) \coloneqq \exists l(n,n_1) \in L\colon (l.type = IMPL) \wedge \left(n_1 \in C_q\right)$
- $T_{CONTRIB}(q,n) \coloneqq \exists l(n,q) \in L\colon (l.type = CONTRIB)$

Combining all the above conditions, we can create a complete formula for $FullyConsidered(\ \ )$:

$$FullyConsidered(q,n) = T_0(q,n) \vee T_{AND}(q,n) \vee T_{OR}(q,n) \vee T_{DECOMP}(q,n) \vee T_{ME}(q,n) \vee T_{DEPEND}(q,n) \vee T_{IMPL}(q,n) \vee T_{CONTRIB}(q,n)$$

With this predicate, it is easy to create the sets A_q, P_q and S_q, introduced in the previous sub-section, to keep track of all the quality requirements.

Figure 7-2 is an example of an incomplete analysis. According to the above definitions, the sets C_q, A_q, P_q and S_q have the following values:

$C_{Promptness} = \{Book\ Found, Catalogue\ Searched, Online\ Search\ Done\}$

$A_{Promptness} = \{Online\ Search\ Done\}$

$P_{Promptness} = \{\ \ \}$

$S_{Promptness} = \{Result\ Ordered\}$

$C_{Low\ Cost} = \{Book\ Bought, Book\ Ordered, Sell\ Used\ Book\}$

$A_{Low\ Cost} = \{Book\ Offered\}$

$P_{Low\ Cost} = \{Sell\ New\ Book\}$

$S_{Low\ Cost} = \{\ \ \}$

As we have already noticed, the direction of quality transfer and the direction of node decomposition are not always correlated. It is the forward direction in dependency links and the backward direction in other links.

From Figure 7-2, we can conclude that the Promptness quality requirement is fully considered at the Book Found and Catalogue Searched nodes but it is not fully considered at the Online Search

Done node. The Low Cost quality requirement is fully considered at Book Bought and Sell Used Book but it is not fully considered at the Book Offered node. The full consideration of Low Cost requirement at Sell Used Book is not sufficient to conclude the full consideration of Low Cost at its parent node, i.e. Book Offered hard-goal. At this stage, neither quality requirement is fully considered overall. One can examine the four qualifier status sets to decide, for a quality requirement, which nodes must be further analyzed so that the quality requirements can be fully considered.

In general, a quality requirement must be fully considered before its satisfaction can be taken into account.

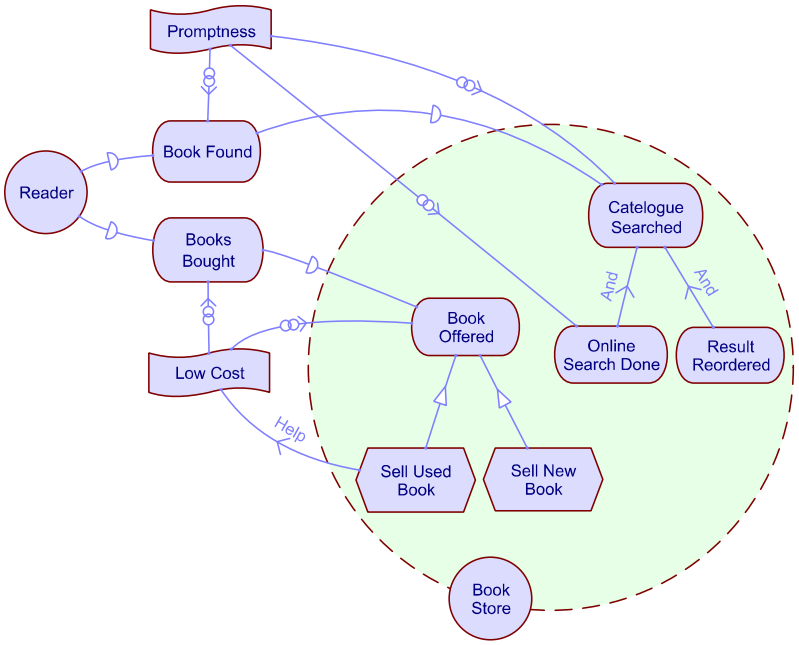


Figure 7-2: Example of Quality Requirement in action

7.4. Implementation of QCase

Based on the above specification, we have built a preliminary version of QCase in which our refined goal analysis and QTropos are included. In fact, all the nodes and links used in analysis diagrams of Tropos and the refined goal analysis can be created using those defined for QTropos. As a consequence, we only need to implement QTropos in QCASE.

Our latest version of QCase was implemented by more than 13000 lines of Java. The choice of Java is to guarantee the greatest

portability among the popular platform. The detailed design of this tool is carried out as generically as possible in order to offer a high extensibility to other type of analysis from other methodologies. The objective is to validate the described treatment of quality requirements to maximum possible context.

All the models analyzed by QCase are saved in a XML format. Each QTropos document contains a model together with all the graphical properties that are used to render diagrams. The structure of each document can be summarized as below:

- A Document is a container of Nodes, Links and Diagrams. A Document is saved in a XML file, currently named with then QGL extension.
- Nodes and links (visible or invisible in diagrams) describe the structure of the models. In fact, one can include as many models as he likes in a document. Two different models should not have any elements in common, i.e., nodes, links and diagrams. However, for the reason of clearness, it is recommended to store different models in separate documents.
- A Diagram is a graphical representation of a portion of a model, i.e, a subset of nodes and links. Different diagrams are linked together by sharing their modelling elements, i.e. nodes and links. An element can have different display properties and relative positions with others in different diagrams. This allows developers to have multiple views on the models.
- One can create an overview of a model by creating a diagram including all nodes and links of the model. For large project, this may not be possible. Instead, hierarchical views should be created, as done with QTropos, in which the Strategic Dependency analysis is an overview of dependencies between actors (modules) and the Strategic Rationale analysis focus on the details of one or a few actors (modules) in the relations with other actors.

QCase use the VectorGraphics package offered in the FreeHEP Java Libraries (downloadable at http://java.freehep.org/) to generate vector graphics for exchanging and printing high quality and scalable diagrams.

All of the automatic procedures described earlier in this chapter as well as some others useful checks are implemented in the latest

version of QCase. Besides the treatments of qualifiers, the main theme of this thesis, many automatic treatments are added for other type of nodes, i.e., containers and intentional nodes. In particular, applying those general treatments provide even much richer set of capabilities for Tropos that one can find in other existing analysis tools for Tropos and i* such as: DesCARTES (http://www.isys.ucl.ac.be/descartes/), OpenOME (https://se.cs.toronto.edu/trac/ome/), etc.

We will briefly describe the user interface and the most important features of QCase.

7.4.1. Main interface

Figure 7-3 shows the graphical interface of the latest version of QCase. The main menu and the main toolbox are shown in the upper part of the window above the main working space where users can edit diagrams. Each open diagram is shown in a tabbed window that allows users to easily navigate between diagrams.

Three dialogs can be permanently shown on the QCase interface:

- The Main Toolbox dialog allows users to select an editing tool in order to create dialogs. Except for the Select button that serves for selecting an existing element in diagram, other buttons are for adding new elements into the current diagram, see Figure 7-4.
- The Diagrams dialog contains the list of all available diagrams in the opened document. Using this dialog, users can add new diagrams, remove old diagrams from the document, open a diagram in a new tab or change the properties of a diagram, see Figure 7-5.
- The Log Box dialog shows the trace of user manipulations. The manipulations are displayed in chronological order. Each entry is prefixed by a timestamp at which the manipulation is done. Actual log information includes the creation, deletion and modification of all the elements: nodes, links and diagrams, see Figure 7-6.

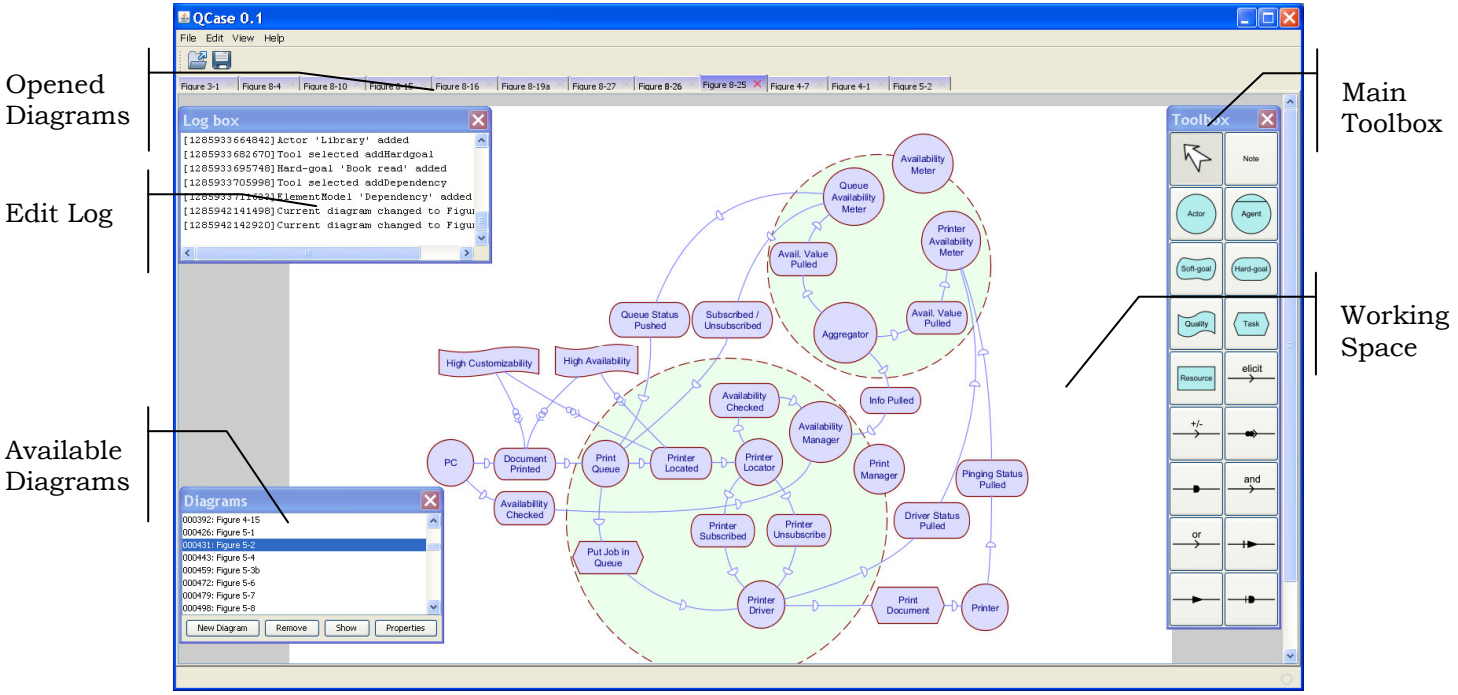


Figure 7-3: Main windows of QCase

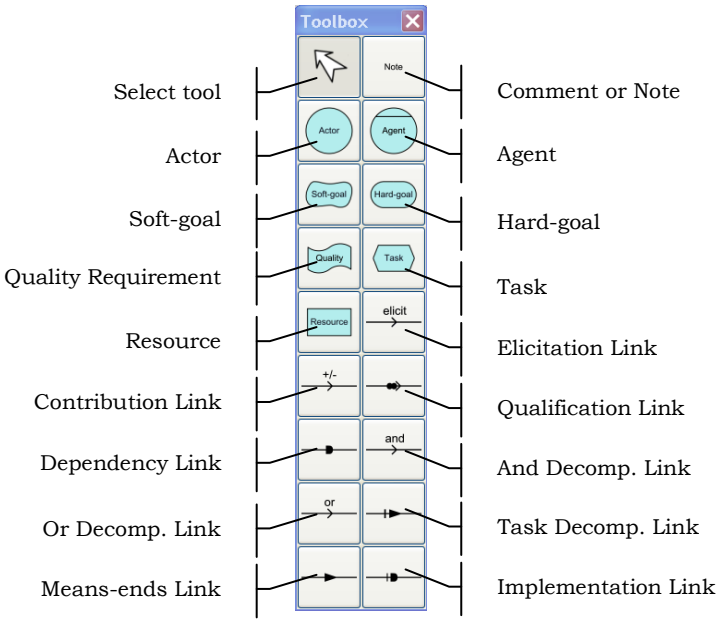


Figure 7-4: Main Toolbox of QCase

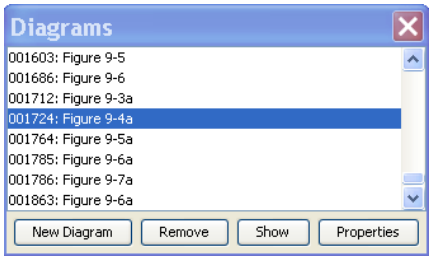


Figure 7-5: Diagrams dialog

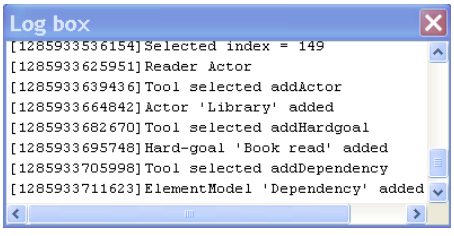


Figure 7-6: Log box

7.4.2. Element manipulations

i) Diagrams

When a user wants to add a new diagram, he can set several properties it. Those properties can be set at creation time or can be modified later through a Diagram dialog. Modifiable properties are: "Name" of diagram, a descriptive "Note" and the "PaperSize" of the diagram, see Figure 7-7.

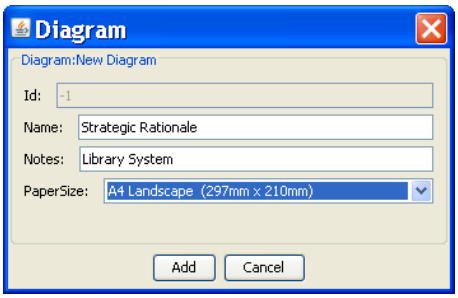


Figure 7-7: Diagram dialog

ii) Nodes

The "Node" dialog is for creating new nodes or modifying properties (or attributes) of existing nodes. Like links, for unbound nodes, users can change the "Name" of the node and its descriptive "Notes". The type of the edited node is shown by the drawing in "Element information". For bound nodes, we cannot change their properties but we can see a list of diagrams where the edited link is used. Users can decide to bind or unbind a link by using the left panel, see Figure 7-8.

iii) Links

The "Link" dialog is for creating new links or to modify properties (or attributes) of existing links. For unbound links, users can change the "Name" of the link and its descriptive "Notes". The type of the edited link is shown by the drawing in "Element information", see Figure 7-9.

For bound links, we cannot change their properties but we can see a list of diagrams in which the edited link is used. Users can decide to bind or unbind a link by using the left panel.

Note that there are more possibilities to bind a node than a link, since to bind a link, the bound link and the target link must share both "From" node and "To" node.

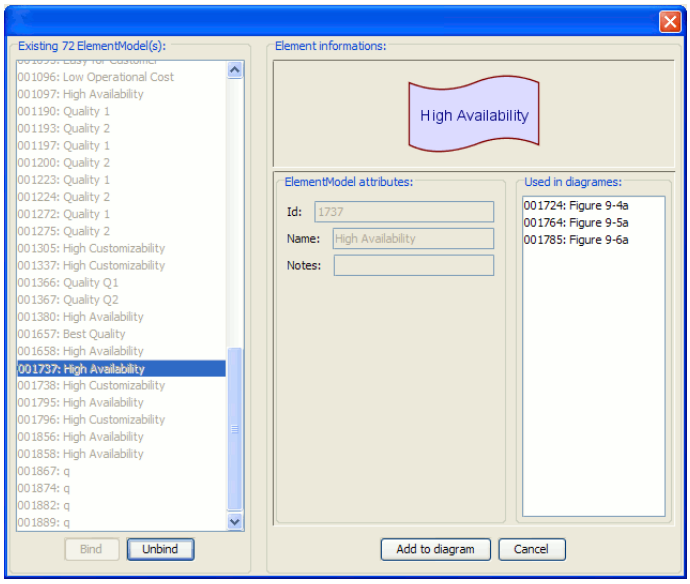


Figure 7-8: Node Dialog

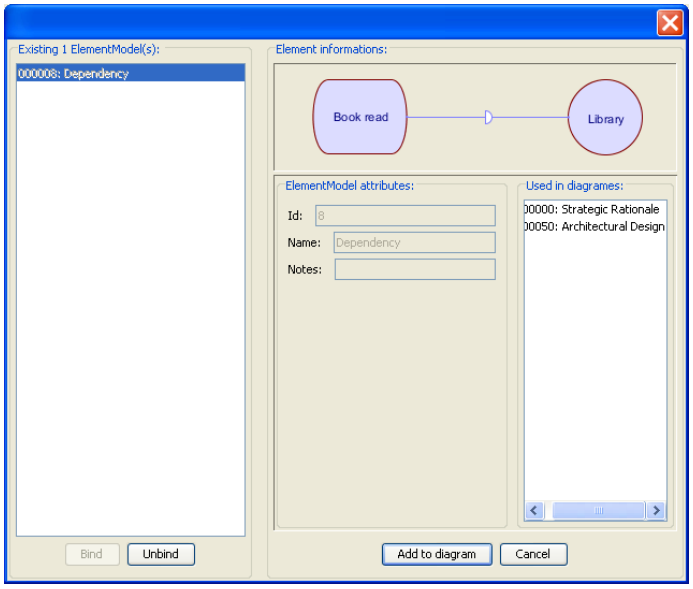


Figure 7-9: Link Dialog

7.4.3. Qualifier consideration

Among different types of nodes, qualifiers are given more attention. As discussed in Section 7.1, a qualifier can be considered *fully considered* if certain conditions are met in accordance to each methodology. This notion is also implemented for QTropos in QCase. The consideration state of a qualifier can be displayed in a permanent dialog called Qualifier Explorer, as shown in Figure 7-10.

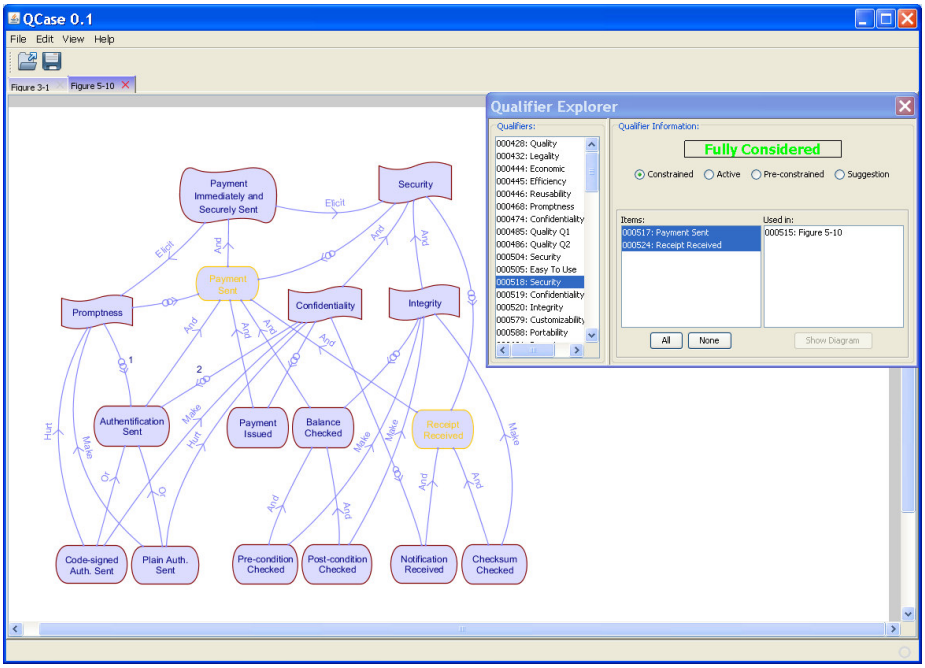


Figure 7-10: Qualifier Explorer (Fully Considered)

If one selects a qualifier in this dialog, its consideration state will be displayed thanks to the four sets C_q, A_q, P_q and S_q described above, which are respectively labelled as Constrained, Active, Pre-constrained and Suggestion in QCase. When one of those sets is chosen, selected nodes can be highlighted in the diagram editing region. If one node of one of those lists is chosen, a list of diagrams in which the node appears is given for possible diagram switch.

Figure 7-10 shows a situation where the selected qualifier, Security, is automatically determined as Fully Considered. The consideration state becomes green when the selected qualifier is fully considered. Otherwise, it becomes red when the selected qualifier is only partly considered, as shown in Figure 7-11.

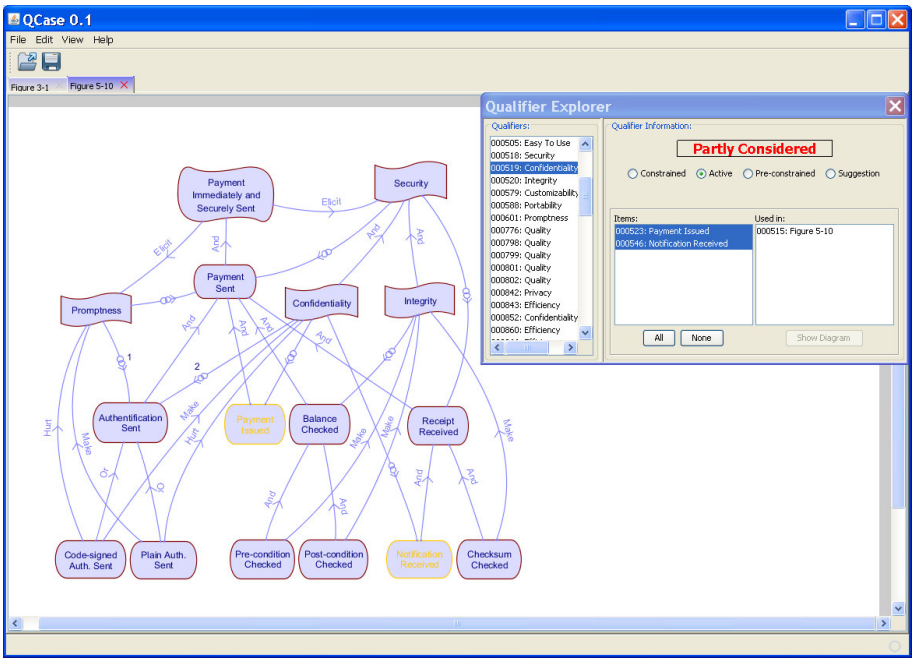


Figure 7-11: Qualifier Explorer (Partly Considered)

In this figure, the qualifier Confidentiality is only partly considered. The Active set of nodes at which this qualifier has not been fully considered yet is showed. This gives hints to analysts to take necessary action to fully consider this qualifier.

7.4.4. Extra automatic procedures

We show here a nonexhaustive list of automatic procedures offered by the latest version of QCase.

- Automatic node assignment: when a node is decomposed, the child nodes are automatically assigned to the container of the parent, depending on the decomposition type. This general procedure is further extended in QTropos to automatically assign a node to an actor when a dependency link from the strategic dependency analysis is refined in the related strategic rationale analysis.
- Automatic boundary draw: this complements the above node assignment procedure in rendering diagram. Given that the contained nodes are correctly determined, QCase will automatically determine the boundary of the container by

computing the smallest circle that contains all the graphical representations of all contained nodes.

- Node binding suggestion: when a node is manually added by a user, a list of already added nodes is proposed to users to bind nodes in different diagrams. Figure 7-8 shows an example in which a quality node is added and bound to High Availability quality that is already used in three other diagrams.
- Link binding suggestion: same as node binding. However, since it is usually not allowed to have duplicate links between two nodes, link binding is mandatory in some situations. Figure 7-9 shows an example in which a dependency link is added between the User actor and the "Best Print Service Offered" soft-goal. QCase notices that such a link already exists and is used in two other diagrams, and it proposes to the user to do the binding.

When using QCase in practical applications, those automatic procedures free us greatly from these usually tedious tasks and let us concentrate on more essential issues. Among them, three most effective and accelerating tools in the latest version are: consideration of state determination, node assignment, and boundary drawing.

This chapter has shown how our proposed approach can be applied to existing methodologies. The usability of the chosen methodologies, i.e., Tropos and KAOS, has been confirmed both in real-life applications and in a large supporting literature. QCase not only shows how to implement the material proposed in the previous chapters, but it also defines effective tools, i.e., consideration states, from this material to facilitate the analysis of quality requirements. As far as we know, the features like automatic node assignment and boundary drawing have not been introduced into existing Tropos tools. The proposed treatments of quality requirements and their consideration states could give a better vision on the analysis of quality aspects of quality requirements, especially in the context of goal-based requirements.

Chapter 8 PRINT SERVICE CASE STUDY

This chapter presents an application of QTropos, described in Chapter 6, to a case study of a printing service called "Print Shop" that needs improvements concerning the availability and the customizability of its services.

The main objective of this case study is to illustrate how our approach can be applied in a real application. We aim to redesign the existing system in order to deal with the additional quality requirements. We want to reflect the additional quality requirements into the Print Shop structure through our analysis and to have some visible improvement with respect to the availability and customizability of the system. A really optimal solution for this Print Shop case study is not what is sought here.

The case study is structured into three main sections. Section 8.1 describes the context of the Print Shop operation. Section 8.2 analyzes the information system currently deployed at the Print Shop. Section 8.3 gives a detailed analysis of how the system will fulfil the newly added requirements about availability and customizability. Then in Section 8.4, a simulation of the new Print Shop is implemented using Jadex (POKAHR, A. et al., 2005). Some remarks about the applicability of our approach will be drawn from this case study at the end of this chapter.

8.1. Application Scenario

Suppose that a company has set up a printing service, called Print Shop, where customers can rent a computer, create or import and then print documents. Their service is targeted at students on a large university campus. The company rents a large office in which they install some personal computers on which users can import (from

some storage support) and edit documents. Printers offer different capabilities like colour or black & white printing, etc. After one year of operation, they notice, on the one hand, an unbalanced usage of printers. Some printers are almost not used, while others are overloaded and create long queues of jobs. On the other hand, infrequent users struggle selecting a printer with their desired printing quality. In order to shorten the waiting time for users and to more efficiently use the printing infrastructure, the owners decide to invest in a better print management solution.

8.2. Current Situation and New Requirements

In order to improve the current service of the shop, the owners provide us with an analysis in Tropos with two principal actors: “User” and “Print Shop”. Users of the “Print Shop” can rent a computer to:

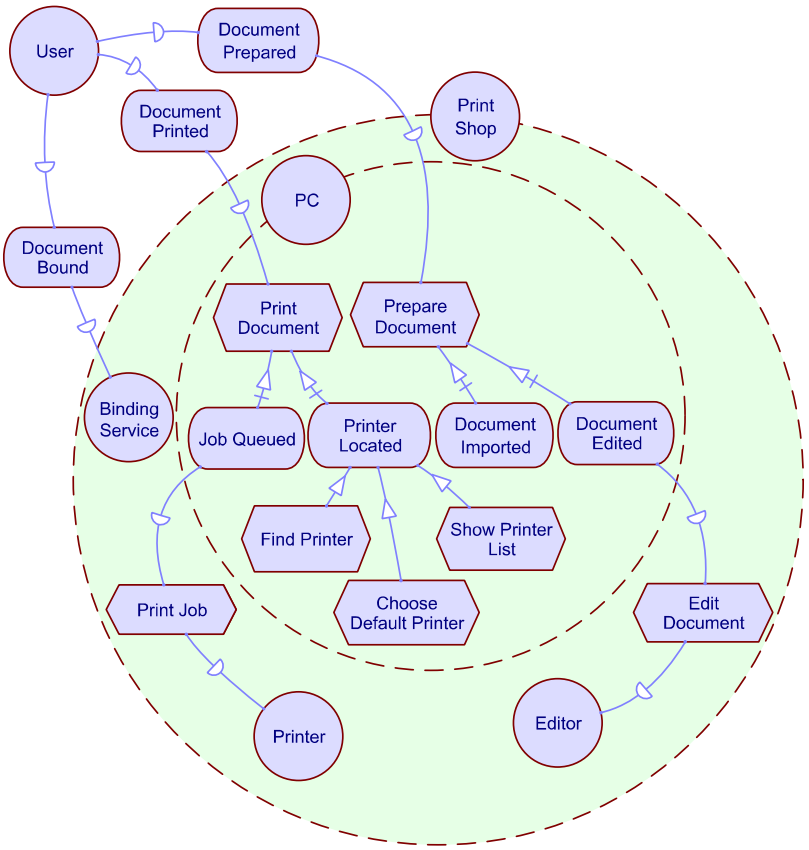


Figure 8-1: Current State of Print Shop

- import ready-to-print documents from their storage supports like: USB stick, CD, online directories, etc. into the rented computers,
- edit documents with both basic and professional text editors with a large support for different formats,
- print documents with desired qualities and formats that a personal printer cannot offer,
- use other facilities like: scanners, photocopiers, book binders, etc. in order to produce desired documents.

Figure 8-1 shows three dependencies representing services offered by the Print Shop: “Document Prepared” (document imported and edited), “Document Printed”, and “Document Bound”. Those services are further analyzed to identify four sub-actors inside the “Print” Shop: “PC” (personal computer), “Document Editors”, “Binding Service”, and “Printers”.

Every PC has been assigned a printer as its default printer on which users can print their documents. When failures are found in a printer or when users need to print a document with quality that is not supported by the default printer, users can change the default printer for their computer.

As mentioned above, this mode of operation drastically affects the capacity of the Print Shop and it has to be changed.

Why a Printer Manager?

In order to improve the service, we collect feedback from both shop users and shop operators. We have identified two major sources of the poor performance of the shop.

First, from the shop point of view, its poor performance is due to the fact that many printers are left unused while others are overloaded. From the user point of view, it is uncomfortable to have to wait too long for a document to be printed. It is also frustrating for a user to discover that the default printer runs out of order while having print jobs waiting in its queue. In this case, the user has to select another printer and move the documents into the new queue. Novice users may have to ask the shop operator for help, which results in a waste of time for both users and shop operators.

Second, since users may not know the capabilities of each printer, they cannot choose a printer for each job. Furthermore, since the price of each printed page depends on the quality of printing, users want to print their documents only at the desired quality to reduce printing costs. Indeed, for a draft, it is preferable to print it with a

lower quality and with multiple pages on a paper side. Unfortunately, these options are not supported by all of the available printers. Shop operators have to help users to choose the suitable printer

With those two problems in mind, we replace the hard-goal "Document Printed" in the original design (see Figure 8-1) by the soft-goal "Superior Print Service Offered".

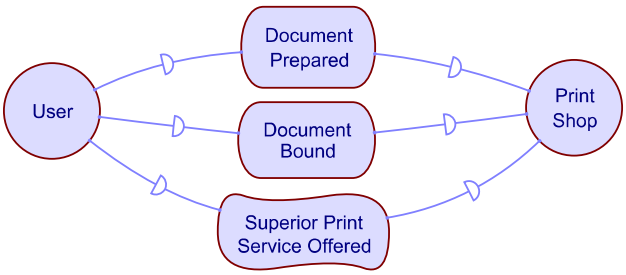


Figure 8-2: Early Requirement Analysis for Print Shop – Strategic Dependency Model

We use QTropos to analyze the requirements. Figure 8-2 shows a strategic dependency analysis of the early requirement phase, which describes the main tasks that must be carried out.

If we do not replace the hard-goal "Document Printed" and we consider only three hard-goal dependencies, the current system (as in Figure 8-1) is already a good solution. With the new soft-goal, we carry out, in Figure 8-3, the strategic rationale analysis based on the strategic dependency model in Figure 8-2.

The soft-goal "Superior Print Service Offered" elicits two additional quality requirements: "High Availability" (i.e., short delays) and "High Customizability" (i.e., in terms of printing quality – colour, resolution, etc.). In fact, with the above soft-goal, one could elicit other quality requirements such as: "Confidentiality" (i.e., protection of printed contents), "legality" (i.e., copyright protected), etc.

In Figure 8-3, the task "Print Document" is decomposed into two hard-goals: "Printer Located" and "Job Queued". For a requested print job, the normal procedure is the following: first, a printer must be selected to receive the print job ("Printer Located") and then the text editor (or local print queue of each computer) puts the print job at the end of the print queue of the selected printer ("Job Queued").

We constrain the hard-goal “Printer Located” with the quality requirements “High Customizability” and “High Availability” to emphasize that the new solution must allow the system to choose the right printer for each print job. Remember that the right printer is the one that can print the document in the requested quality and with the shortest delay.

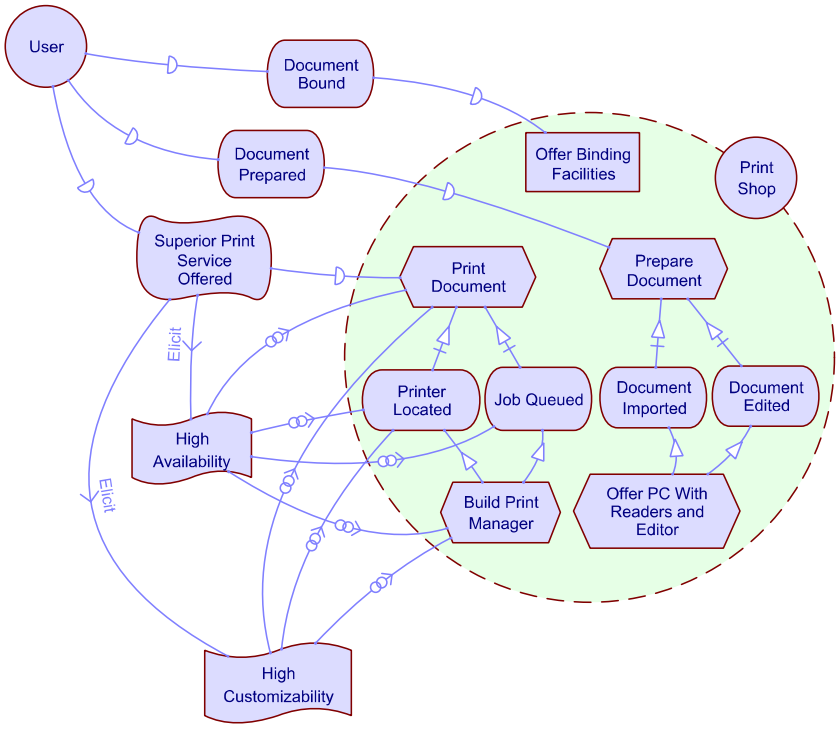


Figure 8-3: Early Requirement Analysis for Print Shop – Strategic Rationale model

The hard-goal “Job Queued” is also constrained to have “High Availability”. “Job Queued” is the starting point for the printing process when the right printer has been selected. However, many other factors can delay the pended jobs. The estimated delay (calculated from the printer queue contents) and the true delay (calculated when the job is actually printed) can differ significantly. The consequences can vary between some extra delays and a possible change of selected printer in case of major problems like paper jams, printer errors, etc. In those cases, when satisfying the hard-goal “Job Queued”, the estimation of delay must be controlled and compared against the initially estimated value. Extra treatments must be introduced to cut back the delay when possible.

At this stage, in the current system, technical solutions to fulfil the additional quality requirements do not exist. Therefore “Build Printer Manager” (constrained by “High Availability” and “High Customizability”) can be considered as a good solution for the hard-goals “Printer Located” and “Job Queued” on which “High Availability” and “High Customizability” requirements are added. It is expected that with a well-built “Printer Manager”, the current problems of printer availability and quality will be resolved.

In the next subsection, we carry out the analysis, in the late requirement phase, with the presence of the “Printer Manager” actor. The objective is to specify the functions of the “Printer Manager” and then to design it to fulfil the desired goals.

8.3. Towards an Improved Print Shop

We introduce the “Print Manager” actor between computers and printers. Instead of sending print jobs directly to the predefined printer, PCs send their print jobs to the “Print Manager”. When the “Print Manager” receives a print job, it will select a printer that can print the job in the requested quality and in the shortest time. The “Print Manager” will be responsible for sending print jobs to printers as well as for keeping track of the state of printers.

In Figure 8-4, in order to simplify the analysis, we include the “Print Shop” actor as the new version of the “Print Shop” actor in Figure 8-1. Compared with Figure 8-1, the “Printer” agent is taken out of the boundary of the Print Shop actor in order to better situate the new “Print Manager” actor. To see what should be changed in the current implementation, we first analyze the new “Print Shop” actor in Figure 8-5.

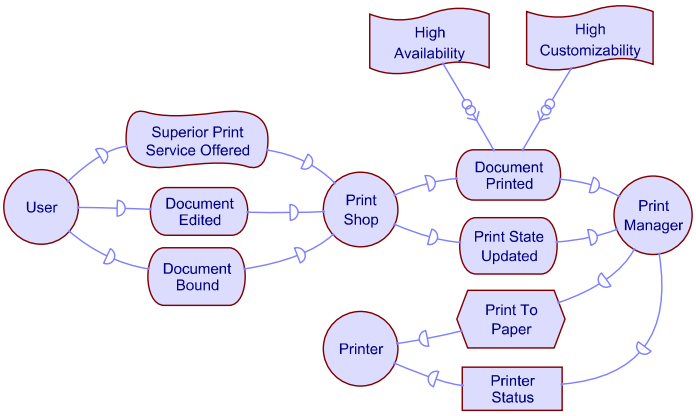


Figure 8-4: Printer Manager for the Print Shop

The most important change in the “Print Shop” actor is that the task “Print Document”, in Figure 8-3, has been replaced by the task “Handle Print Job” in Figure 8-5, that is constrained by two quality requirements: “High Availability” and “High Customizability”. The PC no longer sends directly print job to printers. Instead, it transfers the hard-goal “Document Printed” to the actor “Print Manager” and expects that the “Print Manager” to report back the “Updated Print State”.

We now shift the attention to the main actor “Print Manager” whose main functions are to satisfy two hard-goals: “Document Printed” and “Print Status Updated”.

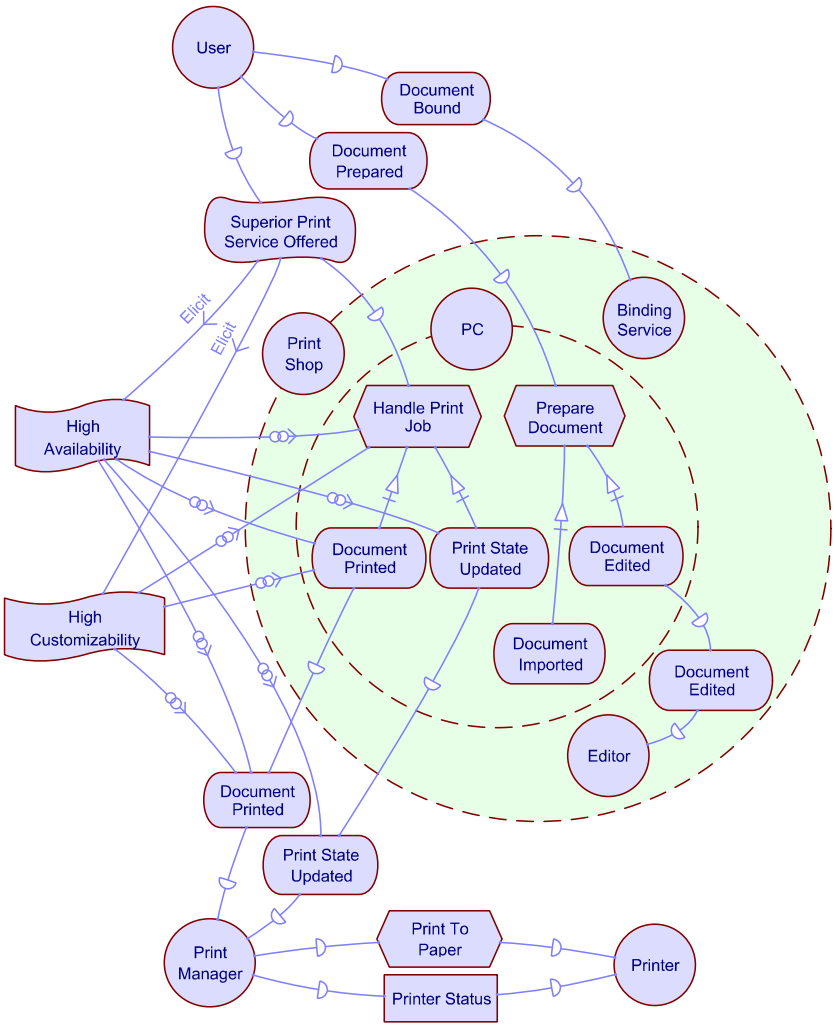


Figure 8-5: The new Print Shop actor

The preliminary analysis of the actor “Print Manager” was carried out in Section 6.5.2 as an example of how to use social patterns. We copy Figure 6-21 as Figure 8-6 for the sake of completeness.

In the following analysis, we present how to fulfil the quality requirements using the patterns introduced in Chapter 5. Our objective is to create a sub-system that keeps track of the state of all available printers in order to distribute print jobs. The sub-system will keep track not only the working printers but also the ones that are in sleeping (power-saving) mode. When needed, sleeping printers can be woken up to receive print jobs.

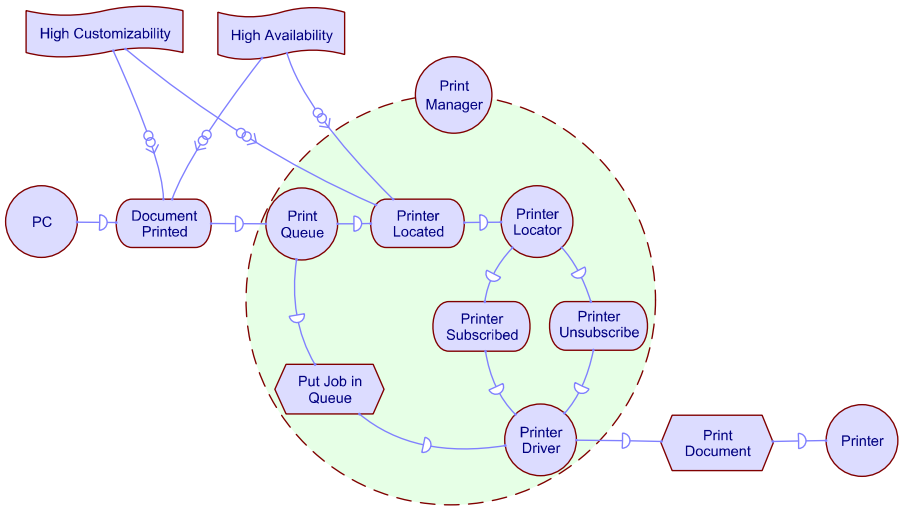


Figure 8-6: High quality print service

It is simple to fulfil the “High Customizability” requirement. In fact, the printer drivers often provide in detail all the printing qualities of the associated printers. The “Printer Locator” agent needs only to filter all printers that cannot offer the desired printing mode out of the candidate list. To cope with such binary decisions, we only have to monitor the printer capabilities to automatically assign printer. More difficult task for the “Printer Locator” is to select, among those candidates, a printer that can carry out the print job in shortest time.

We identify the following signal sources that can provide indicators of printer availability and capabilities:

- printer state: standby, stopped, started, printing, connected or disconnected; For printing status, information about the current state of printer internal queue is also included in the returned signal; 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 more detail about the printer (printer parameters and capabilities), if the printer is online,
- print queue state: the number of documents in the queue that will delay make the printing of the newly introduced document.

In our application, we have to consider also the physical positions of PCs and printers. Indeed, the walking time to retrieve the printed pages become considerable if the chosen printer is far away from the PC. The walking time will be added in the delay estimation in accordance to the relative position between PCs and printers.

In Figure 8-6, there are two quality requirements, “High Customizability” and “High Availability”, constraining the “Document Printed” hard-goal. In the remaining part of this section, we detail only the “High Availability” requirement, which is the more complex. The other requirement can be easily included when the decision of printer selection is made, since this only influence the final decision in a binary way, i.e., a printer is capable to print with desired printing parameters or not.

For the availability signal, since printers may be disconnected accidently from the network, they cannot notify themselves their absence to the printer manager, therefore we decide to build an agent that check the availability of each printer using the Signal Pulling pattern (see Section 5.2.2).

In Figure 8-7, 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 is pinged by the agents “Printer Availability Meter” at every small interval of time (e.g. 0.5 seconds). The drivers of pingable 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”. Since this signal is changed by precise events, we use the Signal Pushing pattern to report the queue status. The “Queue Availability Meter” agent is the responsible actor for recording the states at every events reported by the “Print Queue” agent. We group the two meter agents into a sub-actor called “Availability Meter”.

Two sources of signal received by “Printer Availability Meter” and “Queue Availability Meter” are aggregated at the “Aggregator” agent. The “Aggregator” agent uses the pulling scheme to pull reports from the two meters and then combines them into reports ready to be sent

to the “Availability Manager” agent. Since the three agents inside the sub-actor “Availability Meter” are hard-bound, we only use the simplified version of Signal Pulling pattern, in which the Signal Manager is left-out.

Using this structure, the “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).

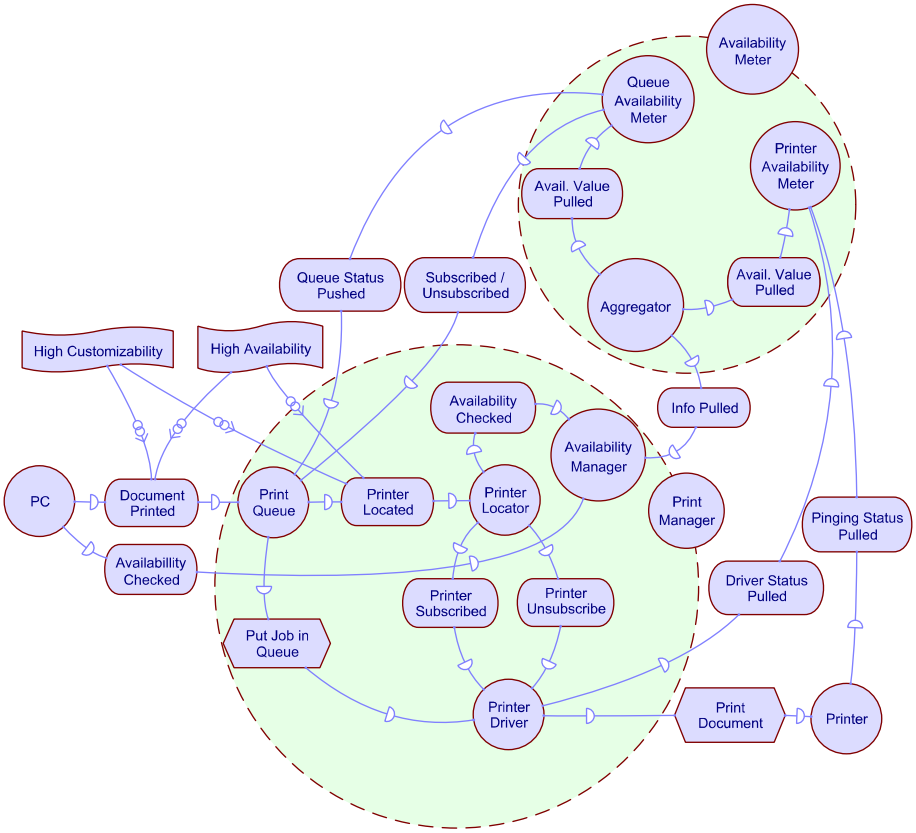


Figure 8-7: Printer availability monitoring

In Figure 8-7, all the applications of patterns are simplified. For the Signal Pulling and Signal Pushing patterns, the control of presence is omitted. In order to guarantee the presence of the newly added agents for monitoring the availability of printing service, we add also a Yellow Pages sub-actor into the system. In this sub-actor, there are three agents: “Singleton Manager”, “Printer Manager”, and “Driver Manager”. Singleton agents, agent with only one instance, are managed by the “Singleton Manager”. “Printers” and their “Driver”

are managed respectively by the “Printer Manager” agent and the “Driver Manager” agent. These managers play actually the role of “Signal Manager” in the Signal Pulling and Signal Pushing patterns.

Figure 8-8 shows the missing part in Figure 8-7 where the three agents in the Yellow Pages sub-actor complete the missing parts of applied patterns.

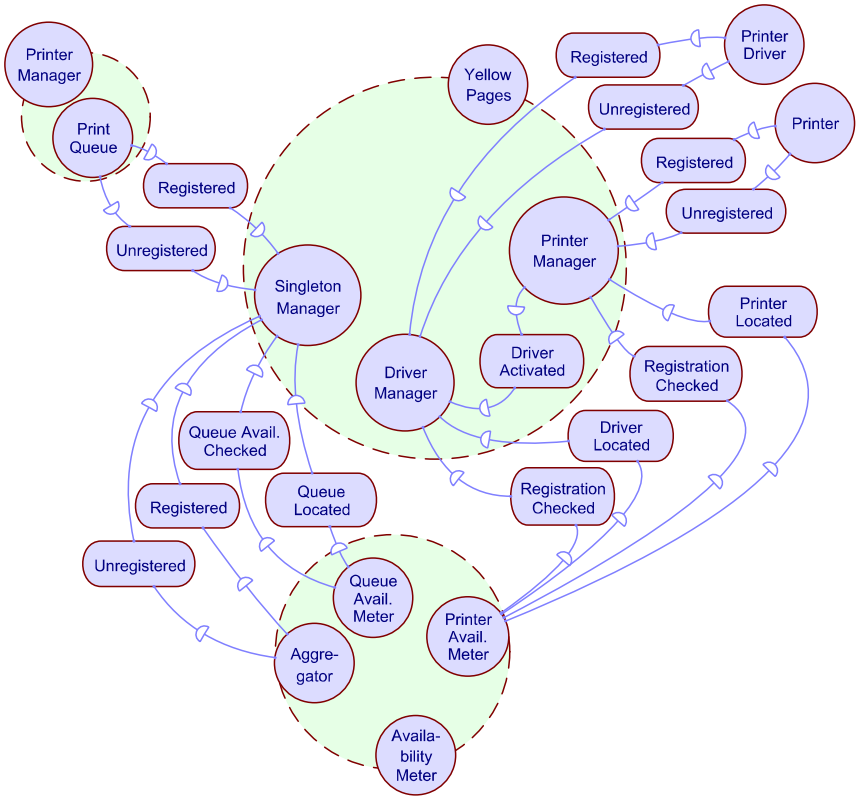


Figure 8-8: Yellow pages sub-actor

At the PC side, we have to introduce an agent named “Print Agent” that has to carry out necessary actions to satisfy two hard-goals: “Document Printed” and “Print State Update”, as showed in Figure 8-9. In this agent, the most noticeable part is its plan for the printing “Printing Plan” that is provoked when a job is pended in the “PC Print Queue”. To carry out correctly this “Printing Plan”, this agent has to depend on the global “Print Queue” agent and the “Availability Manager” agent. For the sake of brevity, more details of other parts of this agent are omitted in this document. The following paragraphs show how this plan triggers the printing process when there is a pending job in the “PC Print Queue”.

A printing job is initially launched by a PC or more precisely by the "Print Agent" via its "Printing Plan". Before printing, the "Print Agent" refers to the "Printer Availability Manager" to check whether the document can be printed in less than, 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.

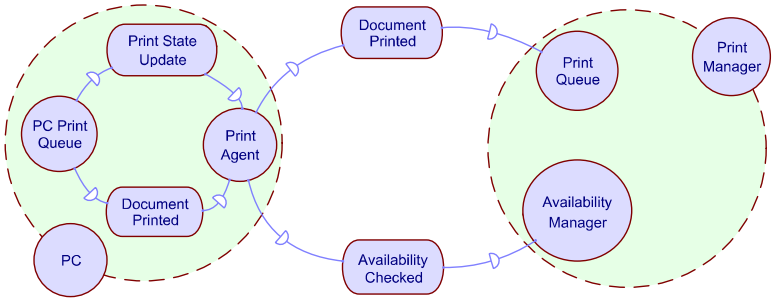


Figure 8-9: Print Agent for PC

If the printing job is successfully scheduled and sent to the print queue, it has to waits for its turn. It can happen that between the moment when the job enters the queue and the moment the job is taken out of the queue for printing, the expected printing time can be changed and no longer respects the initial schedule. The "Printer Locator" is responsible to check this and ask the "Availability Manager" to start more printers when necessary.

Figure 8-10 shows two applications of the Quality Assurance pattern between the "Print Agent" and "Printer Locator" towards the "Availability Manager". In the present example, the printing plan is also constrained by the requirement of "high resolution". At the end of the design process, the "Print Agent" and "Printer Locator" will have to be connected to at least two quality managers: "Availability Manager" and "Resolution Manager" where the latter is omitted in this development.

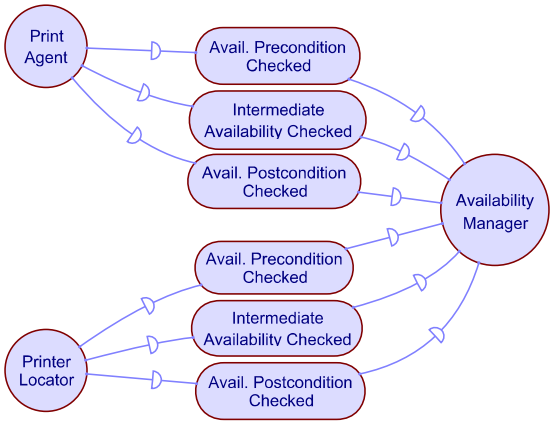


Figure 8-10: Availability Quality assurance of printing service

A portion of the class diagram focusing on the quality management subsystem is shown in Figure 8-11. The main class in Availability control is the agent "Availability Manager". It subscribes to the "Aggregator" to receive the update of the "Queue" state and "Printer" state. From the received data, it can estimate the maximum delay for a new job requested by any "Print Agent". The "Print Agent" has references to two quality managers: "Resolution Manager" (which is not shown) and "Availability Manager" to assure the quality of its printing plan. The "Print Agent" also 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". The "Printer Availability Meter" has references to all the physical "Printers" and their software drivers in order to ping and query the configuration of the "Printers".

In Figure 8-11, 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 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 "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.

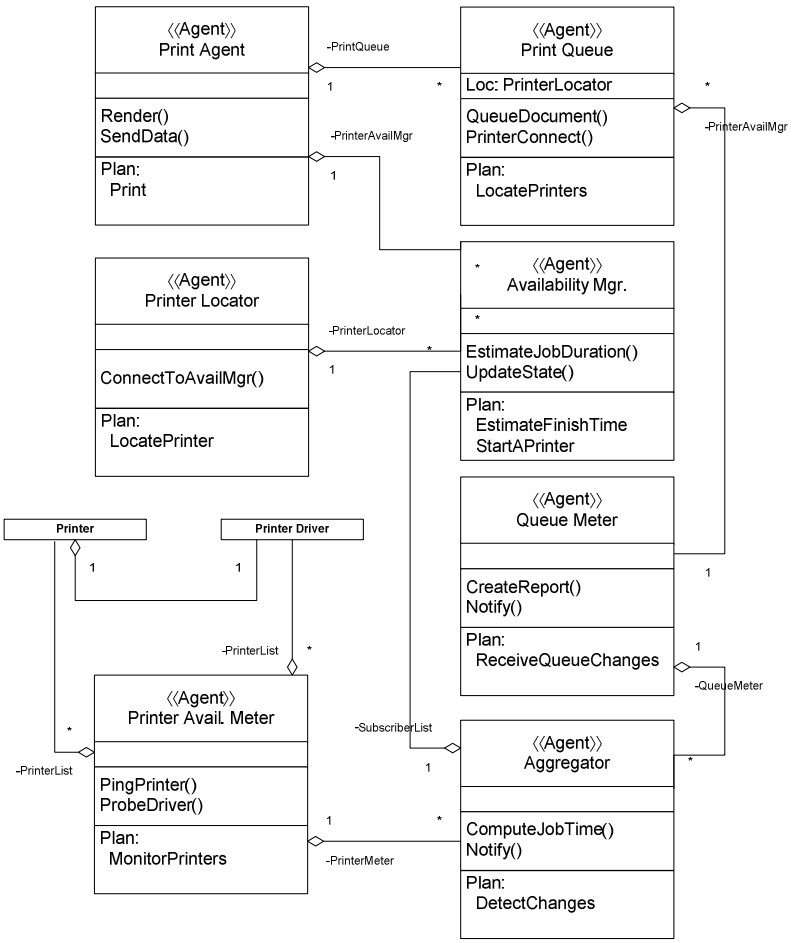


Figure 8-11: Class diagram of Quality Management Subsystem

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 new instructions. 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 after the transmission is not far from reality, since the print job is already in the “Print Queue”. However, if it happens that the expected time cannot be respected when the printing job is about to be effectively carried out, the “Printer Locator” and the “Availability Manager” will try to make more printers available to guarantee the deadlines.

Figure 8-12 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.

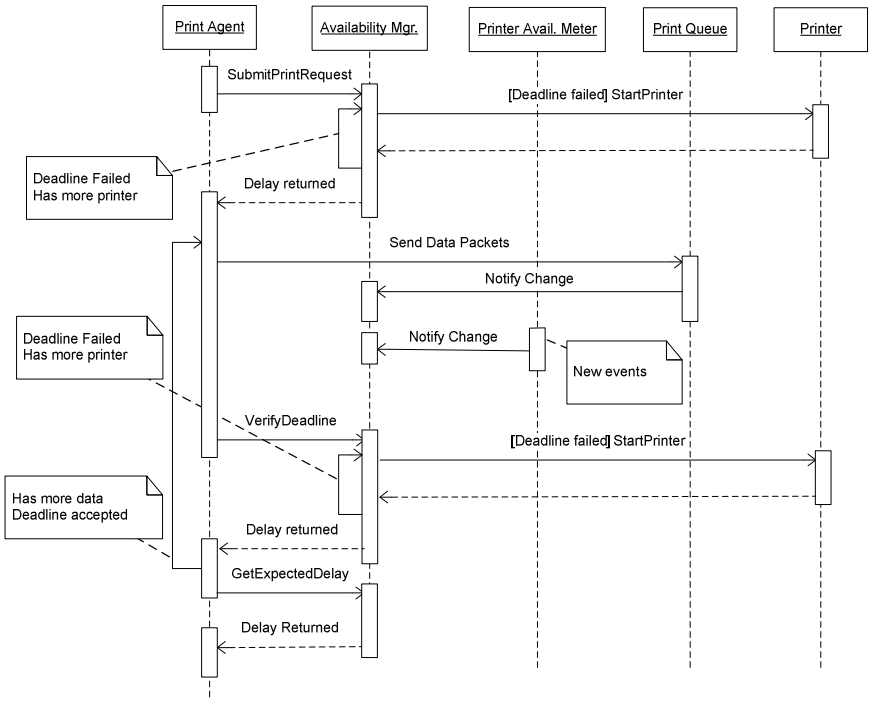


Figure 8-12: Top-level interaction of Printing Plan

8.4. Print Shop Simulation

In order to test our "Print Management System", we create a simulation of a "Print Shop"

Figure 8-13 shows a map of a print shop on which we do some simulations. The print shop is divided into several sections due to the original layout of the rented building. We can see, in the print shop, there are PCs on which users prepare documents and send the print command, and printers. For each PCs, an agent is created to simulate the activities of a user.

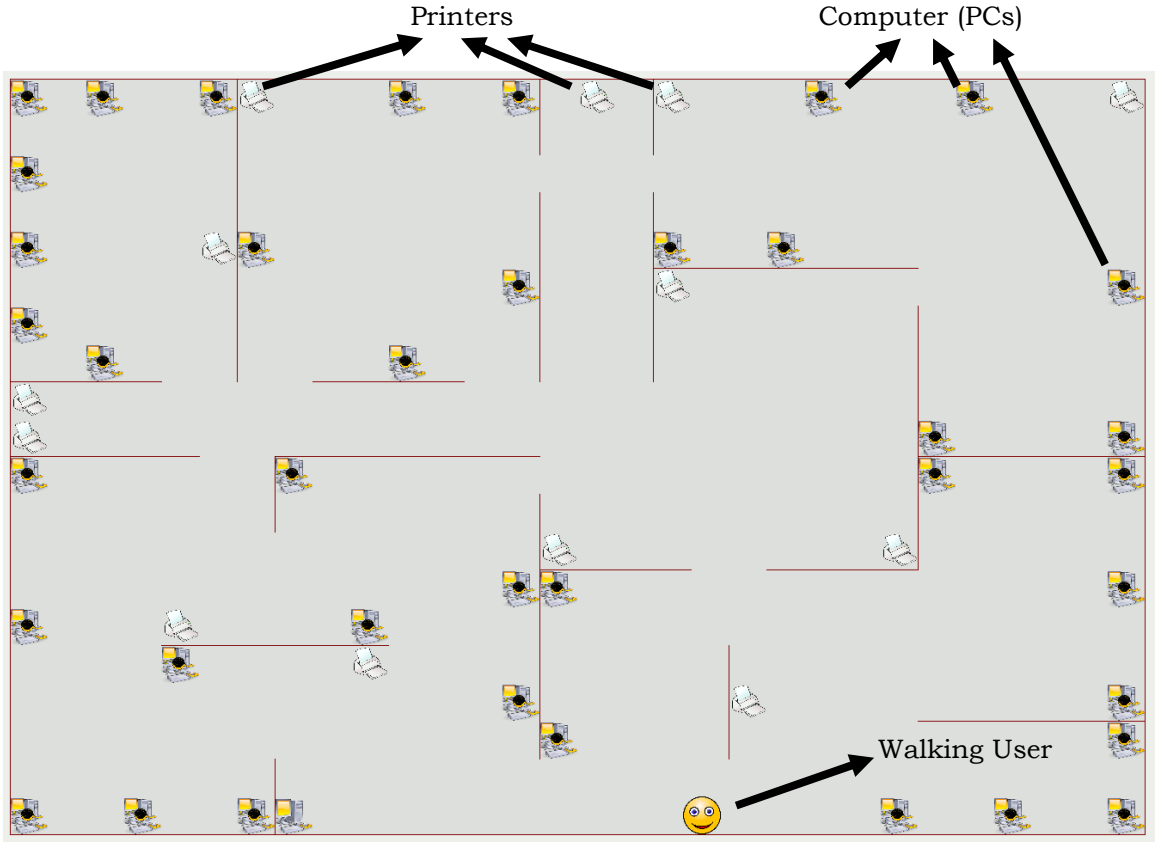


Figure 8-13: a Print Shop plan

The behaviours of a user can be summarized in the following states:

- He opens one of his documents and edits it. If he has finished typing, he sends a printing command.
- He takes a short pause.
- If he is notified that one of his documents is printed, we walks towards the printer to taken the printed pages.

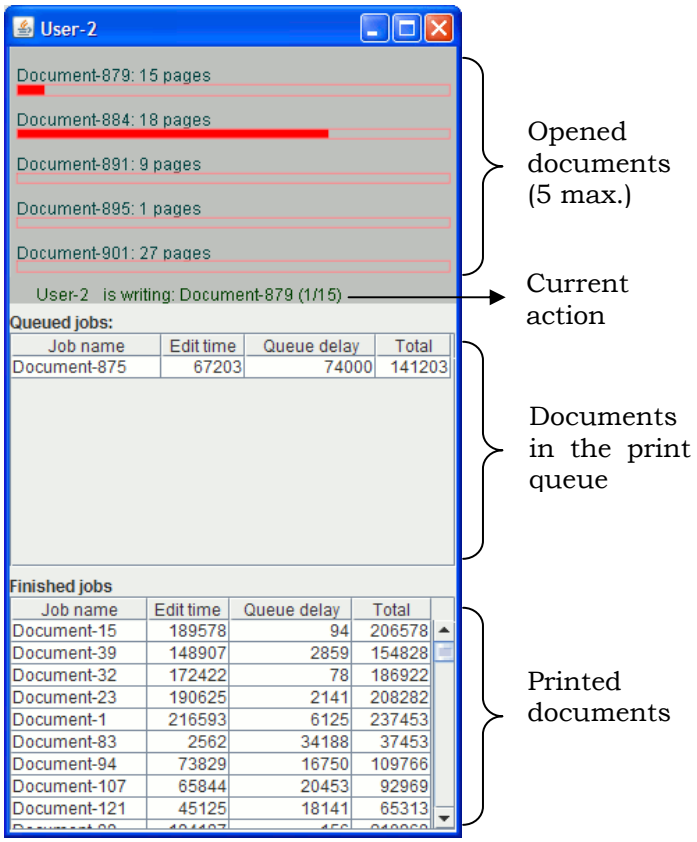


Figure 8-14: User's status windows

Figure 8-14 shows the status of a user in the Print shop. The window is divided into three parts. The first part contains the status of opened documents. For each document, there is a progress bar that indicates the completeness level. The status line under the status bar shows the current action of the user. When the user is preparing a document, this status line indicates which document is chosen and how many pages have finished. The second and third parts contain

two lists of documents: documents in the printing queue and documents that have been printed. Each line in these two lists contains the information about the document: document name (Job name), time spent in document preparation (Edit time), waiting time in the printing queue before the start of printing (Queue Delay) and total time spent for the document (Total – starts when the document is opened for writing and ends when the document is completely printed).

We also create status windows as shown in Figure 8-16 and Figure 8-18. There is only a progress bar indicating the printing status of the current printed document. This is followed by three statistical data items:

- Printed document: the number of documents (and number of pages) that have been printed on this printer since the start of the system.
- Free in (time): is the estimated amount of time after which the printer will be free. If a new document is added into the queue of this printer, this indicates, in fact, the estimated waiting time in the queue for the document to be actually printed.
- Pages in queue: total number of pages in the queue.

The following paragraphs describe the result of the simulation. We create a simple print shop that contains 10 computers and 3 printers as shown in Figure 8-15. We suppose that all printers have the same capabilities.

We will carry out two simulations:

- Without Availability Control: in which the Availability Control is switched off. This is equivalent to the situation of the Print Shop before the introduction of the Print Manager. To simulate the behavior of the users, print jobs are scheduled depending only on the relative position between the computer and printers. To make it more realistic, the user first chooses the nearest printer. He will take a look at the printer queue to see how many documents are in it. There are two possibilities. If there less than N documents, he will print on this printer. N is randomly chosen from 10 to 20 and differently for each user. If not, he will compare the queue of the nearest and the second nearest printer, and choose the one have less document in queue to print his document.

- With Availability Control: where Availability Control is switched on. As described above, this Print Manager is responsible in scheduling all the requested print jobs depending on the availability of available printers.

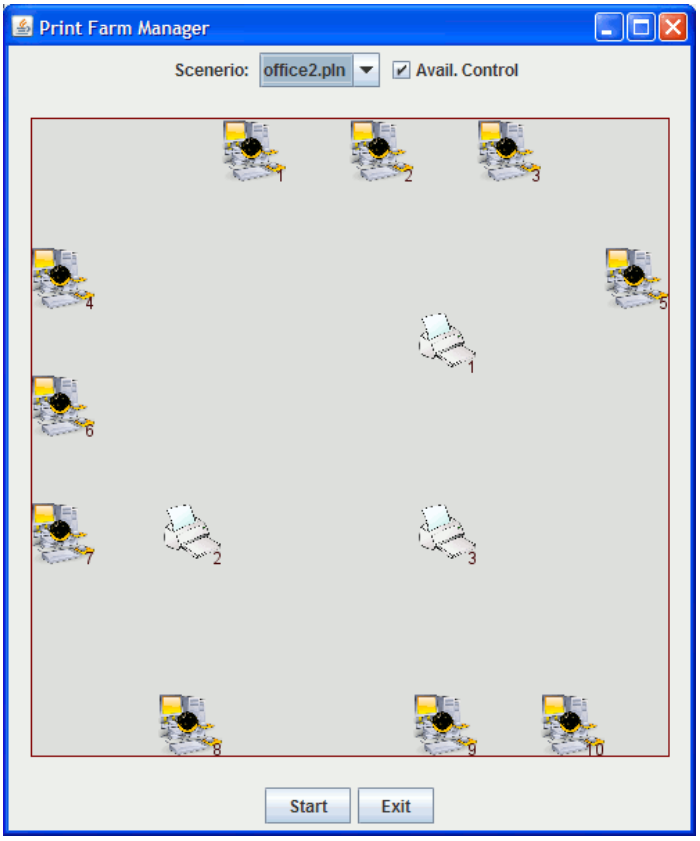


Figure 8-15: Simulation setup

Figure 8-16 and Figure 8-17 present the results for the first simulation, i.e. Without Availability Control.

Figure 8-16 presents the queue state of the three printers at a chosen moment. Since users can select any printer to send print jobs, this simulation seems to be unrealistic in terms of the hard-binding of computer-printer. However, it still reflects the current situation of the Print Shop, because the problem of the old implementation of the print shop favours the unbalanced job scheduling due to user preferences for the neighbour printers. We notice that at certain moment, Printer-3 receives fewer documents than Printer-1 and Printer-2. The consequence is that its queue is

shorter than that of the other two without being known by all the users. The documents in the queue of Printer-1 and Printer-2 will have to wait longer than they should have to.

Figure 8-17 shows the average time that a print job stays in the queue of the printer before being actually printed. The vertical axis is the Average Time in Queue (measured in milliseconds) and the horizontal axis is the ordinary time line. As we can see, the waiting time of documents in the three printers is all increasing. However, the documents in the queue of Printer-3 are expected to have to wait less time than those in the queue of other printers. These graphs are coherent with the status of the printing queue of printers, since the longer the queue is, the longer the documents in the queue (especially the newly added ones) have to wait before being effectively printed.

The simulated results reflect well the problem encountered in the old implementation of the print shop. In most runs, we had to stop the simulation at some moment, because the print queue of Printer-1 and Printer-2 was becoming too long. In reality, users should have already stopped sending documents to printers much sooner than we do in our simulation.

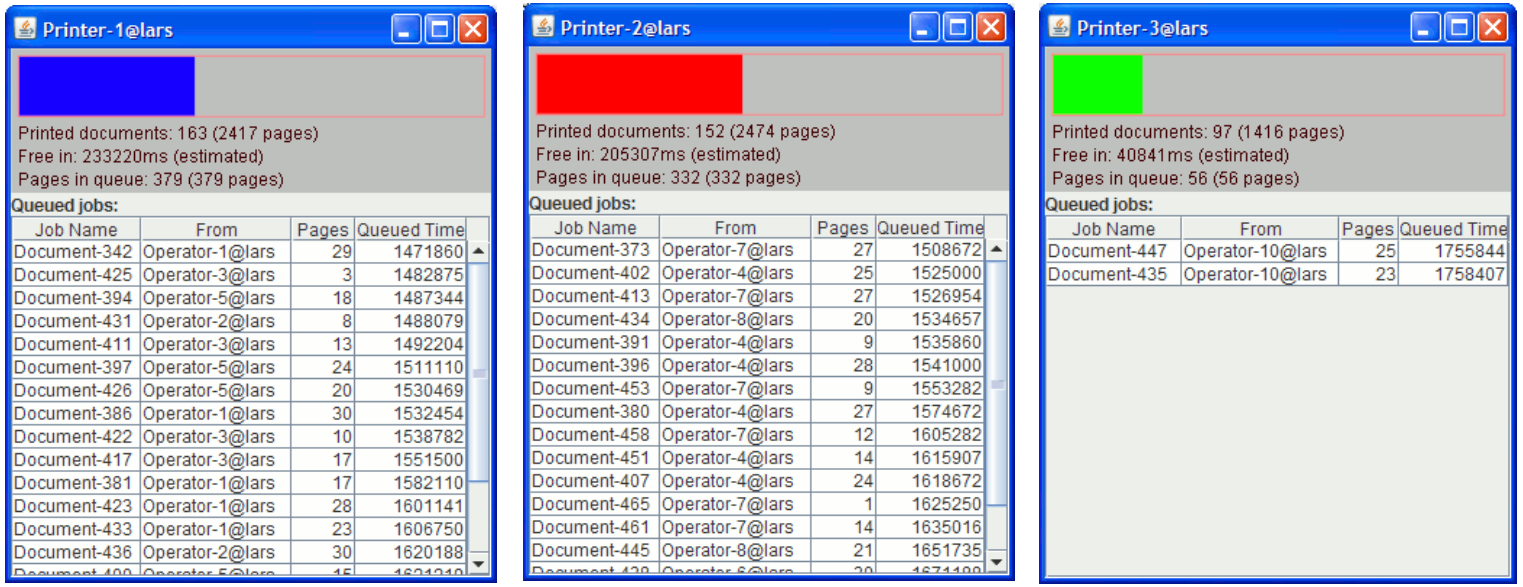


Figure 8-16: Printing queue of printers - Without availability control

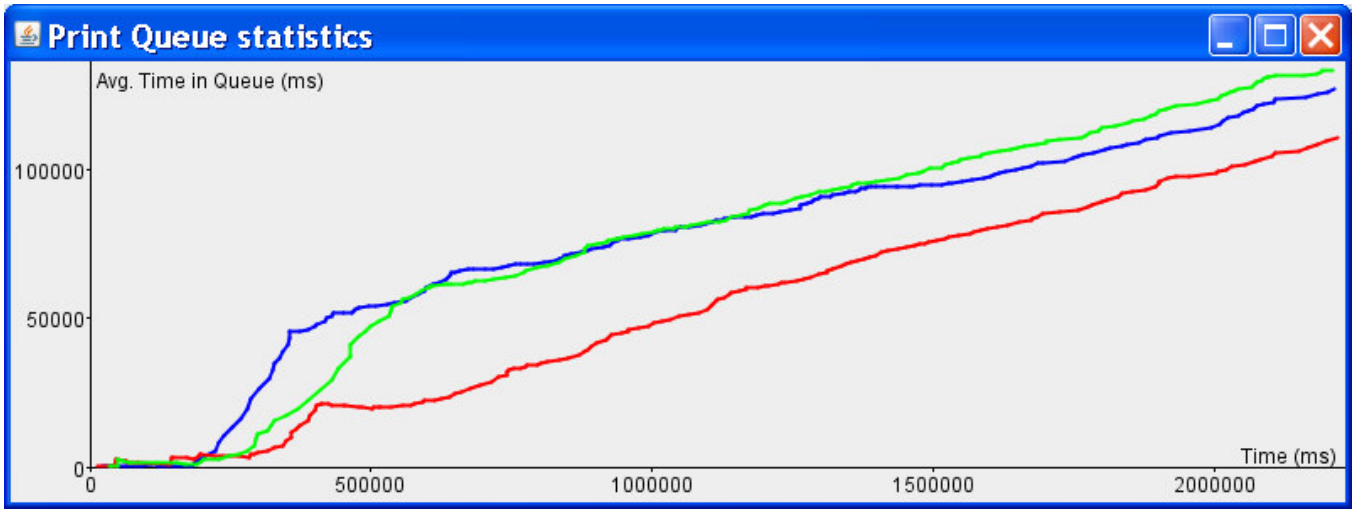


Figure 8-17: Average Waiting Time in Printing Queue – without availability control

Figure 8-18 and Figure 8-19 show the simulated results of the Print Shop when the Availability Control is switched on.

The first remark for this simulation is that we need not stop too early the simulation as we did in the first one. The queue size of the three printers is kept short, as one can see in Figure 8-18.

We should point out that our Print Manager is not simply a job balancing system. An ordinary job balancing does not usually take into account the physical positions of computers and printers which is an important factor for the Print Shop application. As we pointed out above that, when the area of the Print Shop is large and divided into several sections, the walking distance between the computer and the printer becomes significant. In our simulation, this distance is converted into additional waiting time by the introduction of the average walking speed. The choice of printer is actually based on the total waiting time that is calculated by the sum of estimated waiting time in queue and the estimated walking time. A user may have to take a longer walk to take its printed documents if all the neighbour printers are busy. In other cases, when the difference between the waiting times of neighbour printers is small, the Availability Manager will give preferences to nearer printers.

We see in Figure 8-19 that the waiting time in queue is shared among all the three printers. At 4000000 ms of simulation time, we can conclude that the waiting time in queue is stabilized between 40000 and 45000 milliseconds. This is indeed a better situation, compared to the previous simulation where, at 2000000 ms in simulation time in Figure 8-17, the waiting time in queue is already above 100000 milliseconds.

Figure 8-19 is also an example that shows the effect of walking time. Reconsidering the first simulation in which the relative distances between computers and printers decide the choice of printer, printing requests from computer 1-8 are likely sent to Printer-1 and Printer-2, which make them overloaded. In the second simulation, if the estimated queuing time in Printer-3 plus the additional walking time towards Printer-3 is smaller than the delay estimated for Printer-1 and Printer-2, print request is sent to the Printer-3 even if the requesting computer is nearer to Printer-1 and Printer-2 than to Printer-3. However, the Average Time in Queue in Printer-3 is likely smaller than that of Printer-1 and Printer-2, as shown in Figure 8-19, due to the additional walking time of users to reach Printer-3.

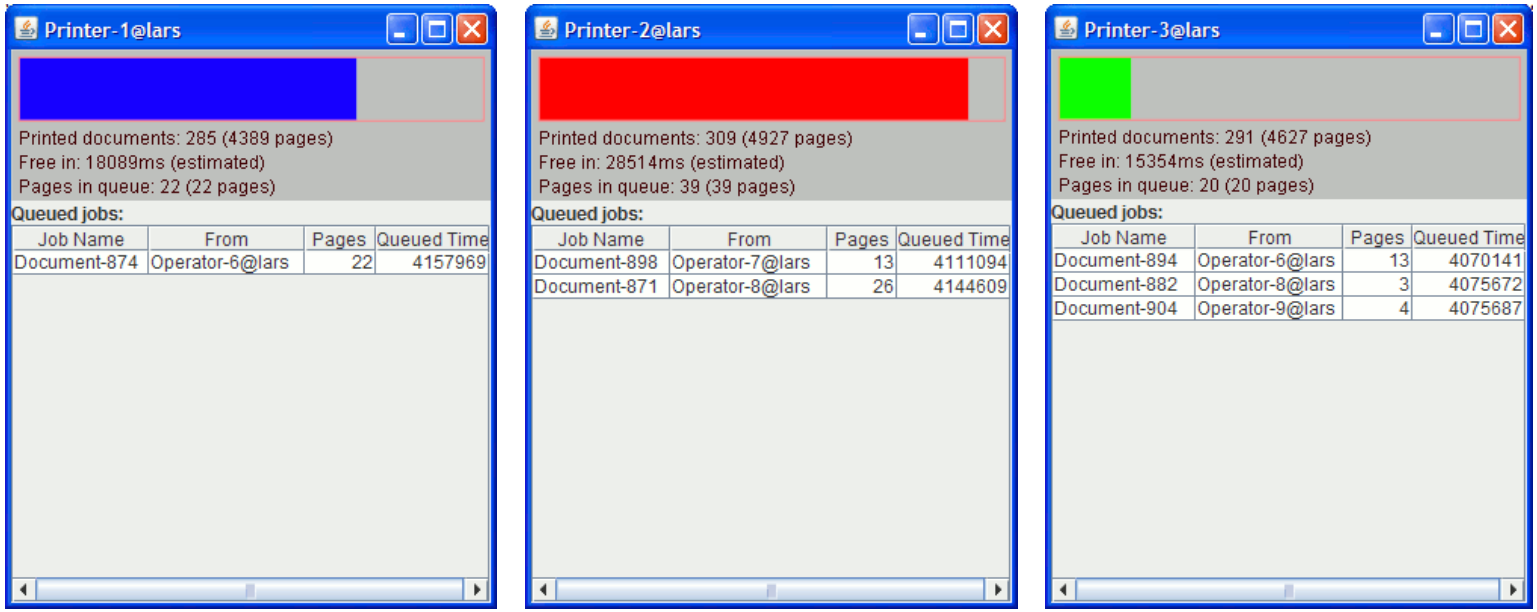


Figure 8-18: Printing queue of printers - With availability control

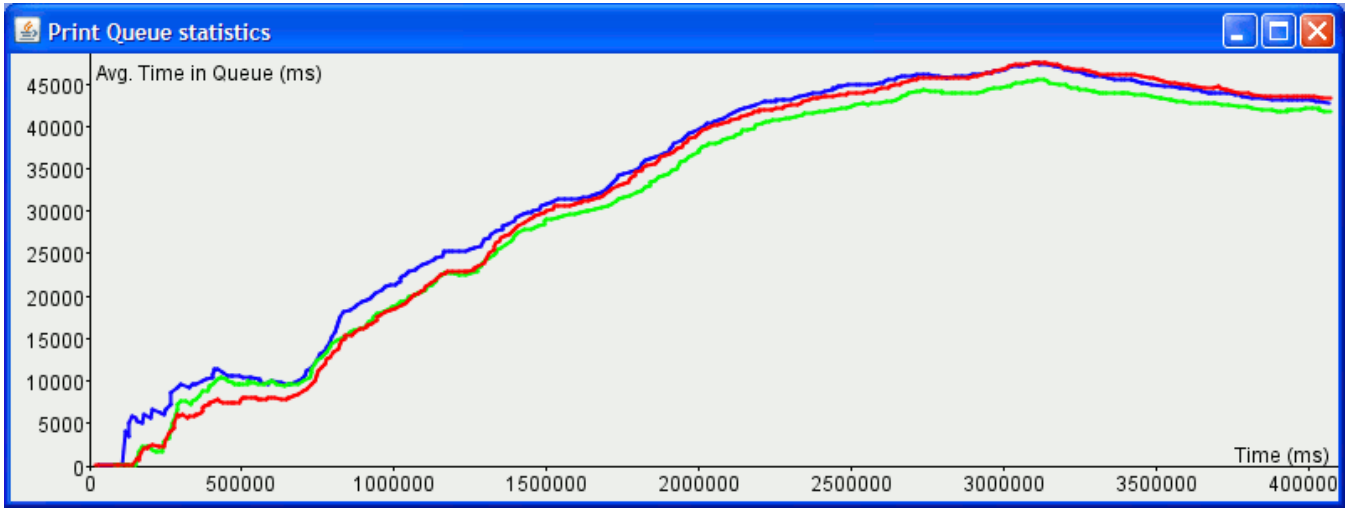


Figure 8-19: Average Waiting Time in Printing Queue – with availability control

The above simulation results are rather simple to be interpreted, which shows an improvement done by the Print Manager. Simulations for more complex shop plans, e.g. the shop plan in Figure 8-13 are also carried out. Considerable improvements can also be seen when the Print Manager is switched on in those cases. But since the results are much more complicated and exceed the scope of this thesis, they are omitted.

In this simulation, what we want to illustrate is not how effective the proposed solution is in terms of waiting time. For better simulations, one would record the true behaviors of the users in a real print shop and study more about the queuing theory to better analyze the simulated results in larger scales.

What we have illustrated in this case study is in terms of the treatment of quality requirements during the software life, i.e., during both development time and execution time. The conclusions about this can be summarized as follows:

- The techniques and analysis developed in Chapter 4 and Chapter 6, as well as in (HOANG, T.T.H. and Kolp, M., 2010), have real applicability. In Print Shop application, we have walked through all the most important phases while keeping track of the fulfillment of the Availability and Customizability requirements.
- The social patterns introduced in Chapter 5, as well as in (HOANG, T.T.H. and Kolp, M., 2009), can be used to construct a sub-system that supervises and controls the fulfillment of quality requirements, e.g., Availability requirement in the example.
- When quality requirements are correctly addressed, the performance of the system can be improved. The fulfillment of quality requirements must be considered not only during the development, e.g., good account and tracking of quality requirement using QTropos, but also during the operation of the system, e.g., quality control system at runtime as in the Print Shop example.

Chapter 9 CONCLUSIONS

In his recent book (JONES, C., 2009), Jones summarizes three popular approaches to define software quality: “conformance to requirements”, “reliability, portability and other –ilities”, and “absence of defects”. Our vision of software quality is a version of the first approach.

Our research argues that defining quality requirements as of a different nature than soft-goals helps strengthen the conformance to and the effectiveness of requirements. As far as we know, as we report in our literature reviews on goal-based requirements engineering (in Chapter 2) and on quality requirements in software engineering (in Chapter 3), this distinction is original.

Inspired by the notion of non-functional requirements in (CHUNG, L. et al., 2000) that are viewed as constraints on system functionality and treated as soft-goals, we define in Chapter 4 quality requirements as constraints on hard-goals and soft-goals.

The proposed definition of hard-goals, soft-goals, and quality requirements in Chapter 4 allows us not only to elicit, i.e., make explicit, quality requirements that are sometimes closely integrated inside soft-goals but also to enrich the AND/OR goal analysis, introduced by KAOS (DARIMONT, R. et al., 1997). In this refined goal analysis, quality requirements become a third type of nodes that can constrain the usual types of nodes (hard-goals and soft-goals) in the decomposition tree. In addition to the usual “And” and “Or” link types, three new types of links are introduced to connect quality requirement nodes to goal nodes. We call them elicitation, qualification, and contribution links. Examples and discussions throughout the thesis suggest that this enriched goal analysis improves the effectiveness of the transformation of the initial requirements into system goals.

A catalogue of social patterns, defined in Chapter 5, is meant to help design sub-systems for quality control at system runtime. This gives multi-agent systems the capability of monitoring the satisfaction of some quality requirements that cannot be fulfilled solely by goal analysis. Such a quality control sub-system gives more possibility for the system to better take into account the relevant requirements, identified and refined by our enriched goal analysis.

The combination of the enriched goal analysis and the catalogue of new social patterns thus helps to build better system products in terms of our perception of software quality. It is also used to extend Tropos to create Quality-aware Tropos (QTropos) in Chapter 6, built into a CASE tool for quality requirements (QCASE) in Chapter 7 and applied to a real-size case study of a Print Shop in Chapter 8.

We argue that the main idea developed in the thesis (namely the distinction that we make between soft-goals and quality requirements) and its consequences have the following positive effects on several aspects in an effective treatment of quality requirements:

i) Visibility of quality requirements

Separating like we do quality requirements from soft-goals alleviates some of the confusion created by the ambiguity and fuzziness of the notion of non-functional requirements when adapted to agent-based software development methodologies. This separation has a number of interesting consequences for developers:

- They can trace the evolution of a particular quality requirement from the moment it is elicited or explicitly introduced until a hard-goal or an operationalization of a hard-goal can take the resulting quality requirements into account.
- They can more easily identify during system analysis, during system design, as well as system implementation components (goals, soft-goals, their relationships, and also, e.g., in Tropos, taks, resources, etc.) that are constrained by or are introduced to fulfil quality requirements.

An example of such an enhanced visibility of requirements is exhibited in the Print Service case study in Chapter 8, where special views are extracted from the goal decomposition tree that focus on one particular quality requirement.

A greater visibility on quality requirements also enables to make all human actors in the software development process systematically aware of the quality requirements that need to be taken into account

when carrying out an action. This is crucial since other approaches allow only software analysts and designers to interact directly with quality requirements. Other actors like programmers and testers are not directly made clearly aware of those quality requirements and can only know about them indirectly (e.g., by searching informal documentation).

ii) Ubiquity of quality requirements

Separated from soft-goals, quality requirements can be propagated from the early phases to the late phases of the development process. This means that they can be used to improve every aspect of system design, implementation plan, deployment plan, operational tuning, etc.

With our refined analysis, the ubiquity of quality requirements does not simply mean their presence in separate phases. At a development phase, every quality requirement is, in fact, explicitly stated in a user requirement from the early requirement phases, elicited from a soft-goal or inherited from the analysis during the preceding phases. The presence of a quality requirement is ended only when it is sufficiently fulfilled, otherwise it should be automatically propagated to the next phase for further considerations.

iii) Flexibility of quality requirement analysis

In our approach, developers have all freedom to choose, at each analysis step, whether to take action to fulfil a quality requirement or to propagate the requirement to treatment in later stages. Automatic procedures described and implemented in QCase in Chapter 7 can be used to keep track of the consideration states of each quality requirement.

iv) Conformity to quality requirements

Methodologies like Tropos define quality requirements at the global level, i.e., at system level. This makes sense for requirements that indeed concern and constrain the global system.

In our approach, quality requirements are defined as constraints on goals. Moreover, quality requirements constrain only goals whose satisfaction can influence their satisfaction. The way that quality requirements are elicited and propagated down the decomposition tree shown in Chapter 4 and Chapter 6 allows us to pinpoint which components are actually responsible for fulfilling the required qualities. This precision, together with a greater usability of quality requirements as argued above, can help make software systems better conform to quality requirements.

Here are some possible directions that could be considered as research topics connected to this research:

- Much work on software quality originates from software engineering. An interesting question to address could be the specificity of the quality issues that arise for business information systems as contrasted to systems developed with software engineering techniques.
- In our approach, quality requirements are decoupled from goals. In this thesis, we used the Jadex programming language. Other programming schemes may be more suitable, like, for example, Aspect-Oriented Programming (KICZALES, G.J. et al., 2002) that could be explored and adapted to multi-agent systems.
- In Tropos, the organizational styles are used to regroup agents during the architectural design phase. We have applied those styles to QTropos without any modifications to incorporate our notion of quality requirements. These necessary modifications and the introduction of new quality-aware organizational styles are needed.
- We introduced a catalogue of social patterns that can be used for designing a quality control sub-system. A more complete catalogue of social patterns could be defined. To do so, the identification and definition of new social patterns for the development of agent systems that favor the awareness of quality requirements are still needed. Existing patterns from multi-agent system and from broader scopes (or from other disciplines) should be adapted to be made quality-aware.

Possible work following this thesis could include the following practical issues:

- The implementation of a module for generating code for systems developed with our QCase tool.
- The adaptation of our QCase tool to other modeling diagrams taken from other methodologies.
- The transformation of our QCase tool into distributed software.
- The application of the proposed approach to practical projects to draw more conclusions about its applicability.

REFERENCES

ACHARYA, S., M. FRANKLIN, and S. ZDONIK. 1997. Balancing push and pull for data broadcast. *ACM SIGMOD Record*. **26**(2), pp.183-194.

ANTÓN, A. 1996. Goal-based requirements analysis. *In*: *Proceedings ofthe International Conference on Requirements Engineering (ICRE)*. Colorado Springs, Colorado, pp.136-144.

ANTÓN, A. 1997. *Goal Identification and Refinement in the Specification of Information Systems*. PhD Thesis. Georgia Institute of Technology - NC State University.

ARIDOR, Y. and D.B. LANGE. 1998. Agent design patterns: Elements of agent application design. *In*: *Proceedings of the 2nd international conference on Autonomous agents*. New York, USA: ACM, pp.108-115.

ARMBRUST, M., A. FOX, R. GRIFFITH, and OTHERS. 2009. *Above the clouds: A berkeley view of cloud computing*. Technical Report - EECS Department, University of California, Berkeley.

BAUER, B., J. MULLER, and J. ODELL. 2001. Agent UML: A formalism for specifying multiagent interaction. *In*: *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE)*. Limerick, Ireland: Springer-Verlag, pp.91-104.

BELLIFEMINE, F., G. CAIRE, and D. GREENWOOD. 2008. *Developing multi-agent systems with JADE*. John Wiley & Sons.

BERNON, C., M.P. GLEIZES, S. PEYRUQUEOU, and G. PICARD. 2003. Adelfe: A methodology for adaptive multi-agent systems engineering. *Engineering Societies in the Agents World III - Lecture Notes in Computer Science*. **2577/2003**, pp.70-81.

BOEHM, B. 1986. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*. **11**(4), pp.14-24.

BOOCH, G., J. RUMBAUGH, and I. JACOBSON. 1999. *The Unified Modeling Language: User Guide*. Addison-Wesley.

BRESCIANI, P., A. PERINI, P. GIORGINI et al. 2004. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*. **8**(3), pp.203-236.

CABAC, L. and D. MOLDT. 2005. Formal semantics for AUML agent interaction protocol diagrams. *Agent-Oriented Software Engineering V - Lecture Notes in Computer Science*. **3382/2005**, pp.47-61.

CAIRE, G., W. COULIER, F. GARIJO et al. 2002. Agent oriented analysis using MESSAGE/UML. *Agent-Oriented Software Engineering II - Lecture Notes in Computer Science*. **2222/2002**, pp.119-135.

CASTRO, J., M. KOLP, and J. MYLOPOULOS. 2002. Towards requirements-driven information system engineering: the Tropos project. *Information System Journal*. **27**(6), pp.365-389.

CHENG, B. and J. ATLEE. 2009. Current and Future Research Directions in Requirements Engineering. *Design Requirements Engineering: A Ten-Year Perspective - Lecture Notes in Business Information Processing*. **14**(1), pp.11-43.

CHUNG, L. and J.C.P. LEITE. 2009. On non-functional requirements in software engineering. *Conceptual Modeling: Foundations and Applications - Lecture Notes in Computer Science*. **5600/2009**, pp.363-379.

CHUNG, L., A. B. NIXON, E. YU, and J. MYLOPOULOS. 2000. *Non-functional requirements in software engineering*. Kluwer Academic Publishers.

CLARKE, E.M., J.M. WING, R. ALUR, and OTHERS. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*. **28**(4), pp.626-643.

CLELAND-HUANG, J., R. SETTIMI, O. BENKHADRA et al. 2005. Goal-centric traceability for managing non-functional requirements. *In*: *Proceedings of the 27th International Conference on Software Engineering*. ACM, pp.362-371.

CNET. 2008. *Amazon suffers U.S. outage*. [online]. Available from World Wide Web: <http://news.cnet.com/8301-10784_3-9962010-7.html>

COCKBURN, A. 2007. *Agile software development: the cooperative game*. Addison-Wesley.

COSSENTINO, M. and C. POTTS. 2002. A CASE tool supported methodology for the design of multi-agent systems. *In*: *Proceeding of the International Conference on Software Engineering Research and Practice*. Las Vegas, Nevada, USA, pp.295-306.

DARDENNE, A., A. LAMSWEERDE, and S. FICKAS. 1993. Goal-directed requirements acquisition. *Science of computer programming*. **20**(1-2), pp.3-50.

DARIMONT, R., E. DELOR, P. MASSONET, and A. VAN LAMSWEERDE. 1997. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *In*: *Proceedings of the 19th International Conference on Software Engineering (ICSE)*. ACM, pp.612-613.

DARIMONT, R. and A. VAN LAMSWEERDE. 1996. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*. **21**(6), pp.179-190.

DAVIS, A.M. 1993. *Software requirements: objects, functions, and states*. New Jersey: Prentice-Hall.

DO, T. 2005. *A Framework for Multi-Agent Systems Detailed Design*. PhD Thesis. Université Catholique de Louvain.

DO, T.T., M. KOLP, T.T.H. HOANG, and A. PIROTTE. 2003. A Framework for Design Patterns for Tropos. *In*: *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES)*. Manaus, Brazil, pp.333-343.

DONZELLI, P. 2004. A goal-driven and agent-based requirements engineering framework. *Requirements Engineering*. **9**(1), pp.16-39.

ERL, T. 2005. *Service-oriented architecture: concepts, technology, and design*. NJ, USA: Prentice Hall.

FIPA. *The Foundation for Intelligent Physical Agents*. [online]. Available from World Wide Web: <http://www.fipa.org>

FRANKLIN, S. and A. GRAESSER. 1997. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *Intelligent Agents III Agent Theories, Architectures, and Languages - Lecture Notes in Computer Science*. **1193/1997**, pp.21-35.

FUXMAN, A., M. PISTORE, J. MYLOPOULOS, and P. TRAVERSO. 2001. Model Checking Early Requirements Specifications in Tropos. *In*: *Proceedings of the 5th IEEE Int. Symposium on Requirements Engineering*. Toronto, Canada: IEEE, pp.174-181.

GAMMA, E., R. HELM, R. JOHNSON, and J. VLISSIDES. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

GEORGIADOU, E. 2003. Software process and product improvement: a historical perspective. *Cybernetics and Systems Analysis*. **39**(1), pp.125-142.

GLINZ, M. 2005. Rethinking the notion of non-functional requirements. *In*: *Proceedings of the 3rd World Congress for Software Quality*. Munich, Germany, pp.55-64.

GLINZ, M. 2007. On Non-Functional Requirements. *In*: *Proceedings of the 15th IEEE International Requirements Engineering Conference*. IEEE, pp.21-26.

GLINZ, M. 2008. A risk-based, value-oriented approach to quality requirements. *IEEE Software*. **25**(2), pp.34-41.

GOTEL, O. and A FINKELSTEIN. 1994. An analysis of the requirements traceability problem. *In*: *Proceedings of the 1st International Conference on Requirements Engineering*. IEEE, pp.94-101.

GRADY, R.B. 1992. *Practical software metrics for project management and process improvement*. Upper Saddle River, New Jersey, USA: Prentice-Hall.

HANSEN, S., N. BERENTE, and K. LYYTINEN. 2009. Requirements in the 21st Century: Current Practice and Emerging Trends. *Design Requirements Engineering: A Ten-Year Perspective - Lecture Notes in Business Information Processing*. **4**(1), pp.44-87.

HENDERSON-SELLERS, B. and P. GIORGINI. 2005. *Agent-oriented methodologies*. IGI Global.

HOANG, T.T.H. 2008. *Quality-aware agent-oriented software development*. Internal Report, Université catholique de Louvain.

HOANG, T.T.H. and M. KOLP. 2009. Social Patterns for Quality Control in Multi-agent Systems. *In*: *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*. Sofia, Bulgaria: INSTICC Press, pp.336-343.

HOANG, T.T.H. and M. KOLP. 2010. Goal, Soft-goal and Quality Requirement. *In*: *Proceedings of ICEIS (International Conference on Enterprise Information Systems)*.

HOANG, T.T.H and M. KOLP. 2010. QTROPOS: Quality-Aware TROPOS. *Accepcted for IStar Workshop*.

IEEE. 1990. Glossary of software engineering terminology. *IEEE Standard*.

IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Standard 830*.

IGLESIAS, C.A., M. GARIJO, J.C. GONZALEZ, and J.R. VELASCO. 1997. Analysis and design of multiagent systems using MAS-CommonKADS. *In*: *Proceedings of Intelligent Agents IV: Agent Theories, Architectures, and Languages: 4th International Workshop*. Providence, Rhode Island, USA, pp.313-327.

ISO/IEC, 9126-1. 2001. *Software Engineering - Product Quanlity - Part 1: Quality Model*.

ISO-9000. 2000. Quality Management Systems - Fundamentals and Vocabulary. *International Organization for Standardization*.

JACK, Agent Oriented Software Pty. Ltd. 2002. *JACK intelligent agents - User guide*.

JONES, C. 1990. *Systematic Software Development Using VDM*. New York: Prentice Hall.

JONES, C. 2009. *Software Engineering Best Practices*. New York, USA: McGraw-Hill, Inc.

JUNG, H.W., S.G. KIM, and C.S. CHUNG. 2005. Measuring software product quality: a survey of ISO/IEC 9126. *Software IEEE*. **21**(5), pp.88-92.

JURETA, I.J., S. FAULKNER, and P.Y. SCHOBBENS. 2007. Achieving, Satisficing, and Excelling. *In*: *Proceedings of the Conference on Advances in Conceptual Modeling: Foundations and Applications*. Springer-Verlag, pp.286-295.

JURETA, I., J. MYLOPOULOS, and S. FAULKNER. 2008. Revisiting the core ontology and problem in requirements engineering. *In*: *Proceedings of the 16th IEEE International Requirements Engineering Conference*. Barcelona: IEEE, pp.71-80.

JURETA, I., J. MYLOPOULOS, and S. FAULKNER. 2009. A core ontology for requirements. *Applied Ontology*. **4**(3), pp.169-244.

KAN, S.H. 2002. *Metrics and models in software quality engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing.

KAVAKLI, E. 1999. *Goal-Driven Requirements Engineering: Modelling and Guidance*. PhD Thesis, University of Manchester.

KICZALES, G.J., J.O. LAMPING, C.V. LOPES et al. 2002. *Aspect-oriented programming*. Google Patents.

KNUTH, D.E. 1997. *The Art of Computer Programming. 3rd edition, Volume 1*. Addison-Wesley.

KOLP, M., T.T. DO, S. FAULKNER, and T.T.H. HOANG. 2005. Introspecting agent-oriented design patterns. *Advances in Software Engineering and Knowledge Engineering*. **III**, p.105–134.

KOLP, M., P. GIORGINI, and J. MYLOPOULOS. 2001. A goal-based organizational perspective on multi-agents architectures. *In*: *Proceedings of the 8th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages*. Seattle, USA, pp.128-140.

KOLP, M. and J. MYLOPOULOS. 2001. Software architectures as organizational structures. *In*: *Proceedings of ASERC Workshop on The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems*. Edmonton, Canada.

LANDES, D. and R. STUDER. 1995. The treatment of non-functional requirements in MIKE. *Software Engineering ESEC'95 - Lecture Notes in Computer Science*. **989/1995**, pp.294-306.

LETIER, E. 2001. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis. Université catholique de Louvain. Belgium.

LEVESON, N.G. 2004. Role of software in spacecraft accidents. *Journal of spacecraft and Rockets*. **41**(4), pp.564-575.

LIONS, J.L. and et AL. 1996. Ariane 5 flight 501 failure. *Report by the Inquiry Board*.

LIU, L. and E. YU. 2001. From Requirements to Architectural Design – Using Goals and Scenarios. *In*: *Proceedings of the 1st International Workshop From Software Requirements to Architectures*. Toronto, Canada, pp.22-30.

MARTIN, J. 1991. *Rapid application development*. Prentice-Hall: Macmillan Publishing.

MASOLO, C., S. BORGO, A. GANGEMI et al. 2003. *Ontology library - WonderWeb Deliverable D18*. Technical Report - Laboratory For Applied Ontology.

MCCALL, J.A., P.K. RICHARDS, and G.F. WALTERS. 1977. *Factors in software quality*. Technical Report RADC-TR-77-369, US Department of Commerce: The National Technical Information Service.

MOURATIDIS, H. and P. GIORGINI. 2007. Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*. **17**(2), pp.285-309.

MYLOPOULOS, J., L. CHUNG, S. LIAO et al. 2001. Exploring alternatives during requirements analysis. *Software IEEE*. **18**(1), pp.92-96.

NAUR, P. and B. RANDELL. 1968. Software Engineering: Report of a conference sponsored by the NATO Science Committee. *Scientific Affairs Division, NATO*. **7**(11).

NCUBE, C. 2000. *A Requirements Engineering Method for COTS-Based Systems Development*. PhD Thesis - City University London.

NORIN, L.A. and F. STOCKEL. 1998. *Open-source software development methodology*. Technical Report - Lulea University of Technology.

NUSEIBEH, B. and S. EASTERBROOK. 2000. Requirements engineering: a roadmap. *In*: *Proceedings of the Conference on the Future of Software Engineering*. New York, USA: ACM, pp.35-46.

NWANA, H. S. 1996. Software agents: An overview. *The Knowledge Engineering Review*. **11**(3), pp.205-244.

ODELL, J. 2000. *Agent technology*. Green Paper of the OMG Agent Working Group.

ODELL, J., H. VAN DYKE PARUNAK, and B. BAUER. 2000. Extending UML for agents. *In*: *Proceedings of the Agent-Oriented Information Systems Workshop*. Austin, USA: ICue Publishing, pp.3-17.

PADGHAM, L. and M. WINIKOFF. 2003. Prometheus: A methodology for developing intelligent agents. *In*: *Proceedings of the 3rd international conference on Agent-oriented software engineering*. Springer-Verlag, pp.174-185.

PAECH, B., and D. KERKOW. 2004. Non-Functional requirements engineering-Quality is essential. *In*: *Proceedings of Requirements Engineering: Foundation for Software Quality (REFSQ)*. Citeseer, pp.237-250.

PAVON, J. and J. GOMEZ-SANZ. 2003. Agent oriented software engineering with INGENIAS. *In*: *Proceedings of the 3rd Central and Eastern European conference on Multi-agent systems (CEEMAS)*. Berlin: Springer-Verlag, pp.394-403.

PCWORLD. 2009. *Gmail Outage Marks Sixth Downtime in Eight Months*. [online]. Available from World Wide Web: <http://www.pcworld.com/article/160153/gmail_outage_marks_sixth_downtime_in_eight_months.html>

POKAHR, A., L. BRAUBACH, and W. LAMERSDORF. 2005. Jadex: A BDI reasoning engine. *Multiagent Systems, Artificial Societies and Simulated Organizations*. **15**(2), pp.149-174.

POSLAD, S. and P. CHARLTON. 2006. Standardizing agent interoperability: The FIPA approach. *Multi-Agent Systems and Applications - Lecture Notes in Computer Science*. **2086/2006**, pp.98-117.

RAO, A.S. and M.P. GEORGEFF. 1995. BDI agents: From theory to practice. *In*: *Proceedings of the 1st international conference on multi-agent systems (ICMAS)*. San Francisco, CA, US: The MIT Press, pp.312-319.

REEVES, C.A. and D.A. BEDNAR. 1994. Defining quality: alternatives and implications. *Academy of Management Review*. **19**(3), pp.419-445.

REGEV, G. and A. WEGMANN. 2005. Where do goals come from: the underlying principles of goal-oriented requirements engineering. *In*: *Proceedings 13th IEEE International Conference on Requirements Engineering*. IEEE Computer Society, pp.353-362.

ROBERTSON, S. and J. ROBERTSON. 1999. *Mastering the requirements process*. Addison Wesley.

ROMAN, G.C. 1985. A taxonomy of current issues in requirements engineering. *Computer*. **18**(4), pp.14-23.

ROPPONEN, J. and K. LYYTINEN. 2000. Components of software development risk: How to address them? A project manager survey. *IEEE Transactions on Software Engineering*. **26**(2), pp.98-112.

RUMBAUGH, J. 1995. What is a method? *Journal of Object Oriented Programing (JOOP)*. **8**(6), pp.10-16.

SCHREIBER, G., B. WIELINGA, R. DE HOOG et al. 1994. CommonKADS: a comprehensive methodology for KBS development. *IEEE Expert Intelligent Systems and Their Applications*. **9**(6), pp.28-37.

SEARLE, J.R. 1969. *Speech acts: An essay in the philosophy of language*. Cambridge University Press.

SHOHAM, Y. 1993. Agent-oriented programming. *Artificial intelligence*. **60**(1), pp.51-92.

SIMON, H.A. 1955. A behavioral model of rational choice. *The quarterly journal of economics*. **69**(1), pp.99-118.

SLAUGHTER, S.A., D.E. HARTER, and M.S. KRISHNAN. 1998. Evaluating the cost of software quality. *Communications of the ACM*. **41**, pp.67-73.

SOFFER, P. and Y. WAND. 2005. On the notion of soft-goals in business process modeling. *Business Process Management Journal*. **11**(6), pp.663-679.

SOMMERVILLE, I. 2007. *Software engineering*. Addison-Wesley.

SPILLNER, A. 2001. The W-model--Strengthening the bond between development and test. *In*: *Proceedings of STAReast*. Germany: University of Applied Sciences, pp.6-15.

SPIVEY, J. M. 1988. *Understanding Z: a specification language and its formal semantics*. New York: Cambridge University Press.

SUDEIKAT, J., L. BRAUBACH, A. POKAHR, and W. LAMERSDORF. 2005. Evaluation of Agent--Oriented Software Methodologies--Examination of the Gap Between Modeling and Platform. *Agent-Oriented Software Engineering V - Lecture Notes in Computer Science*. **3382/2005**, pp.126-141.

Twitter. [online]. Available from World Wide Web: <www.twitter.com>

VAN LAMSWEERDE, A. 2000. Requirements engineering in the year 00: A research perspective. *In*: *Proceedings of the 22nd International Conference on Software Engineering*. ACM, pp.5-19.

VAN LAMSWEERDE, A. 2002. Goal-oriented requirements engineering: A guided tour. *In*: *Proceedings of the 5th IEEE International Symposium on Requirements*. Washington, DC, USA: IEEE Computer Society, pp.249-262.

VAN LAMSWEERDE, A. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

WIEGERS, K.E. 2003. *Software Requirements*. USA: Microsoft Press Redmond.

WOOD, M.F. and S.A. DELOACH. 2001. An overview of the multiagent systems engineering methodology. *Agent-Oriented Software Engineering - Lecture Notes in Computer Science*. **1957/2001**, pp.1-53.

WOOLDRIDGE, M.J. and N.R. JENNINGS. 1995. Intelligent agents: theory and practice. *Knowledge Engineering Review*. **10**(2), pp.115-152.

WOOLDRIDGE, M.J. and N.R. JENNINGS. 2002. Software engineering with agents: Pitfalls and pratfalls. *IEEE on Internet Computing*. **3**(3), pp.20-27.

WOOLDRIDGE, M.J., N.R. JENNINGS, and D. KINNY. 2000. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*. **3**(3), pp.285-312.

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

YU, E. 1997. Towards modelling and reasoning support for early-phase requirements engineering. *In*: *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*. Washington D.C., USA, pp.226-235.

YU, E., P. GIORGINI, N. MAIDEN, and J. MYLOPOULOS. 2011. *Social Modeling for Requirements Engineering*. Edited by Eric Yu, Paolo Giorgini, Neil Maiden and John Mylopoulos.

YU, E. and L. LIU. 2001. Modelling trust for system design using the i* strategic actors framework. *Trust in Cyber-Societies - Lecture Notes in Computer Science*. **2246/2001**, pp.175-194.

YU, E. and J. MYLOPOULOS. 1996. Using goals, rules and methods to support reasoning in business process reengineering. *International Journal of Intelligent Systems in Accounting Finance and Management*. **5**(1), pp.1-13.

YU, E. and J. MYLOPOULOS. 1998. Why goal-oriented requirements engineering. *In*: *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*. Pisa, Italy: Dubois E., Opdahl A.L., Pohl K., eds. Presses Universitaires de Namur, pp.15-22.

ZAVE, P. 1997. Classification of research efforts in requirements engineering. *ACM Computing Surveys (CSUR)*. **29**(4), pp.315-321.

ZAVE, P. and M. JACKSON. 1997. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. **6**(1), pp.1-30.

Appendix A THE TROPOS METHODOLOGY

The Tropos software development method (CASTRO, J. et al., 2002) proposes to use requirements as the driving force to guide the development process through system analysis, design, and implementation. Tropos adopts the *i** framework from (YU, E., 1995) to model intentional and social concepts: actor, hard-goal, soft-goal, resource, task, dependency, plan, capability, and belief. These concepts are used consistently through four phases of development, namely analysis of early requirements, analysis of late requirements, architecture design, and detailed design. Goal decomposition, means-end analysis, and predefined design patterns are the principal development tools. The process is completed with the implementation step preferably with an agent-oriented programming language that implements the BDI (belief, desire, and intention) paradigm (RAO, A.S. and Georgeff, M.P., 1995). Jadex (POKAHR, A. et al., 2005) and Jack (JACK, Agent Oriented Software Pty. Ltd., 2002) are the most popular programming platforms using Java technology.

This appendix reviews the essentials of the i* framework and of the Tropos methodology. Tropos is extended with our quality-aware Tropos (Q-Tropos) proposed in Chapter 6.

A.1. The i* Framework

The i* framework (YU, E., 1995) (YU, E., 1997) was introduced to model and to reason about information systems and their environment. It provides two main models: the strategy dependency model and the strategic rationale model.

Strategy dependencies model dependencies among stakeholders, represented as actors in i*, in the organisational context.

Dependencies make explicit the interactions among stakeholders in the context of the information system.

Strategic rationale analysis shifts the focus to the internal structure of stakeholders. It analyses their interests and concerns towards their dependencies with others and towards the information system.

A.1.1. Strategic Dependency

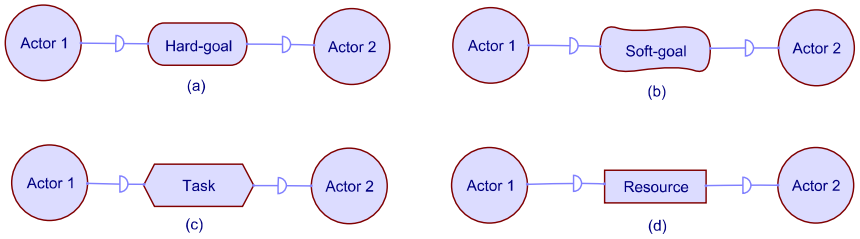


Figure A-1: i* dependency

In the strategic dependency model, the so-called notion of "distributed intentionality" models the overall intention of a group of actors. A dependency is formed if an actor (called "depender") depends on another actor (called "dependee") to acquire a "dependum". There are four types of "dependum": hard-goal, soft-goal, task, and resource. Note that the i* and Tropos literature usually abbreviates "hard-goal dependency" as "goal dependency".

In i*, actors are represented by a filled circle as shown in Figure A-1, where Actor 1 plays the role of depender and Actor 2 is the dependee.

In short, a hard-goal is a well-defined state that the depender wants the dependee to achieve. In a hard-goal dependency (see Figure A-1 (a)), represented by a filled oval shape, the depender specifies only the desired results but does not make explicit how the dependee realizes them nor which operations will be carried out. Hard-goals are those that can be totally achieved (satisfied) if the appropriate tasks are carried out while soft-goals can be only partially achieved ("satisficed"). Soft-goal dependencies are represented by filled irregular curvilinear shapes as in Figure A-1(b).

A task, represented by a filled hexagon (as in Figure A-1(c)), is an operation that the depender wants the dependee to carry out.

A resource, represented by a filled rectangle (as in Figure A-1(d)), is a physical or informational entity that can be delivered by the depender to the dependee.

Two actors can have any number of dependencies between them. One actor can play the role of depender in one dependency and that of dependee in another one.

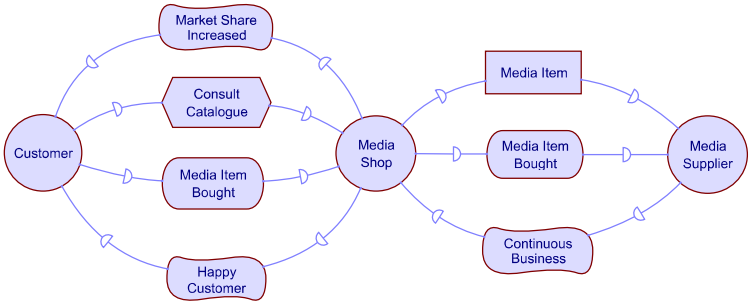


Figure A-2: Agent dependencies in Mediashop case study

In the Mediashop example presented in (CASTRO, J. et al., 2002), there are three main actors: “Customer”, “Mediashop”, and “Media Supplier”. In Figure A-2, “Customer” depends on “Mediashop” to “Consult Catalogue” and to “Media Items Bought”. In turn, “Mediashop” needs “Customer” to have “Market Share Increased” and “Happy Customers”. It also depends on “Media Suppliers” for “Media Items Bought”. “Media Supplier” depends on “Mediashop” to maintain a “Continuous Business”. Among the goals, “Market Share Increased”, “Happy Customers”, and “Continuous Business” are soft-goals.

A.1.2. Strategic Rationale

While the strategic dependency model describes inter-actor relations, the strategic rationale model analyzes the internal motivations of actors to explain the external dependencies.

A strategic rationale diagram can be considered as a detailed version of the corresponding strategic dependency diagram, in which the anatomy of each actor is made explicit.

Strategic rationale diagrams are expressed on the nodes introduced in dependencies: hard-goal, soft-goal, task, and resource. When an actor is analyzed, each dependency link between a dependum and the actor can be replaced by another dependency link between the dependum and an internal node inside the actor.

In addition, to describe a state of the world that the actor knows to be true regardless of its intentions, i* introduces the notion of belief. Beliefs can be any conventions, regulations, natural laws, etc. that the actor has to comply with, to adapt itself to or to use in its favour. A belief is represented with a cloud shape in the i* framework.

Figure A-3: Belief node

Essential constructs in the strategic rationale model are three types of links: means-ends links, task-decomposition links, and contribution links, that are introduced in the following subsections.

i) Means-Ends link

A means-ends link indicates a relationship between an end – which is a hard-goal to be achieved – and a means for attaining it – which is a task (YU, E., 1995).

A means-ends link is represented graphically by a solid line with a hollow arrowhead indicating the direction from the task to the hard-goal.

In Figure A-4, the "Message Communicated" hard-goal can be satisfied by carrying out "Send a Letter by Post" or "Send an Email". When multiple means-ends links are used to analyze a hard-goal, they represent alternatives to fulfil the hard-goal.

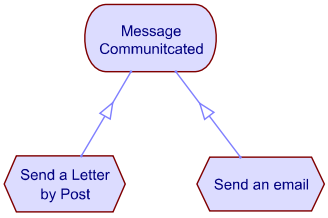


Figure A-4: Means-Ends link

ii) Task Decomposition link

In the strategic rationale model, a task represents an operation that the owner actor wants to carry out in a particular way. How a task is carried out is described by task decomposition links.

A task can be decomposed in hard-goals, soft-goals, tasks, and resources. The decomposition is graphically represented as solid lines with hollow arrowheads with an additional bar as shown in Figure A-5. In order to "Buy a Music CD", one has to "Check the Availability of the CD", to "Check-out the CD" and to "Securely Issue the Payment".

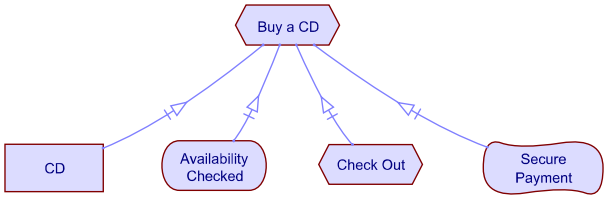


Figure A-5: Task decomposition links

iii) Contribution Links

Since soft-goals can only be satisfied to a certain degree (satisficed), one way to analyze them is to introduce "contribution links" that relate a soft-goal to other nodes that contribute in the fulfilment of soft-goal.

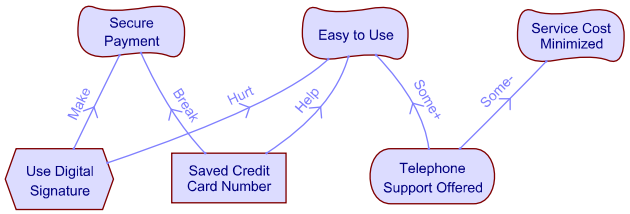


Figure A-6: Contribution links

In practice, conflicts are frequent between soft-goals. Figure A-6 exhibits an example where "Use Digital Signature" is said to make "Secure Payment" possible. However, it makes the payment less "Easy to Use". "Saved Credit Card Number" simplifies the payment process but makes the payment insecure. "Telephone Support Offered" will help users in their transactions, to the detriment of "Service Cost Minimized".

The i* framework proposes seven labels for contribution links:

- **Make**: a sufficiently positive contribution to satisfice a soft-goal. Example: "Use Digital Signature" is believed to be able to satisfice the "Secure Payment" soft-goal.
- **Help**: a positive contribution to a soft-goal not strong enough to satisfice it. Example: "Saved Credit Card Number" helps the payment to be "Easy to Use".
- **Some+**: a positive contribution whose strength is unknown. Example: "Telephone Support Offered" contributes positively to the "Easy to Use" soft-goal.

- **Null** (or unknown): a contribution whose effect is not yet known to be positive or negative.
- **Break**: a sufficiently negative contribution that breaks the fulfilment of a soft-goal. Example: "Saved Credit Card Number" may break "Secure Payment".
- **Hurt**: a negative contribution that is not strong enough to break the fulfilment of a soft-goal. Example: "Use Digital Signature" may complicate the payment process.
- **Some**-: a negative contribution whose strength is not yet well known. Example: "Telephone Support Offered"' hampers "Service Cost Minimized".

iv) Goal Decomposition

In i*, decompositions of hard-goals and soft-goals were made only through three types of links: means-ends, task decomposition, and soft-goal contribution. However, hard-goals and soft-goals are often decomposed into sub-goals. This was an important extension to i* when it was used in Tropos. Since hard-goals and soft-goals represent states of the world to be achieved, sub-goals are smaller states that contribute to the fulfilment of their parent goal through AND/OR relations. These AND/OR decompositions are borrowed from KAOS (DARIMONT, R. et al., 1997).

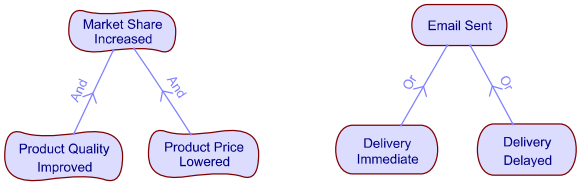


Figure A-7: Goal decomposition

There are some remarks on the usage of AND/OR Decomposition. First, it is meaningless to mix AND and OR links in a decomposition. Second, AND and OR links are usually reserved for hard-goals and soft-goals. However, in i*, since AND and OR links can be used as abbreviations for some decomposition chains, other types of elements can appear as well. Figure A-8 shows an example where two analyses (a) and (b) can be considered equivalent. In Figure (a), by using a chain of three standard links, one can create two alternative tasks: "Allocate More Developers" or "Engage Subcontractors" to satisfice the "Development Time Reduced" soft-goal. The same effect can be obtained by using just OR links in

Figure (b). It is not difficult to create examples in which AND links can be used to reduced the complexity of the i* diagram.

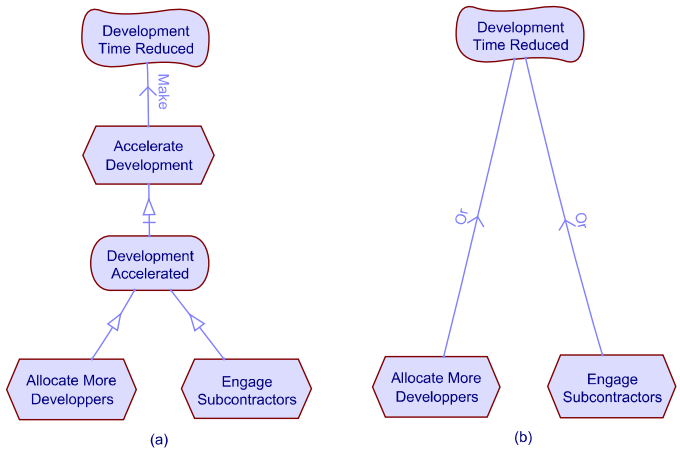


Figure A-8: OR decomposition

However, using AND/OR links with other elements than goals are rarely used and should be avoided since hard-goals and soft-goals represent the intentions and desires of actors while tasks and resources represent the operationalization of intensions.

A.2. The Tropos Methodology

Many software projects have failed to build a product that matches their initial requirements. This is because in traditional development processes:

- modelling tools for requirements and those for software cannot be easily integrated.
- programming notions are not distinct enough from notions model requirements.

In both issues, the traditional culture of requirement engineers is still largely to privilege concerns about the implementation of the future system. Requirements are often approximated too early by system functions in the development process.

Tropos (CASTRO, J. et al., 2002) was proposed as a modelling tool to better address the modelling of requirements and their easier transformation into agent-oriented software using agent programming languages such as Jack Agent (JACK, Agent Oriented Software Pty. Ltd., 2002) and Jadex (POKAHR, A. et al., 2005).

With i*, developers can model high-level intentions of stakeholders to explain why software should be built, in the early requirement phase. This phase is followed phases devoted to the analysis of late requirements, architectural design, and detailed design, which depends extensively on i* notions as well as Agent Unified Modelling Language (AUML) (ODELL, J. et al., 2000). The implementation is rather straightforward from the design by using BDI agent programming language (RAO, A.S. and Georgeff, M.P., 1995). In this section, we go quickly through all the main development phases of Tropos. These phases will be revised in Chapter 6 when quality requirements in Tropos are reconsidered.

This section presents a simple example of developing a third-party payment system for online shops on the internet.

A.2.1. Early Requirement

Software development methodologies usually start with a phase of requirement analysis. Tropos comprises both an “early requirement” phase and a “late requirement” phase.

The early requirement phase concentrates on describing the environment in which the future system will live. It also tells us why the system needs to be built, and it justifies the design and implementation options of the future function as driving forces during the development process.

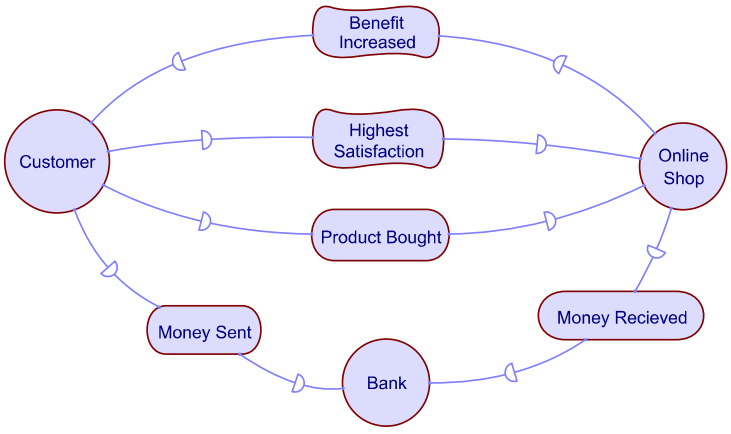


Figure A-9: Early Requirements - Strategic Dependency

Figure A-9 shows a simplified environment with three principal actors: “Customer”, “Online Shop”, and “Bank”. The principal relationships are between “Customer” and “Online Shop”. “Customer” depends on “Online Shop” to buy desired products and

to enjoy “Highest Satisfaction” about products bought. However, since “Online Shop” has not been equipped with an online payment system, “Customers” need to send a transfer order to their bank to finalize transactions. This could delay the delivery of the merchandises.

Figure A-10 exhibits an analysis of “Online Shop” of Figure A-8 obtained by using strategic rationale. Offering its own online payment service is considered as a tool to meet the shop objectives of “Cost Decreased” and “More Customers Attracted”. Figure A-10 shows the analysis of only the “Online Shop” actor. A similar analysis could be performed on the “Customer” actor to show that “Fast Delivery” could be an important factor contributing to the “Highest Satisfaction” of “Customers” and that “Customers” will be happy to be able to finalize their transactions right away after selecting a product without having to go through their bank.

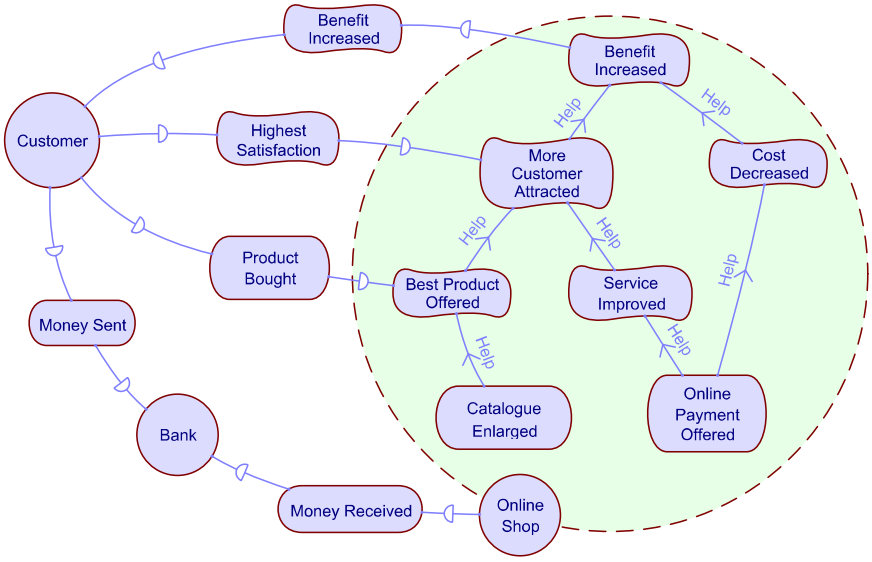


Figure A-10: Early Requirements - Strategic Rationale

The two diagrams show a situation in which, if both “Online Shop” and its “Customers” were offered “Online Payment” service, they would be both beneficial. In order to confirm this and to clarify what is needed for such a service, we consider, in the “late requirement” phase, a new situation where the intended system is inserted.

A.2.2. Late Requirement

While the “early requirement” phase is crucial for understanding why the system should be built, the “late requirement” phase is also very

important in answering the question: *if a system is to be built, what do the other actors expect it to offer?*

Again, strategic dependency and strategic rationale are applied to introduce, for the first time in the process, the intended system as a new actor. Dependencies between other actors and the future system, in the strategic dependency analysis, are drawn to trigger the analysis of the inside of the future system.

Based on Figure A-9, we add a new actor named “Payment System” to the strategic dependency model in Figure A-11. New dependencies are added between “Payment System” and other actors. Some dependencies are removed to reflect the new situation.

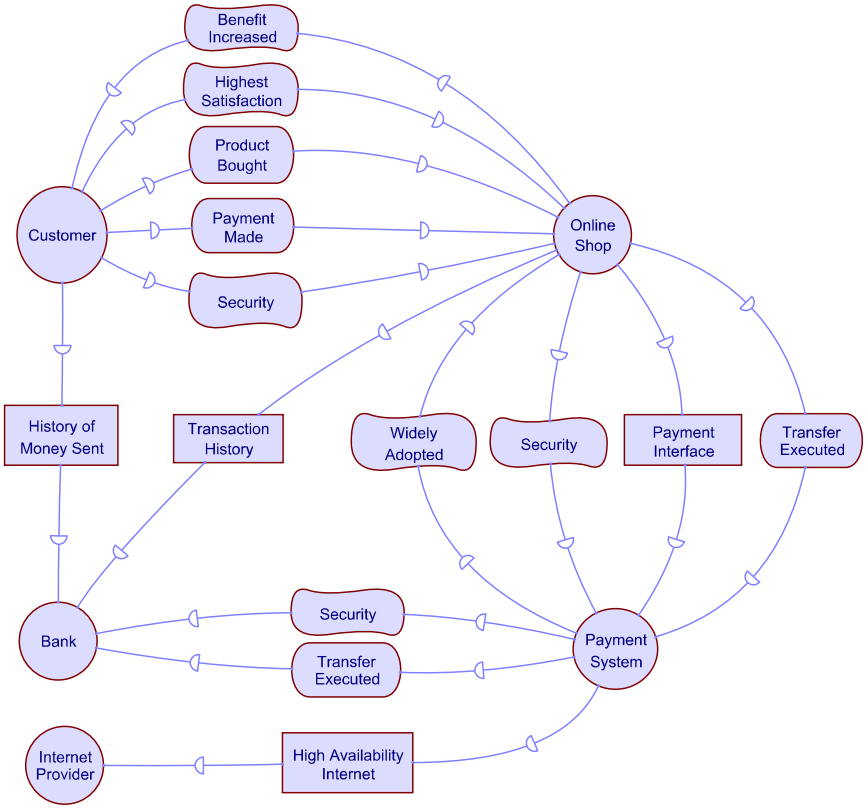


Figure A-11: Late Requirements - Strategic Dependency

“Customers” no longer need to request their “Bank” for money transfers to finalize their transactions with “Online Shop”. Thanks to the new “Payment System”, when customers want to check out their electronic caddies, “Online Shop” will propose an option to pay through the new “Payment System”. If accepted, it will ask the

"Payment System" to provide a "Secure Interface" on which "Customers" can request directly the necessary payment. Since everything is automatically and securely treated from the "Payment System" and the corresponding "Bank", transfers can be executed quickly. Payment arrival is notified directly to "Online Shop" that will, in turn, trigger the delivery of the merchandises.

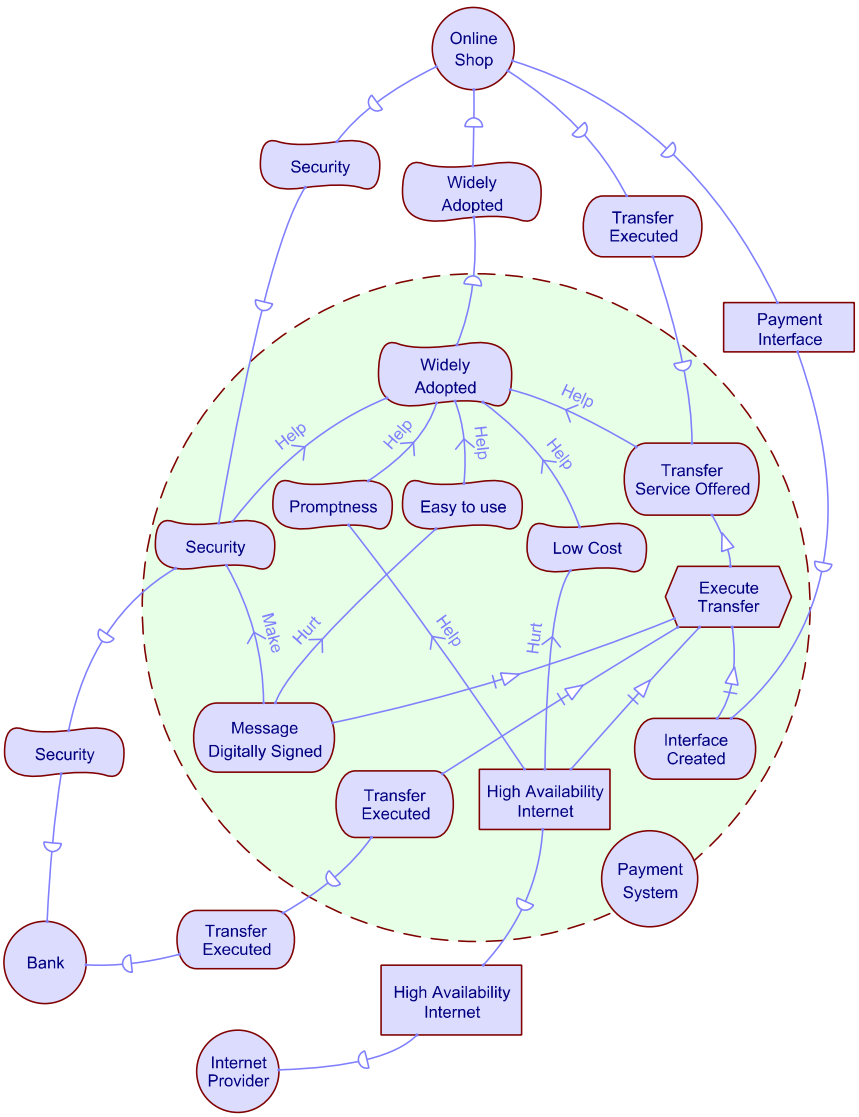


Figure A-12: Late Requirements - Strategic Rationale

Figure A-12 shows a strategic rationale analysis at the late requirement phase with the focus on “Payment System”. Every dependum with “Payment System” from the strategic dependency is retained and connected to an internal element of the “Payment System” actor.

From the point of view of building “Payment System”, the most important objective is to create payment facilities that will be “Widely Adopted” by online shops on the internet. To do so, being a payment system, the system must, of course, offer competitive facilities for money transfer (“Transfer Services Offered”). In addition, the service must be secure (“Security”), prompt (“Promptness”), and “Easy to Use” while keeping “Low Cost” for its clients.

In Tropos, qualities like security or promptness are treated in the widest scope. This implicitly means that all the elements inside the Payment System actor are constrained to have these qualities. Since the “Execute Transfer” task has to be secure, “Message Digitally Signed” is included in its decomposition. A “High Availability Internet” connection insures that customers can easily access the service and that the system can connect to banks to request transfers.

The system does not make money transfers itself; it actually depends on banks to do it. To guarantee the full security of the whole system, connections between the system and banks should be in high security.

Figure A-12 presents a simplified version of a real system where more elements can be added and more analyses can be made using the i* links presented in previous sections. Actually, the three leaf hard-goals: “Interface Created”, “Transfer Executed”, and “Message Digitally Signed” are further decomposed until there remain only simple tasks. These details are omitted here and are presented in the next section on architectural design.

At the online shop side, when using their newly introduced payment service, shop managers can create their own customized web interface to embed the “Payment Interface” offered by the “Payment System”. As a consequence, “Online Shop” also has to adapt to offer a “Secure Payment” option. The details of these adaptations are omitted here.

A.2.3. Architectural Design

In Tropos, the first two phases (analysis of early requirements and of late requirements) rely on i* that provides a powerful notion to capture the intentions of actors interacting with the system-to-be and to describe what the system-to-be is intended for. In the later

phases, we have to transform the social intentions into the design of the system.

At the first design phase (architectural design), the system is divided into sub-systems possibly using a predefined organizational style (joint venture, structure-in-5, flat structure, pyramid, takeover, arm's length, vertical integration, co-operation, bidding, etc (CASTRO, J. et al., 2002). Subsystems are represented as sub-actors inside the system actor. Every node in the strategic rationale model of the late requirement analysis becomes a dependency between two of these subsystems. At the end of this phase, each subsystem is further decomposed into agents that inter-depend within the subsystem to guarantee the subsystem responsibility. In the second design phase, inter-dependencies between agents of each subsystem are detailed using the capability diagrams, plan diagrams and dynamic diagrams. In this section, we focus on the *architectural design* where quality requirements are added to each node as described in the previous sections. The *detailed design* phase with quality requirement will be the subject of Section A.2.4.

At the macro level of Tropos, the overall architectural structure design is completely quality-driven and is based on a catalogue of organizational styles (KOLP, M. et al., 2001) and (KOLP, M. and Mylopoulos, J., 2001).

i) Organizational Style Selection

In Tropos, quality requirements are considered at a global scope and they are used to decide which organisational style will be used for the system. Predefined styles are considered in accordance to a classification of quality requirements such as those presented in Appendix A.

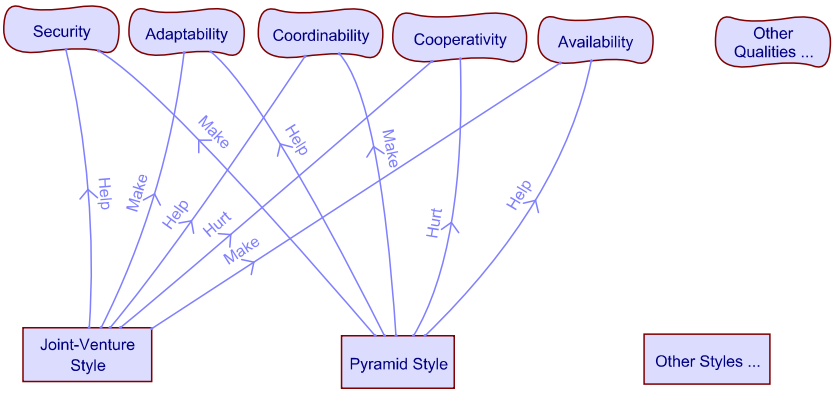


Figure A-13: Correlation Catalogue of Organizational Styles

Because quality requirements in Tropos are considered to be soft-goals, contribution links are used to evaluate the adequacy of predefined organizational styles to the required quality requirements for the future system. Contribution levels are gathered in a table called correlation catalogue (CASTRO, J. et al., 2002) illustrated in Figure A-13.

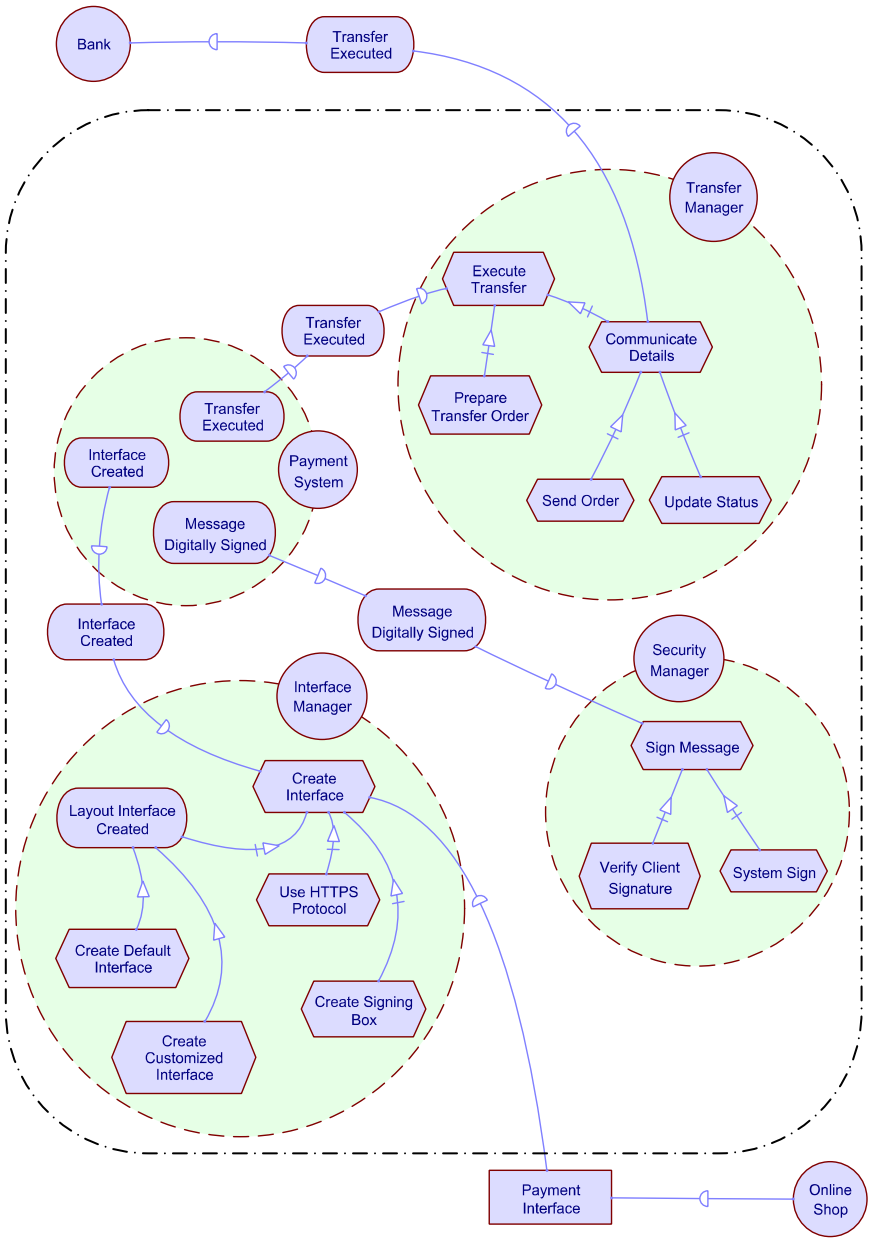


Figure A-14: Actor structure of Payment System

Here we consider only two styles: joint-venture and pyramid, together with five quality requirements: security, adaptability, coordinability, cooperability, and availability.

Going back to the payment system example, where the quality requirements of security and availability (for promptness) are most important, predefined labels on contribution links suggest that the pyramid style (KOLP, M. et al., 2001) is most suitable among those considered.

Applying the hierarchical structure of the pyramid style to our simple example, we group the elements of the "Payment System" into four sub-actors: "Payment System", "Interface Manager", "Security Manager", and "Transfer Manager" as shown in Figure A-14. Details about the top level sub-actor ("Payment System" actor) are the same as in Figure A-12 and the decompositions of the leaf hard-goals are assigned to the newly created actors: "Interface Manager", "Security Manager", and "Transfer Manager".

ii) Sub-actors to agents

After having identified all sub-actors of the future system, the last step of the phase of architectural design is to create agents inside each sub-actor. Note that, at this stage, only tasks and resources are allowed to be leaf nodes. If there is any leaf node that is other than task and resource, the decomposition is incomplete and it should be continued.

When all the leaf nodes are task or resource, one can begin to create agents. One of the easiest ways to carry this out is to transform each task or resource into a dependency between two Agents. The number of agents and how to organize agents are left to the developers to decide in accordance to the functional and communicative aspects.

In Figure A-15, sub-actor "Transfer Manager" is detailed. Five agents are added to handle the tasks of "Transfer Manager". However, in other circumstances, one can also decide to create a single agent for the whole sub-actor if it is thought that all tasks are related and should not be split.

There exist, in the literature, some catalogues of social patterns that can help developers organize agents inside actors and sub actors. One such catalogue is proposed in (DO, T., 2005) from which the wrapper pattern can be applied to our transfer manager. Three agents that are involved in this pattern are "Exchange Handler", "Transfer Wrapper", and "Bank". "Transfer Wrapper" plays the central role as a message translator between the "Payment System" and the allied "Banks".

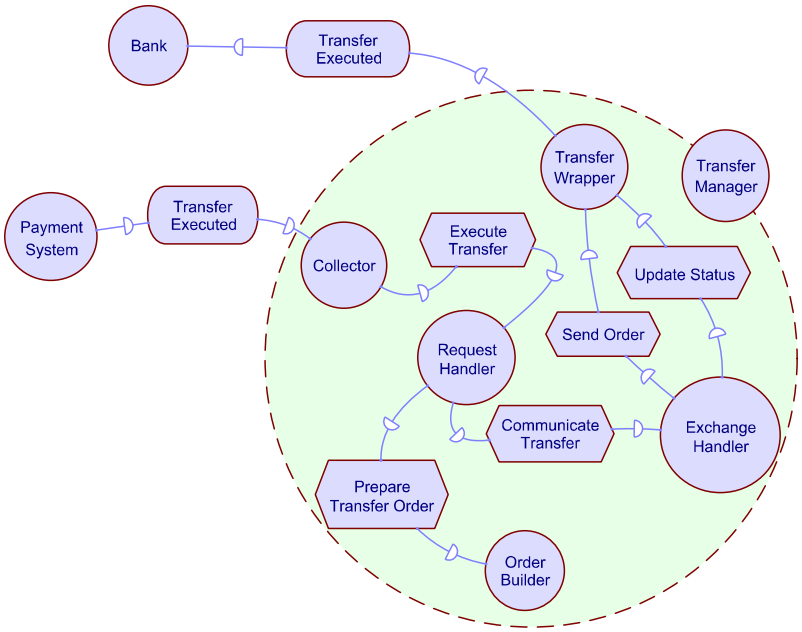


Figure A-15: Actor structure of Transfer Manager

The use of social pattern implies not only the way to organize agents but also agents' internal and interactions among agents that are involved in the chosen pattern. These details are elaborated in the last design step: Detailed Design.

A.2.4. Detailed Design

While architectural design can be considered as a macro-level design, detailed design goes into the micro-level of each agent previously identified. Only at this stage, the implementation platform is taken into account in the choice of design parameters.

To support this phase, Tropos chooses to adopt existing extensions of the Unified Modelling Language (UML) (BOOCH, G. et al., 1999) to agents, such as Agent UML (BAUER, B. et al., 2001) (ODELL, J. et al., 2000) proposed by the Foundation for Intelligent Physical Agents (FIPA) and the OMG Agent Work group.

UML diagrams that are extensively reused and extended in the context of agents are class diagrams, sequence diagrams, and activity diagrams. We will not go into details of how to use these diagrams in Tropos. Interested readers can refer to, e.g., (CASTRO, J. et al., 2002) for more information. Chapter 8 suggests a way to adapt this phase to deal with quality requirements.

Appendix B CLASSIFICATIONS OF QUALITY REQUIREMENT

"Quality words" typically take similar forms:

- -ilities: modifiability, interoperability, reliability, portability, maintainability, scalability, (re-)configurability, customizability, adaptability, variability, volatility, traceability, understandability, usability…
- -ities: security, simplicity, clarity, maturity, ubiquity, integrity, modularity, nomadicity…
- -ness: user-friendliness, structuredness robustness, timeliness, responsiveness, correctness, completeness, conciseness, cohesiveness…
- …and other: performance, efficiency, accuracy, precision, cost, development time, low coupling, fault tolerance…

The large diversity of non-functional and quality requirements has made it appealing to classify them in taxonomies. In spite of the difficulty, many classifications have been proposed by researchers and practitioners. They typically take the form of catalogues of characteristics in general defined as extensible. Still, there is no agreed-upon list of quality requirements Ambiguities, redundancies, and inadequacies have proved difficult to avoid in classifications.

A modern classification of non-functional requirements from a popular textbook (SOMMERVILLE, I., 2007) is structured as in Figure B-1 where non-functional requirements are classified into three main classes:

- **Organisational requirements**: requirements which are consequences of policies and procedures in the customer's and developer's organization. Examples are process standards; implementation requirements such as: design method or programming language; and delivery requirements such as: delivery time of product and its documentation.
- **Product requirements**: requirements which specify that the delivered product must behave in a particular way. For examples: efficiency requirements such as how fast the system must execute a particular job or how much memory it requires; reliability requirements; portability requirements; and usability.
- **External requirements**: requirements which arise from the outside of the system and its development process. Examples are: interoperability requirements define how the system will interact with others systems on other environments; ethical requirements; and legislative requirements.

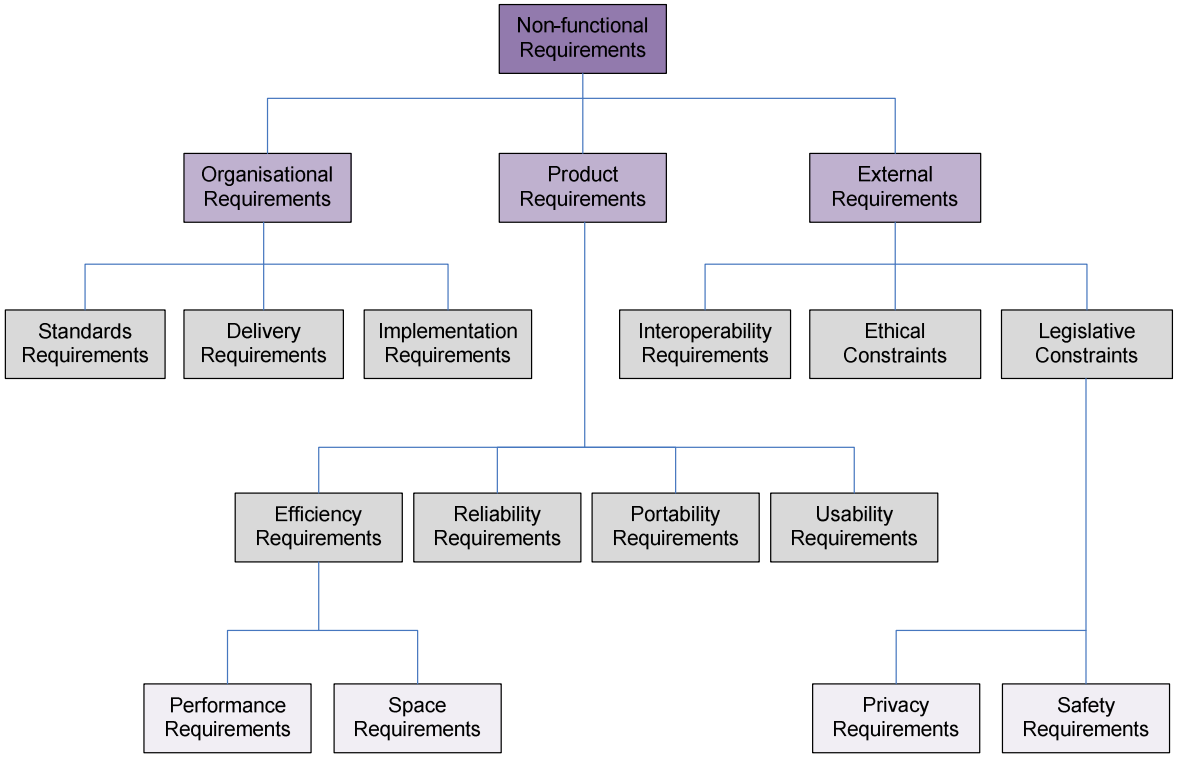


Figure B-1: Classification of NFRs of Sommerville

The classification suggested by (ROMAN, G.C., 1985) classifies the non-functional requirements into the following categories:

- **Interface requirements**: describe how the system is to interface with its environment, users and other systems. For examples, user interfaces and their qualities (e.g., user-friendliness)
- **Performance requirements**: describe performance constraints involving
  - time/space bounds, such as workloads, response time, throughput and available storage space. E.g., "system must handle 100 transactions/second"
  - reliability involving the availability of components and integrity of information maintained and supplied to the system. E.g., "system must have less than 1hr downtime/3 months"
  - security, such as permissible information flows
  - survivability, such as system endurance under file, natural catastrophes.
- **Operating requirements**: include physical constraints (size, weight), personnel availability, skill level considerations, system accessibility for maintenance, etc.
- **Lifecycle requirements**: can be classified under two subcategories:
  - quality of the design: measured in terms such as maintainability, enhanceability, portability.
  - limits on development, such as development time limitations, resource availability, methodological standards, etc.
- **Economic requirements:** immediate and/or long-term costs
- **Political requirements**

The approach in (MCCALL, J.A. et al., 1977) showns in Figure B-2 splits qualities into more detailed characteristics that are supposed to be easy to measure or to assess.

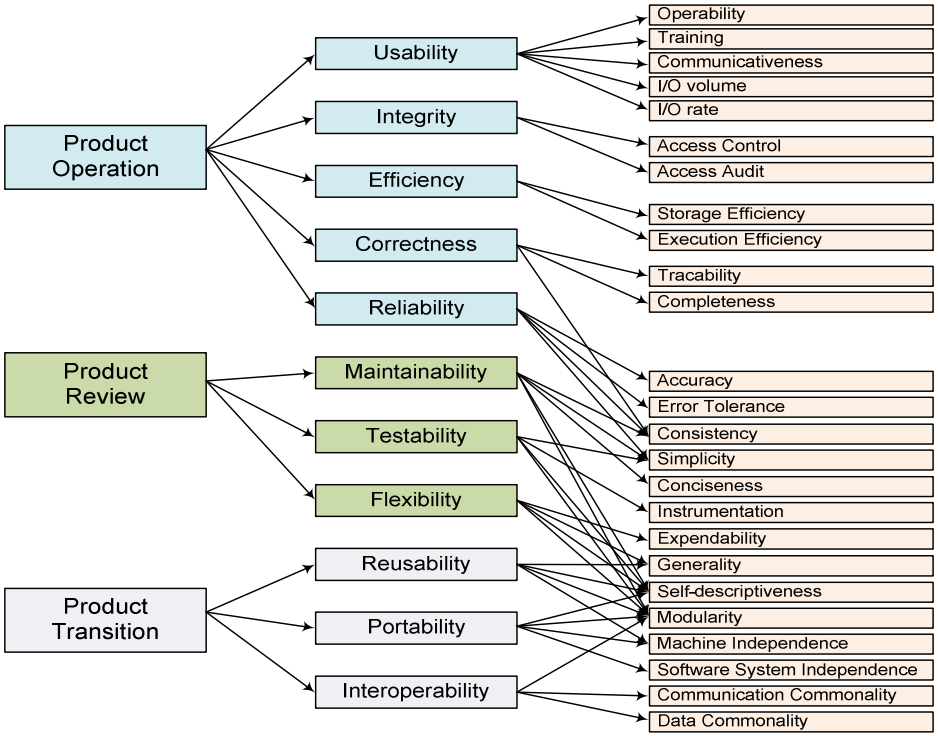


Figure B-2: Classification of software quality by McCall

FURPS (Functionality, Usability, Reliability, Performance and Supportability) approach (GRADY, R.B., 1992) classifies all the requirements including functional and non-functional.

Functionality	Feature set capabilities, security, generality	
Usability	Human factors aesthetics, consistency, documentation	
Reliability	Frequency/severity of failure, recoverability, predictability, accuracy, MTBF	
Performance	Speed efficiency, resource usage, throughput, response time	
Supportability	Testability	Extensibility
	Adaptability	Maintainability
	Compatibility	Configurability
	Serviceability	Installability

Figure B-3: FURPS approach for NFR classification

The revised version FURPS+ adds the following requirements:

- Design requirements: constraints on the system design, e.g. tree-layer client/server structure must be used, etc.
- Implementation requirements: constraints on coding and construction, e.g. programming language, operation platforms, etc.
- Interface requirements: constraints on the interactions with external factors, e.g. web-interface must be built, etc.
- Physical requirements: constraints on the hardware that house the software system, e.g. three servers are needed to assure all the transactions, etc.

The ISO/IEC 9126 (ISO/IEC, 9126-1, 2001) has been one of the most influential standards for the evaluation of software. The standard covers both quality models and metrics. Its main idea is the definition of a quality model and it use as a framework for software

evaluation. The standard is divided into four parts, addressing, respectively, quality model, external metrics, internal metrics, and quality in use. Internal attributes are measured during the development process while external ones concern the testing process. The user view of quality is measured through quality-in-use attributes.

Part one, referred to as ISO/IEC 9126-1, addresses the definition of quality models. It is an extension of previous work by (MCCALL, J.A. et al., 1977), (GRADY, R.B., 1992), and others (see also (KAN, S.H., 2002)). An ISO/IEC-1 quality model is defined with general characteristics of software, further refined into subcharacteristics, themselves decomposed into attributes. At the bottom of the hierarchy, the measurable software attributes have values that can be computed with some metrics. Quality requirements may be defined as constraints on the quality model.

The quality model identifies 6 main quality characteristics, each refined into subcharacteristics as follows:

- Functionality, refined as suitability, accuracy, interoperability, security, and functional compliance
- Reliability, refined as maturity, fault tolerance, recoverability, and reliability compliance
- Usability, refined as understandability, learnability, operability, attractiveness, and usability compliance
- Efficiency, refined as time behaviour, resource utilisation, and efficiency compliance
- Maintainability, refined as analysability, changeability, stability, testability, and maintainability compliance
- Portability, refined as adaptability, installability, co-existence, replaceability, and portability compliance

Definitions of characteristics and subcharacteristics can be found in (ISO/IEC, 9126-1, 2001) of course and in many other places, for example on SQA (Software Quality Assurance) web site (http://www.sqa.net/iso9126.html)

Appendix C SOFTWARE DEVELOPMENT PROCESS

This appendix sets the stage concerning central topics of software development in general and then of agent orientation.

Section C.1 introduces basic concepts and then surveys practical software lifecycles with pointers to the relevant literature.

Section C.2 focuses on the agent-oriented paradigm, and on its relevance and importance to software development. Some agent-based development methodologies are given a brief description with pointers to the literature.

C.1. Software System Development

Ideas in software engineering are usually packaged into methods (or "methodologies") that aim at serving as comprehensive and coherent guidelines throughout the software development life cycle. A methodology, as seen, e.g., by (RUMBAUGH, J., 1995), must have the following set of components:

- a set of modelling concepts to help understand the problem and its solution
- a set of notions and views to model the requirements and the structure of the future software system
- a development process that defines the principal phases from engineering user requirements to deploying and maintaining the final software system
- a collection of guidelines and rules about how to best conduct the development process

The choice of methodology depends on many factors including the size of the software system and the environment where the future system will operate. Most methodologies try to adapt to new technologies and best take into account to the actual application requirements. This chapter briefly reviews some important issues of software engineering.

C.1.1. Evolution of modelling concepts and notions

The following are some important modelling concepts and notions that have contributed to the development of software engineering:

- Function: systems are modelled as a set of interacting functions. Typical engineering notions using the function concepts include: top-down/bottom-up analysis, stepwise refinement, data flows. A system viewed through this analysis can be broken down into functions, possibly grouped into package. Calling relationships can be represented as a graph with functions at the nodes and calls between functions as directed links.
- Object: the system is built on the basis of the concepts of objects that are defined in terms of *methods* and *attributes*. Object-oriented programming has become the dominating style in recent years using the following fundamental building blocks: class, object, instance, method, attribute, message passing, inheritance, abstraction, encapsulation, polymorphism, etc. Unified Modelling Language (UML) (BOOCH, G. et al., 1999) is the well-known modelling language for object-oriented software.
- Service: is a notion that is usually used to design distributed software components. They are characterized by the way in which different components are loosely coupled. Procedural and method calls (as in functions and objects) are replaced by formatted messages exchanged between components through the network. While in functions and objects, links are statically bound at the implementation time thanks to their functional dependency, in services, links are established dynamically at the runtime by matching the description of available services.
- Agent: is a powerful extension of the object notion. Unlike objects, agents are defined in terms of behaviours. While objects are still a passive entity, agents are given a certain degree of *autonomy* in accomplishing tasks on behalf of their users (FRANKLIN, S. and Graesser, A., 1997). While objects are only provoked by calls to their methods, agents run continuously to perceive changes in the environment and react to those changes on the basis of their goals and their

capabilities. Agents inherit the dynamical binding mechanism and some message exchange protocols from services. Compared to services, agents are more autonomous, intelligent and mobile.

C.1.2. Software development lifecycle

The development process aims to break down software development into several phases. Many processes have been proposed with various phases. Five basic development activities of a software engineering process (SOMMERVILLE, I., 2007) have been recurring:

- Requirements engineering phase: aims to make explicit what is expected from the system-to-be in terms of functionalities and constraints. Requirements are collected through documents and interview transcripts with the customers. They are then analyzed and validated to produce a software specification document. This document guides later phases of the development process. Any error in this document will affect the whole process and may lead to substantial problems in software design and implementation. This phase can be divided into several sub-stages such as: feasibility study, requirement capture and analysis, requirement specification, and requirement validation.
- Design phase: converts the software specification into the detailed software structures (architectural design, interface design, component design, data structure design, algorithm design) that can be directly implemented and deployed on computer systems. Some design tools allow developers to generate a part of program codes from the detailed design. Usually, placeholders are added into codes to let the programmers enter specific codes while keeping the structure of codes as close as possible to the design.
- Implementation phase: translates the results from the design phase into program codes then to instructions readable and executable on a real machine or a virtual machine. A complete version of the system is expected at the end of this phase and is ready to be tested in the next phase.
- Validation phase: compares the implemented system with the specification to see if the built software can perform and give the correct results as specified in the specification document. The system can be brought back to the formers phases to fix errors eventually found during tests carried out in this phases. At the end of this phase, we should have the built system ready to be delivered and deployed.

- Evolution: when new requirements arise during the system operation, they will be recorded. These will be used for the next version of the system.

The list above gives the most basic phases of a software development process. However, the way and the order that these phases are carried out are different from one development process to another. We refer to a specific way to carry out the development process as a software development lifecycle model. This lifecycle model will guide the development process and offer aids in planning, monitoring, and controlling project development by defining checkpoints, control flow, etc.

The following paragraphs present an overview of several models that have been well studied and widely used. We try to summarize the pros and the cons of each model.

i) Stagewise model:

One of the first lifecycle models for software development is the *Stagewise* model. The software is developed by consecutive phases and one phase at a time. The next phase begins when the current phase have been completed. Feedbacks to the previous phases in the development process are not allowed.

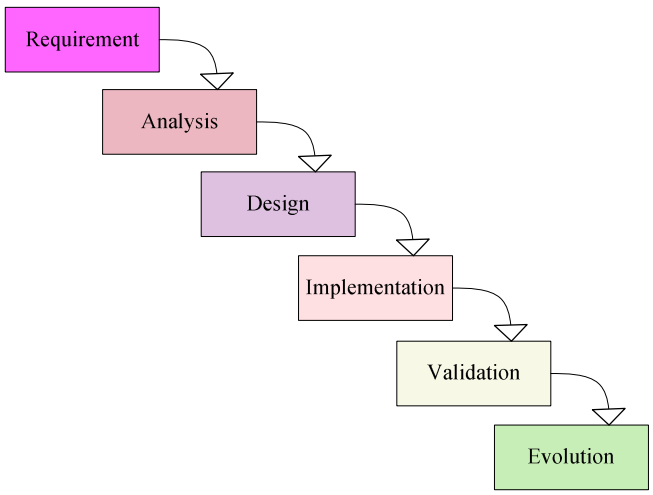


Figure C-1: The Waterfall Model

This model is very simple and well structured but has a lot of disadvantages. Because it is a step by step process with no feedback, every phase must be carried out carefully, especially first phases. In practice, it is really difficult to make the detailed specification

without any iterations and feedback. If everything is correctly addressed, the validation phase could be relatively simple to carry out.

Normally, when there is less no backward links, the process can be accelerated. However, this is not true here. The absence of feedback link makes developers to pay more time in each phase to assure that it is correctly carried out. And this could delay the delivery of the final system.

An improved version of the Stagewise model is the classical Waterfall model that emphasizes feedbacks between two consecutive phases (GEORGIADOU, E., 2003). In this model, the cycle of one phase can be repeated many times during the life of the software system.

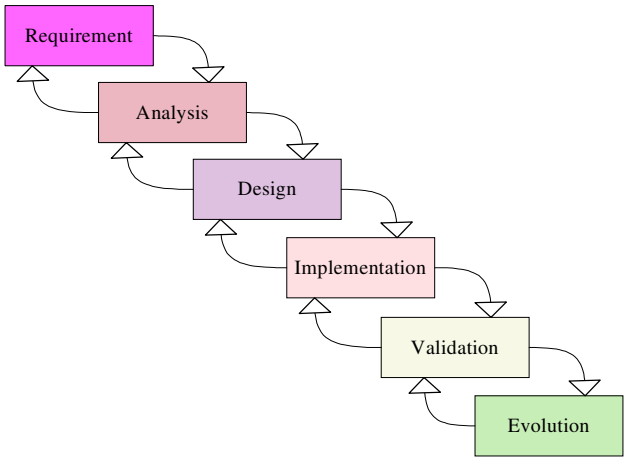


Figure C-2: The Stagewise Waterfall Model

ii) V-model:

V-model can be considered as an extension of the classical Waterfall model. The falling down process is bent up to form the shape of V letter. The V in the model's name is also known as a abbreviation of Verification and Validation (SPILLNER, A., 2001). Indeed, each the stage of the software development lifecycle is connected with its associated phase of testing. In this model, there are 2 branches: phases before implementation phase (analysis, architecture design, and detailed design) are showed on the left and all the corresponding testing phases are placed on the right.

The model is simple and easy to understand. It emphasizes that test scenarios should be designed as early as possible after each phase of the development process. In the Waterfall model, there is no guideline for test scenario design and both test scenario design and

the actual tests are carried out after the implementation phase. However, this makes tests less efficient.

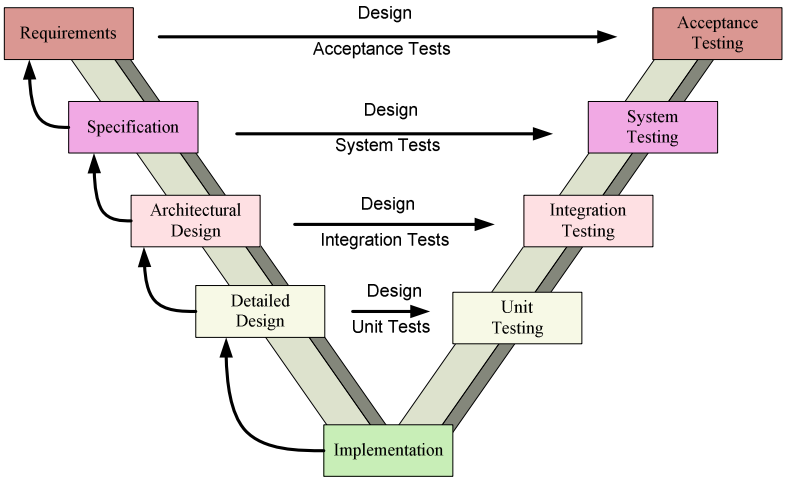


Figure C-3: The V-Model

It turns out that acceptance tests should be designed as early as the system requirements are collected. Likewise, system tests, integration tests and unit tests should be designed respectively when the specification, the architectural design and the detailed design are done. This ensures that test scenarios are correctly and adequately generated before the actual tests are taken place.

iii) Prototyping Model:

With this model, the development process starts with the requirements engineering phase. Then a quick design is carried out, followed by an implementation of a simple prototype. The first prototype is submitted to customers for feedbacks that will be used to improve the prototype (NORIN, L.A. and Stockel, F., 1998). The first feedbacks are used to design and to build a second prototype, and so on. Successive prototypes may not be related one to another. Instead, they can show different parts and/or aspects that are specified. The prototype design-building-evaluation cycle is repeated as long as requirements are correctly specified and sufficiently collected for the development of the final product.

Prototypes usually focus on areas that are visible to users such as user interface and basic functionalities. For customers, being showed a prototype almost instantaneously is very useful. This is an advantage of this model. Besides, prototypes may help customers to

reveal hidden requirements that were not included missing in initial requirements.

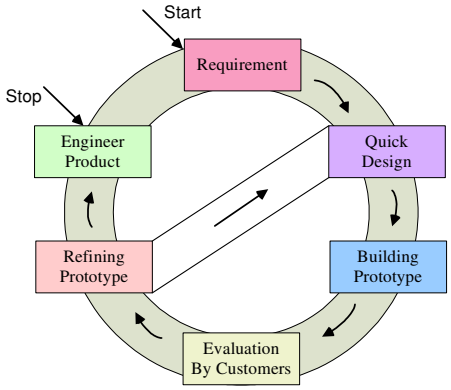


Figure C-4: The Prototype Model

iv) Spiral model:

The spiral-model (BOEHM, B., 1986) is not the first iterative development model but it was the first that showed the interest of iterative processes. The prototyping model above is also an iterative model but it does not carry out all the development phases. In the spiral model, each evolution also produces a small system. However, the difference with the previous model is that those intermediate systems in the spiral model are related and progressively refined until the final system, unlike the prototyping model where prototypes are kept very simple and mostly for clarifying the requirements.

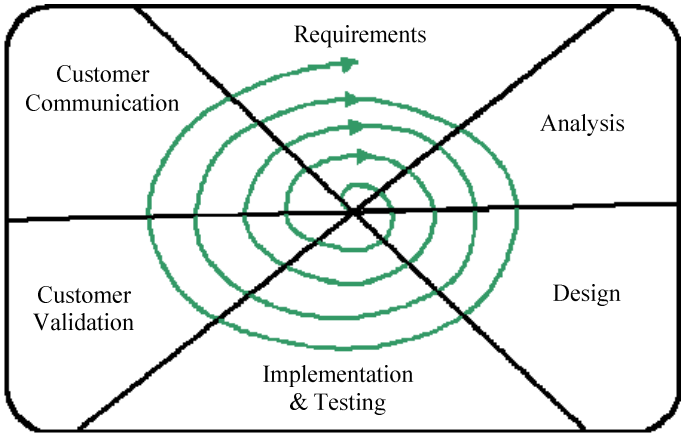


Figure C-5: The Spiral Model

This model is suitable for large software project and promotes the quality assurances of the final product through the refinement process of intermediate systems. Others advantages are: early elimination of errors, high reusability and permanent maintenance, etc.

v) Rapid Application Development (RAD):

Rapid Application Development (MARTIN, J., 1991) is an iterative scheme for small project. It inherits the spiral manner of other iterative models and emphasizes the use of Computer-Aided Software Engineering (CASE) tool to speed up the software development process.

The spiral-like iterative model has been widely adopted by the community of Agile Software Development (COCKBURN, A., 2007). All methodologies, which follow the agile movement, use the spiral model as an important building part. They differ from each other in how to organize their working environment, to interact with customers, to respond to changes, etc.

C.2. Agent-Oriented Paradigm

Agent-Oriented Programming (AOP) is an extension of the Object Oriented Programming (OOP). Objects are replaced by agents living in virtual societies called multi-agent systems. This section focuses on the development process for multi-agent systems in which agents are the central notions.

We first review the definition of agents and multi-agent systems and then some development processes are discussed.

C.2.1. Agents and Multi-Agents System (MAS)

From a software engineering point of view, agents are software units or entities that are located in an environment and can perceive changes in the environment. They are offered certain autonomy and interact with other entities such as other software agents, humans, hardware, etc. in order to achieve their assigned goal (WOOLDRIDGE, M.J. and Jennings, N.R., 1995).

Definition 3: (FRANKLIN, S. and Graesser, A., 1997)

> ***Autonomous* agent** is a system situated within and a part of an environment that sense that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it sense in the future.

The works in (FRANKLIN, S. and Graesser, A., 1997) listed and compared several appealing definitions of agents and proposed the definition shown in Definition 3.

Broadly speaking, an agent can be a person, an organization, a system, a machine, a software entity,... characterized by the faculty to interact with their environment. An agent exhibits the following properties (ODELL, J., 2000):

- **Autonomous –** is capable acting without direct external intervention. It has some degree of control over its internal state and actions based on its own experiences.
- **Interactive** – communicates with the environment and other agents.
- **Adaptive** – capable of responding to other agents and/or its environment to some degree. More advanced forms of adaptation permit an agent to modify its behaviour based on its experience.
- **Sociable** – interaction that is marked by friendliness or pleasant social relations, that is, where the agent is affable, companionable, or friendly.
- **Mobile** – able to transport itself from one environment to another.
- **Proxy** – may act on behalf of someone or something, that is, acting in the interest of, as a representative of, or for the benefit of some entity.
- **Proactive** – goal-oriented, purposeful. It does not simply react to the environment.
- **Intelligent** – state is formalized by knowledge (i.e., beliefs, goals, plans and assumptions) and interacts with other agents using symbolic language.
- **Rational** – able to choose an action based on internal goals and the knowledge that a particular action will bring it closer to its goals.
- **Unpredictable** – able to act in ways that are not fully predictable, even if all the initial conditions are known. It is capable of nondeterministic behaviour.
- **Temporally continuous** – is a continuously running process.
- **Character** – believable personality and emotional state.
- **Transparent and accountable** – must be transparent when required, yet must provide a log of its activities upon demand.
- **Coordinative** – able to perform some activity in a shared environment with other agents. Activities are often

coordinated via plans, workflows, or some other process management mechanism.

- **Cooperative** – able to coordinate with other agents to achieve a common purpose; non-antagonistic agents that succeed or fail together. (Collaboration is another term used synonymously with cooperation.)
- **Competitive** – able to coordinate with other agents except that the success of one agent implies the failure of others (the opposite of cooperative).
- **Rugged** – able to deal with errors and incomplete data robustly.
- **Trustworthy** - adheres to Laws of Robotics and is truthful.

We can reorganize the above characteristics into three categories: individual characteristics (autonomous, character, transparent and accountable, mobile, temporally continuous, trustworthy and rugged), rational characteristics (intelligent and rational) and social characteristics (interactive, adaptive, sociable, proxy, proactive, coordinative, unpredictable, cooperative and competitive).

Multi-Agent Systems are based on the concept of *agent*. The global behaviour of multi-agent system is derived from the interaction among the constituent agents. They cooperate, coordinate and negotiate with one another. A multi-agent software system is generally conceived as a society of autonomous, collaborative, and goal-driven software components.

Additional information about the notion of agents can be found in the following references (NWANA, H. S., 1996)(SHOHAM, Y., 1993).

C.2.2. Agent-Oriented Methodology

Many agent-oriented methodologies have been proposed in the recent years to provide guidelines for the analysis and for the design of a multi-agent system. GAIA, AUML, MESSAGE, TROPOS, etc. place the software developers in front of different choices of modelling languages and methodologies. Besides the agent notion, that varies substantially from a methodology to another, this represents a wide range of different development lifecycle, modelling logics and supporting tools. For the developers, this also represents the difficulty in choosing the most suitable one since they usually address different agent properties.

At first, agent-oriented technology can be viewed as a natural extension of the object-oriented technology with an extra ability for agents to run concurrently. More advanced agents enjoy additional abilities like mobility, intelligent reasoning, self-evolution, etc., which

make agent-oriented technology a powerful alternative to object-oriented technology (SHOHAM, Y., 1993).

Below is a brief summary, taken from (HENDERSON-SELLERS, B. and Giorgini, P., 2005), of prominent methodologies for developing multi-agent systems. Each of them helps software developers in a different ways. None of them has become a standard for agent-oriented software development methodologies.

- **Tropos methodology** (CASTRO, J. et al., 2002): provides guidance for the four major development phases of application development. One of its primary contributions is placing an emphasis on modelling goals and their relationship with the system's actors, task and resources.
- **MAS-CommonKADS methodology** (IGLESIAS, C.A. et al., 1997): is based on both CommonKADS(SCHREIBER, G. et al., 1994) and object-oriented based methodologies. This enables the developer to build agent-based systems while leveraging the experience of pre-agent methodologies and employing familiar techniques and diagrams.
- **PASSI methodology** (COSSENTINO, M. and Potts, C., 2002): brings a particularly rich development lifecycle that spans initial requirements though deployment and, in addition, emphasizes the social model of agent-based systems.
- **Prometheus methodology** (PADGHAM, L. and Winikoff, M., 2003): provides an especially rich goal-driven approach for its BDI-like agents. Its methodology is used today to develop systems on commercial BSI-based agent platforms, such as JACK or Agentis.
- **Gaia methodology** (WOOLDRIDGE, M.J. et al., 2000): is one of the earliest agent methodologies and now reflects this experience in version two of its approach. Using the analogy of human-based organizations, Gaia provides an approach that both a developer and a non-technical domain expert can understand, facilitating their interaction.
- **ADELFE methodology** (BERNON, C. et al., 2003): is a specialized methodology that emphasizes cooperative agents that self-organize and possibly result in emergent systems. More specifically, it addresses designing complex adaptive systems and anticipating emergence within its software agents.
- **MESSAGE methodology** (CAIRE, G. et al., 2002): extends existing object-oriented methodologies for agent-oriented applications. Chartered to address telecommunications applications, its resulting RUP-based approach also supports more general applications.

- **INGENIAS methodology** (PAVON, J. and Gomez-Sanz, J., 2003): supports a notation based on five metamodels that define the different views and concepts of a multi-agent system. Using metamodels provides flexibility for evolving the methodology and adopting changes to its notation.
- **RAP methodology**: is concerned with distributed information systems (such as enterprise resource planning and supply-chain management systems) and places less emphasis on AI-based systems. The philosophy of the Model Driven Architecture is adopted with the goal that executable software agents can be generated using RAP artefacts.
- **MaSE methodology** (WOOD, M.F. and DeLoach, S.A., 2001): is a comprehensive methodology that has been used "to develop systems ranging from heterogeneous database integration applications to biologically based, computer-virus immune systems and cooperative robotics systems. Its hybrid approach can be applied to multi-agent systems that involve implementations such as distributed human and machine planning.

A comparison between some of the above agent-oriented methodologies is proposed in (SUDEIKAT, J. et al., 2005). It relies on different criteria classified in four groups: (1) concepts (internal architecture, social architecture, communication, autonomy, pro-activity and distribution), (2) notation (usability, expressiveness, refinement, dependency of models, traceability, clear definition and modularity), (3) process (coverage of workflow, management, complexity and properties of process), and (4) pragmatics (tool support, connectivity, documentation and usage in projects).