Proceedings Second International Conference
On Enterprise Information Systems,
Stafford, UK, July 4-7, 2000

DEVELOPING AGENT-ORIENTED INFORMATION SYSTEMS FOR THE ENTERPRISE

Jaelson Castro[*] **Manuel Kolp**[‡] **John Mylopoulos**[‡]

[*] *Universidade Federal de Pernambuco, Recife PE, Brazil; email: jbc@di.ufpe.br*
[‡] *University of Toronto, Department of Computer Science, Toronto, Canada; email; {mkolp,jm@cs.toronto.edu}*

Abstract. Enterprise information systems have traditionally suffered from an impedance mismatch. Their operational environment is best understood in terms of agents, responsibilities, objectives, tasks and resources, while the information system itself is conceived as a collection of (software) modules, data structures and interfaces. This paper explores a software development methodology which eliminates this mismatch by treating the concepts of agent and goal as primitives during requirements analysis and design. In particular, our proposal adopts Eric Yu's *i** framework [1], a modeling framework for early requirements, and uses it to model not just early, but also late requirements, as well as architectural and detailed design. The proposed framework, named Tropos, seems to complement nicely current research on agent-oriented programming platforms.

Keywords: software development, software requirements analysis and design, agent-oriented systems, software architectures.

1. Introduction

Enterprise information systems have traditionally suffered from an impedance mismatch. Their operational environment is best understood in terms of agents, responsibilities, objectives, tasks and resources, while the information system itself is conceived as a collection of (software) modules, data structures and interfaces. This mismatch is one of the factors for the poor quality of enterprise information systems, also the frequent failure of enterprise information system development projects.

The main objective of this paper is to explore a software development methodology which is founded on the concepts of agent and goal. The methodology makes it possible to use the same concepts to describe the organizational environment within which a software system will eventually operate, as well as the system itself.

The proposed methodology supercedes traditional software development techniques, such as structured [2, 3] and object-oriented ones [4, 5] in the sense that it is tailored to software systems that will operate within an organizational context.

The software development framework is named Tropos (derived from the Greek "tropé", which means "easily changeable", also "easily adaptable"), and is founded on concepts used during early requirements analysis. To this end, we adopt the concepts offered by *i** [1], a modeling framework offering concepts such as *actor*, *agent*, *position* and *role*, as well as social

dependencies among actors, including *goal, softgoal, task* and *resource* ones.
The proposed methodology spans four phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an existing organizational setting; the output of this phase is an organizational model which includes relevant actors and their respective goals;
- Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities;
- Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows;
- Detailed design, where each architectural component is defined in further detail in terms of inputs, outputs, control, and other relevant information.

Section 2 describes a use case for an e-commerce web application. Section 3 introduces the primitive concepts offered by *i** and illustrates their use with an example. Sections 4, 5, and 6 sketch how the technique might work for late requirements, architectural design and detailed design respectively. Finally, section 7 summarizes the contributions of the paper, offers an initial self assessment of the proposed development technique, and outlines directions for further research.

2. A Use Case Specification Example

The case description has been adopted from [6]. *Flower Shop* is a store selling and shipping different varieties of flowers. *Flower Shop* customers (on-site or remote) can use a catalogue describing available products to make their order. The catalogue is updated regularly and mailed out to the shop's customer base. *Flower Shop* is supplied with quality flowers by *Flower Supplier*. To increase market share, *Flower Shop* has decided to open up a retail sales front on the internet. With the new setup, a customer can order *Flower Shop* products in person, by phone, or through the internet. The internet system has been named *FlowerAlive!*

2.1 Project Objective

The basic objective for the new system is to allow an on-line customer to examine the different kinds of flowers in the *FlowerAlive!* internet catalogue, also to place orders.

2.2 Description

The system is supposed to be available to any potential customer with internet access and a web browser. There are no registration restrictions, or identification procedures to navigate the catalogue. Even if she is not purchasing anything, an anonymous visitor is considered an on-line customer of *FlowerAlive!*.

Potential customers can search the on-line store by either browsing the catalogue or querying the flower database. The catalogue groups flower varieties into hierarchies so that potential customers can view and compare similar types of flowers.

An on-line search engine allows customers with particular flowers in mind to search variety names and descriptions through keywords. Other internet visitors are just expected to navigate the catalogue by browsing *FlowerAlive!* offerings.

Details about flowers include name, short description, breeder name, year introduced, cost, and sometimes pictures (when available).

When the customer decides what flower variety to buy, she can add the item to her shopping cart. At any time, the customer can decide to check out and purchase the items in

the shopping cart. Once items have been checked out, the customer can complete the transaction by giving out relevant information (name, address, credit card number, bank account, delivery address and date, personal message, etc.) by phone or by internet using standard forms or encrypted secure forms.

3. Early Requirements with *i**

During early requirements analysis, the requirements engineer is supposed to capture and analyze the intentions of stakeholders. These are modeled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non-functional requirements of the system-to-be [7]. In *i** (which stands for "distributed intentionality''), early requirements are assumed to involve social actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The *i** framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors. These models have been formalized using intentional concepts from AI, such as goal, belief, ability, and commitment (e.g., [8]). The framework has been presented in detail in [1] and has been related to different application areas, including requirements engineering [9], business process reengineering [10], and software processes [11].

A strategic dependency model is a graph, where each node represents an *actor*, and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goal. We call the depending actor the *depender* and the actor who is depended upon the *dependee*. The object around which the dependency centers is called the *dependum*. By depending on another actor for a dependum, an actor is able to achieve goals that it is otherwise unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals. Figure 1 shows the beginning of an *i** model consisting of two relevant actors coming from the *Flower Shop* use case described in Section 2.

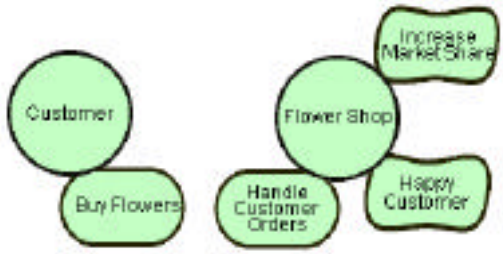

Figure 1: "*Customers* want to buy flowers, while the *Flower Shop* wants to increase market share, handle orders and keep customers happy"

The two actors are named respectively *Customer* and *Flower Shop*. The customer has one relevant goal *Buy Flowers* (represented as an oval-shaped icon), while the flower store has goals *Handle Customer Orders*, *Happy Customer*, and *Increase Market Share*. Since the last two goals are not well-defined, they are represented in terms of softgoals (shown as cloudy shapes).

Once the relevant stakeholders and their goals have been identified, a means-ends analysis determines how these goals (including softgoals) can actually be fulfilled through the contributions of other actors. Let's focus on one such goal, namely the softgoal *Increase Market Share.*

As shown in figure 2, the analysis is carried out from the perspective of the *Flower Shop* actor, who has the (soft)goal *Increase Market Share* in the first place. The analysis begins with that softgoal and postulates a task *Run Shop* (represented in terms of a hexagonal

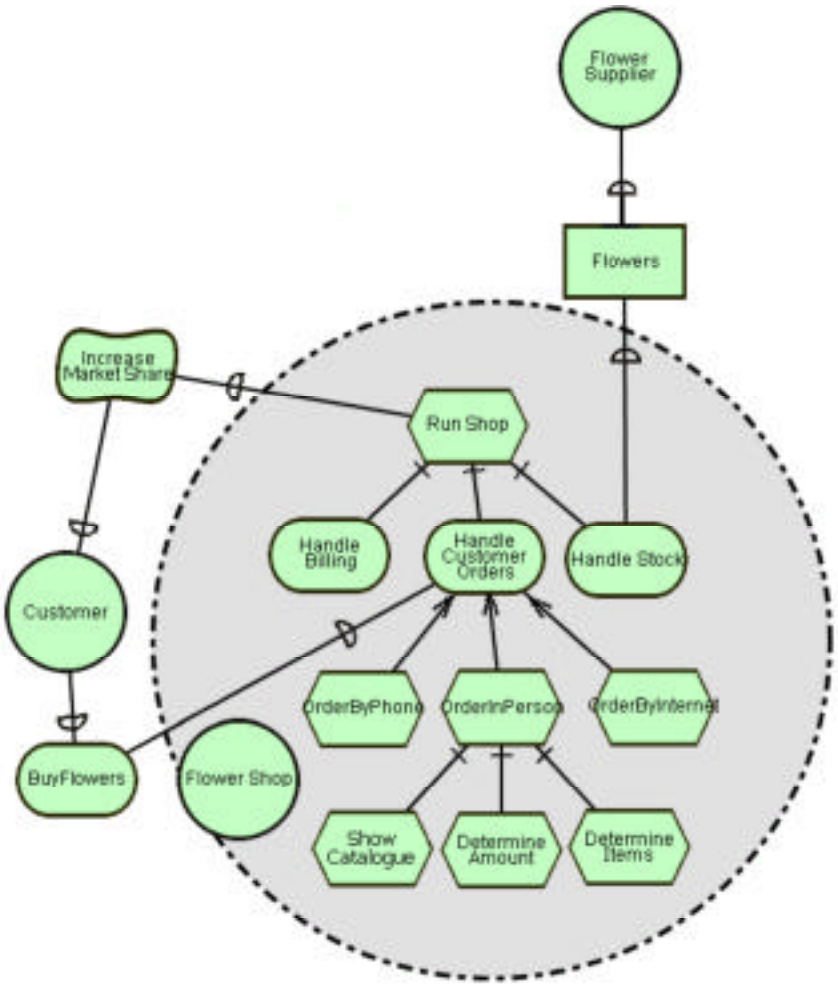

Figure 2: Means-ends analysis for the goal *Increase Market Share*

icon) through which the goal can be fulfilled. Tasks are partially ordered sequences of steps intended to accomplish some (soft)goal. Tasks, in turn, can be decomposed into goals, whose collective fulfilment can complete the task. In the figure, the root task *Run Shop* is decomposed into goals *Handle Billing*, *Handle Customer Orders* and *Handle Stock* which together accomplish the top-level task. In turn, the goal *Handle Customer Orders* might be fulfilled through tasks *OrderByPhone, OrderInPerson* or *OrderByInternet.* Decompositions continue until the means-ends analysis can identify an actor who can accomplish a goal, carry out a task, or deliver on some needed resource. An example of such a dependency in figure 2 is the resource dependency on the actor *Flower Supplier* for supplying flowers.

To complete the analysis, the task *OrderInPerson* can be further decomposed into sub-tasks *Show Catalogue, Determine Amount* and *Determine Items* which together accomplish the task of ordering flowers in person.

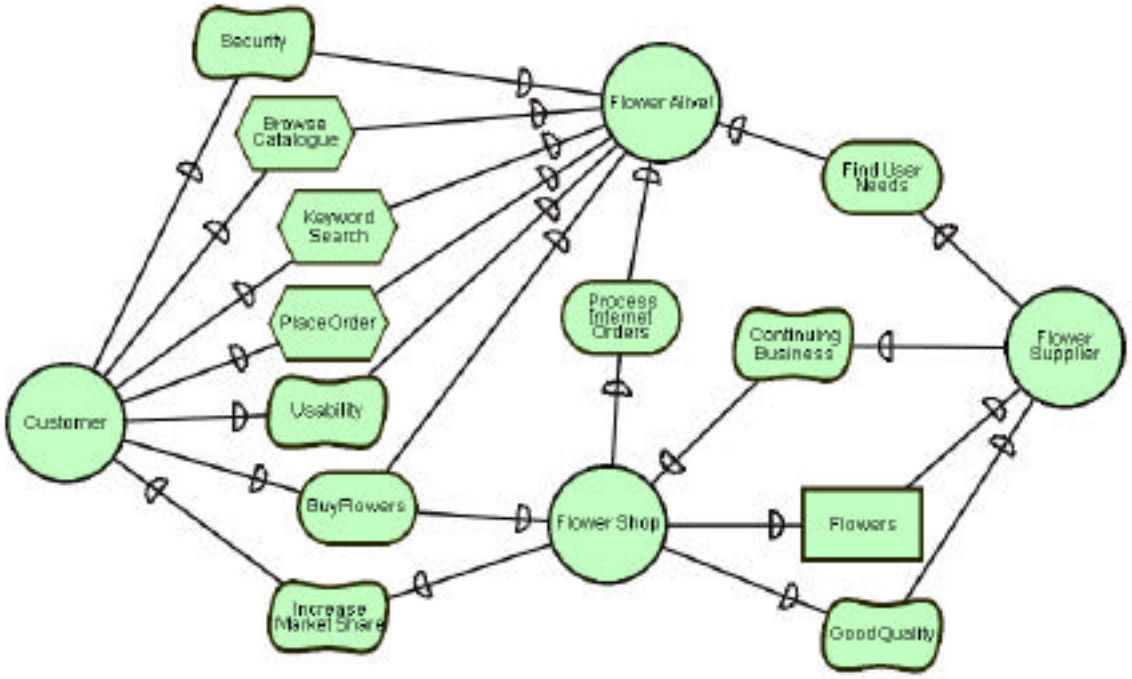

Figure 3: Strategic dependency model for a flower shop

4. Late Requirements Analysis

Late requirements analysis results in a requirements specification document which describes all functional and non-functional requirements for the system-to-be. In Tropos, the software system is represented as one or more actors which participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors which contribute to the fulfilment of stakeholder goals. For our example, the *FlowerAlive!* web system is introduced as an actor in the strategic dependency model depicted in figure 3.

According to this model, the customer depends on the flower shop to buy flowers while the flower shop depends on the customer to increase market share. The flower supplier is expected to provide the flower shop with good quality flowers because of his dependence on the latter for continued long-term business. As indicated earlier, the flower shop depends on the *FlowerAlive!* web system for processing internet orders. The customer, in turn, depends on the *FlowerAlive!* actor to order flowers through the internet, to search the flower database for keywords or simply to browse the on-line catalogue. Moreover, the web system needs to be usable and secure with respect to the customers' personal needs.

Although a strategic dependency model provides hints about why processes are structured in a certain way, it does not sufficiently support the process of suggesting, exploring, and evaluating alternative solutions. That is the role of the Strategic Rationale model. A strategic rationale model is a graph with four main types of nodes -- goal, task, resource, and softgoal -- and two main types of links -- means-ends links and process decomposition links. A strategic rationale graph describes the criteria in terms of which each actor selects among alternative dependency configurations.

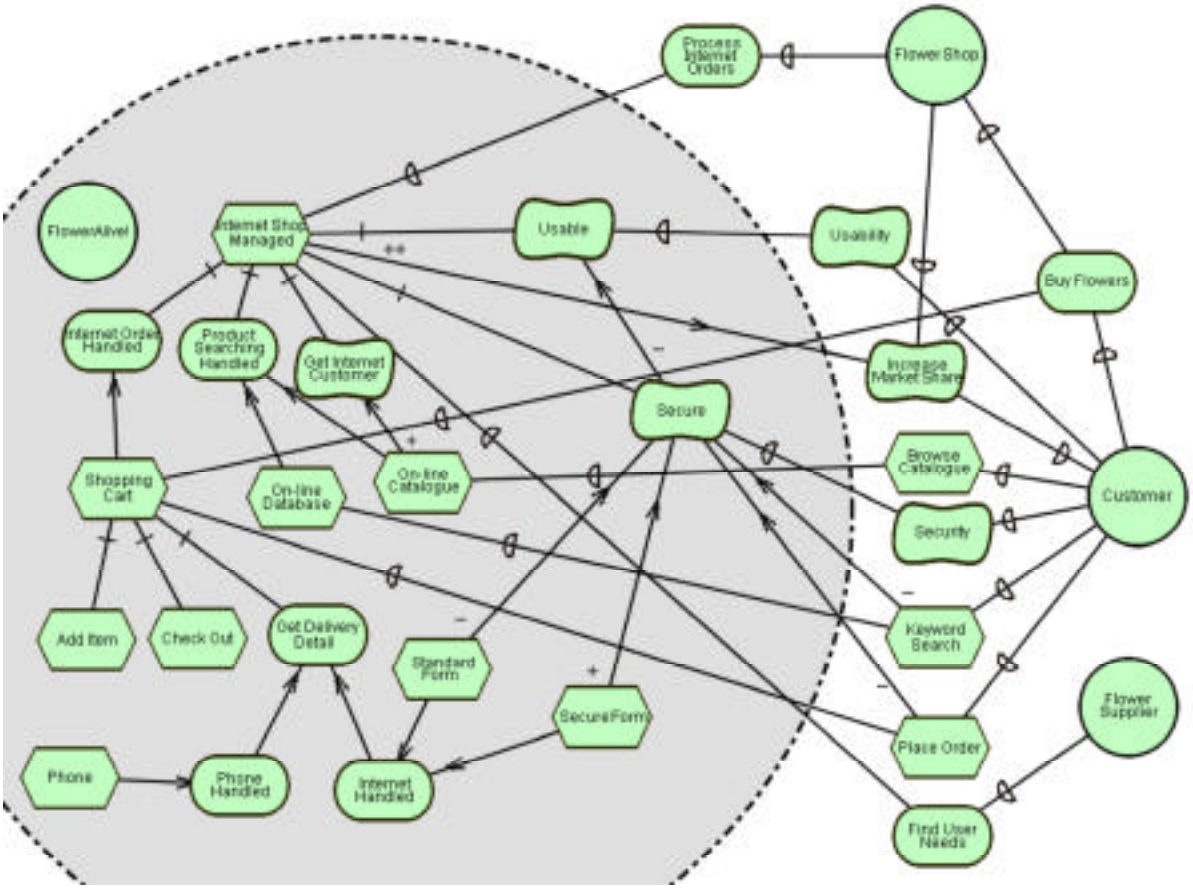

Figure 4: Strategic rational model for the *FlowerAlive!* system actor

The analysis in figure 4 postulates a root task *Internet Shop Managed* contributing positively (++) to the softgoal *Increase Market Share*, associated during early requirement analysis with the *Flower Shop* actor. Of course, as late requirements analysis proceeds, the *FlowerAlive!* system is given additional responsibilities, and ends up as the depender of several dependencies. Moreover, the system is decomposed into several sub-actors which take on some of these responsibilities. This is done using the same kind of means-ends analysis along with the kind of strategic rationale analysis illustrated in figure 2. The task *Internet Shop Managed* consists then of goals *Internet Order Handled* and *Product Searching Handled*, as well as softgoals *Get Internet Customer*, *Secure* and *Usable*. The goal *Internet Order Handled* is achieved through the task *Shopping Cart* which is, in turn, decomposed into sub-tasks *Add Item* and *Check Out*, and goal *Get Delivery Detail*. This goal can be accomplished through subgoals *Phone Handled* or *Internet Handled*. The latter goal is achieved through tasks *Phone* dealing with phone orders and also *Standard form* or *Secure form* managing internet orders. The goal *Product Searching Handled* might alternatively be fulfilled through tasks *On-line Database* or *On-line Catalogue*. In addition, figure 4 models positive (+) or negative (-) contributions to softgoals:

- Task *Secure form* contributes positively to softgoal *Secure*, while the task *Standard form* contributes negatively;

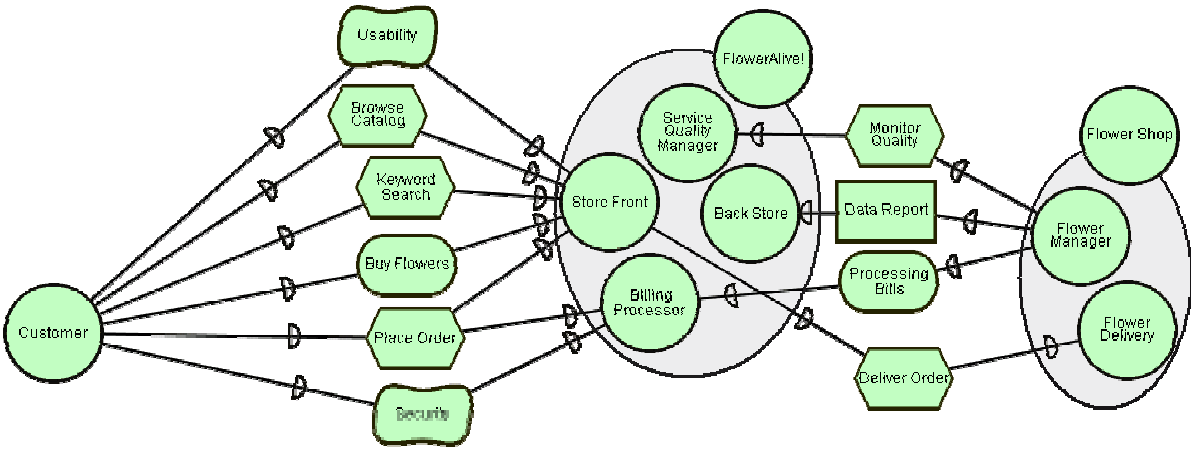

Figure 5: The web system consists of four inside actors, each with external dependencies

- Task dependencies *Keyword Search* and *Place Order* contribute negatively to softgoal *Secure;*
- Softgoal *Secure* contributes negatively to softgoal *Usable;*
- Task *On-line Catalogue* contributes positively to softgoal *Get Internet Customer;*
- As already mentioned, the root task *Internet Shop Managed* contributes positively to softgoal *Increase market share.*

The result of this analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been generated. Figure 5 suggests one possible assignment of responsibilities. The *FlowerAlive!* system is decomposed into four sub-actors: *Store Front*, *Billing Processor*, *Service Quality Manager* and *Back Store*. To complete the figure, we have also identified two sub-actors of the flower shop (not discussed further) interacting with the system: *Flower Manager* deals with the daily management of the shop and *Flower Delivery* carries out physical deliveries to customers. *Store Front* principally interacts with the *Customer* actor and provides her with a usable front-end web application. *Back Store* keeps track of all web information about customers, products, sales, bills and other data of strategic importance to *Flower Shop. Billing Processor* is in charge of the secure management of bills, and other financial data; also of interactions with financial stakeholders (not represented here), such as banks or credit card companies. *Quality Manager* is introduced in order to contribute to the fulfilment of the *Security* and *Usability* softgoals. This actor is responsible for security, confidentiality, integrity and accuracy issues, and is continuously looking for problems. Of course, all four sub-actors need to communicate and collaborate in running the system. For instance, *Store Front* communicates to *Billing Processor* relevant customer information required to process bills. *Store Front* and *Billing Information* are supervised by *Service Quality Manager* who monitors transactions looking for security gaps and usability bottlenecks. *Back Store* organizes, stores and backs up all information coming from *Store Front* and *Billing Processor* in order to produce statistical analyses, historical charts and marketing data.

For the rest of the discussion, we focus on *Store Front*. This actor is in charge of catalogue browsing and flower database searching, also provides on-line customers

with detailed information about flowers. We assume that different flower shops working with *FlowerAlive!* may want to provide their customers with various forms of information retrieval (Boolean, keyword, thesaurus, lexicon, full text, indexed list, simple browsing, hypertext browsing, SQL queries, etc.).

Store Front is responsible for supplying a customer with a web shopping cart to keep track of items the customer is buying when visiting *FlowerAlive!*. We assume that different flower shops using the *FlowerAlive!* system may want to provide customers with different kinds of shopping carts with respect to their internet browser, plug-ins configuration or platform (e.g., java mode shopping cart, simple browser shopping cart, frame-based shopping cart, CGI shopping cart, enhanced CGI shopping cart, shockwave-based shopping cart,…)

Store Front also decides what kind of processing will be done for a given order (phone/fax, internet standard form or secure encrypted form) and how the order will be concretely delivered to the customer. We assume that different flower shop managers using the *FlowerAlive!* web system may be processing various types of orders, such as those listed above differently. Also, the sub-actor *Store Front* relies on the *Flower Delivery* department to physically deliver orders. We postulate that orders can be delivered in different ways according to customers' wishes (UPS, FedEx, DHL, express mail, normal mail, groom service, VIP service, overseas service, …).

As discussed in Section 5, *Store Front* could be further decomposed during architectural design into more specific system actors like *Product Browser, Shopping Cart* and *Order Processor* respectively taking in charge each of the responsibilities described above, i.e., the flower database navigation, the items selection and the processing of orders.

Resource, task and softgoal dependencies correspond naturally to functional and non-functional requirements. Leaving (some) goal dependencies between system actors and other actors is a novelty. Traditionally, functional goals are "operationalized" during late requirements [7], while quality softgoals are either operationalized or "metricized" [12]. For example, *Billing Processor* may be operationalized during late requirements analysis into particular business processes for processing bills. Likewise, a security softgoal might be operationalized by defining interfaces which minimize input/output between the system and its environment, or by limiting access to sensitive information. Alternatively, the security requirement may be metricized into something like "No more than X unauthorized operations in the system-to-be per year".

Leaving goal dependencies with system actors as dependees makes sense whenever there is a foreseeable need for flexibility in the performance of a task on the part of the system. For example, consider a communication goal "communicate X to Y". According to conventional software development techniques, such a goal needs to be operationalized before the end of late requirements analysis, perhaps into some sort of a user interface through which user Y will receive message X from the system. The problem with this approach is that the steps through which this goal is to be fulfilled (along with a host of background assumptions) are frozen into the requirements of the system-to-be. This early translation of goals into concrete plans for their fulfilment makes software systems fragile and less reusable.

In our example, we have left two goals in the late requirements model. The first goal is *Usability* because we propose to implement *Store Front* and *Service Quality Manager* as agents able to automatically decide at run-time which product browser, shopping cart and

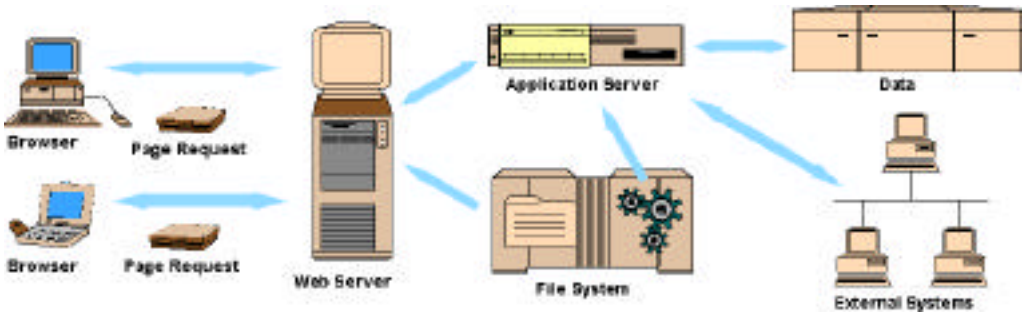

Figure 6: Canonical Web Architecture

order processor architecture fit better to the customer's needs or browser/platform specifications. Moreover, we would like to include different kinds of search engine reflecting search techniques proposed in information brokering or retrieval and let the system dynamically chooses the most appropriate with respect to the customer's needs. The second goal in the late requirements specification is *Security*. To fulfil this goal, we propose to provide in the system's architecture a number of security strategies and let the system decide at run-time which one is the most appropriate, taking into account environment configurations, web browser specifications and network protocols used.

So, instead of operationalizing these goals during requirements analysis, we propose to do so during architectural design.

5. Architectural Design

Architectural design has emerged as a crucial phase of the design process consisting of a number of structural elements and their interfaces. A software architecture constitutes a relatively small, intellectually manageable model of system structure, and how system components work together. For our internet flower shop example, the task is to define (or choose) a web application architecture. The canonical web architecture consists of a web server, a network connection, HTML/XML documents on one or more clients communicating with a Web server via HTTP, and an application server which enables the system to manage business logic and state (see figure 6). This architecture is not intended to imply that a web application cannot use distributed objects or Java applets; nor does it imply that the web server and application server cannot be located on the same machine.

Indeed, software architects have developed catalogues of web architectural style (see, for example, [6]). The three most common styles are the *Thin Web Client*, *Thick Web Client* and *Web Delivery*. The *Thin Web Client* is most appropriate for internet-based web applications, in which the client has minimal computing power or no control over its configuration. The client requires only a standard forms-capable web browser. All the business logic is executed on the server during the fulfilment of page requests for the client browser. The *Thick Web Client* style extends the *Thin Web Client* style with the use of client-side scripting and custom objects, such as ActiveX controls and Java applets. A significant amount of business logic can be executed on the client machine. Finally, in the *Web Delivery* style, the web is used primarily as a delivery mechanism for an otherwise traditional client/server system. The client communicates directly with object servers, bypassing HTTP. This style is appropriate when there is significant control over client and network configuration.

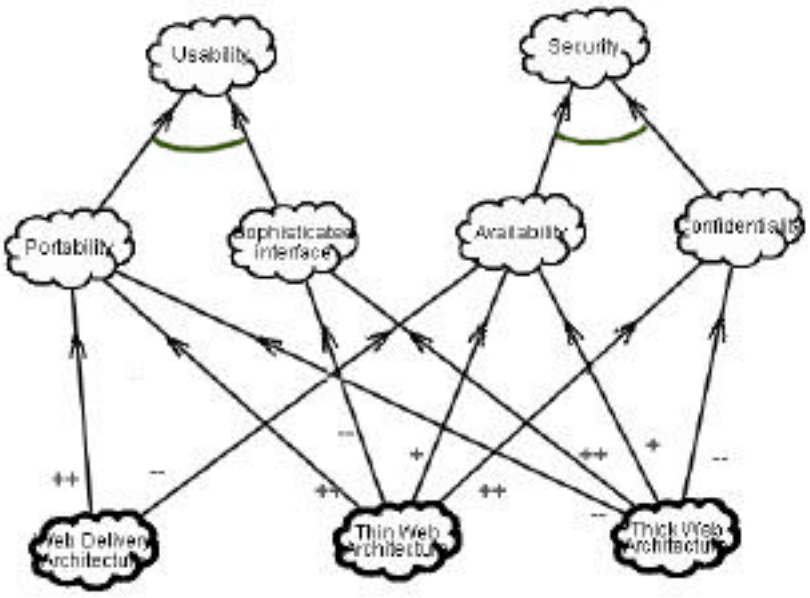

Figure 7: A strategic rationale model, from the perspective of the *FlowerAlive!* actor

During architectural design we concentrate on the key system actors, defined during late requirements analysis, and their responsibilities. These include the desired functionality of the system-to-be, as well as a number of quality requirements related to usability, security, performance, portability, availability, reusability, testability, etc.

Functional requirements can be accomodated using one of several standard methodologies, such as structured analysis and design, or object-oriented design methods. However, quality requirements are generally not addressed by such techniques [13]. For example, as we are building an Internet application, security is certainly an important concern. Indeed, this was captured by the *Security* software goal dependency between the *Customer* and *FlowerAlive!* actors (see figure 3). The software application should do only what it is supposed to do, without compromising the integrity of the data by exposing them to unauthorised users. Likewise, *Usability* is a concern, since customers may have little internet experience. Interfaces needs to be carefully crafted to handle in a user-friendly manner the communication between the customer and the system, as well as the flow of activities of the business process. To deal with this softgoal, we have introduced a *Usability* softgoal dependency between *Customer* and *FlowerAlive!* actors (see figure 3).

To cope with these goals, the software architect, who is another (external) actor, goes through a means-ends analysis comparable to what was discussed earlier. In this case, the analysis involves refining the softgoals to sub-goals that are more specific (and more precise!) and then evaluating alternative architectural styles against them, as shown in figure 7. This analysis is intended to make explicit the space of alternatives for fulfilling the top-level quality softgoals. Moreover, the analysis allows the evaluation of several alternative architectural styles. The styles are represented as operationalized softgoals (saying, roughly, “make the architecture of the

new system *Web Delivery-/Thin Web-/Thick Web*-based") and are evaluated with respect to the alternative non-functional softgoals as shown in figure 7. The evaluation results in contribution relationships from the architectural goals to the quality softgoals, labelled "+', "-", "++", etc.

The *Usability* softgoal has been AND-decomposed into subgoals *Portability* and *Sophisticated Interface.* From the customer point of view it is important for the *FlowerAlive!* application to be portable across browser implementations. Equally important is the quality of the interface. Note that not all HTML browsers support scripting, applets, controls and plug-ins. These technologies make the client itself more dynamic, and capable of animation, fly-over help, and sophisticated input controls. When only minimal business logic needs to be run on the client, scripting is often an easy and powerful mechanism to use. When truly sophisticated logic needs to run on the client, building Java applets, Java beans, or ActiveX controls is probably a better approach. ActiveX, however, is an option only when the client computers are Windows-based.

The *Security* softgoal has initially been AND-decomposed into subgoals *Availability* and client *Confidentiality.* The former guards against interruption of service, the latter guards against unauthorised disclosure of information. Network communication may not be very reliable causing sporadic loss of the server. Clients, especially those on the internet are, like servers, at risk in web applications. It is possible for web browsers to unknowingly download content and programs that could open up the client system to crackers and automated agents all over the net. JavaScript, Java applets, ActiveX controls, and plug-ins all represent a certain degree of risk to the client and the information it manages.

The *Thin Web Client* architecture is useful for internet-based applications, for which only the most basic client configuration can be guaranteed. Hence, this architecture does well for portability (figure 7). However, it has a limited ability to support sophisticated user interfaces. The browser acts as the entire user interface delivery mechanism and in most common browsers these are limited to a few text entry fields and button. Moreover, this architecture relies on a connectionless protocol such as HTTP, which contributes positively to availability of the system since the sporadic loss of a server might not pose a serious problem. Pure HTTP, without client-side scripting, is rather secure.

On the other hand, the *Thick Web Client* architecture is generally not portable across browser implementations. Not all HTML browsers support JavaScript or VBScript. Additionally, only Microsoft Windows base clients can use ActiveX controls. However, these technologies contribute very positively to the goal of having sophisticated interfaces. As in the *Thin Web Client* architecture, all communication between client and server is done with HTTP. Since HTTP is a "connectionless" type of protocol, most of the time there is no open connection between client and server. Only during page requests does the client send information. Hence its positive contribution to availability (figure 7). On the negative side, client-side scripting and custom objects, such as ActiveX controls and Java applets may pose risks to the client confidentiality.

Last but not least, the *Web Delivery* architecture is highly portable, since the browser has some built-in capabilities to automatically download the needed components from the server. However, this architecture requires a reliable network. Connections between client and server objects last much longer than do HTTP connections, and so sporadic loss of the server, poses a

serious problem that has to be addressed for this architecture.

As with late requirements, an interesting feature of the proposed analysis method is that it is goal-oriented. Goals are introduced and analysed during architectural design, and guide the design process.

Apart from goal analysis, this phase involves the introduction of other system actors which will take on some of the responsibilities of the key system actors introduced earlier. For example, to accommodate the responsibilities of the *Store Front* actor of figure 5, the architect may want to introduce actors for placing and tracking orders, browsing the catalogue and managing customer profile. Of course, other actors may be included to deal, for example, with notification of the arrival of new items and recommendation (prediction based on profile and "business intelligence", possibly derived through data-mining techniques).

An interesting decision that comes up during architectural design is whether fulfilment of an actor's obligations will be accomplished through assistance from other actors, through delegation ("outsourcing"), or through decomposition of the actor into component actors. Going back to our running example, the introduction of other actors described in the previous paragraph amounts to a form of delegation in the sense that *Store Front* retains its obligations, but delegates subtasks, subgoals etc. to other actors. An alternative architectural design would have *Store Front* outsourcing some of its responsibilities to some other actors, so that *Store Front* removes itself from the critical path of obligation fulfilment. Lastly, *Store Front* may be refined into an aggregate of actors which, by design, work together to fulfil *Store Front*'s obligations. This is analogous to a committee being refined into a collection of members who collectively fulfil the committee's mandate. It is not clear, at this point, how the three alternatives compare, nor what are their respective strengths and weaknesses.

6. Detailed Design

The detailed design phase is intended to introduce additional detail for each architectural component of a software system. In our case, this includes actor communication and actor behaviour. To support this phase, we may be adopting agent communication languages, message transportation mechanisms, ontology communication, agent interaction protocols, etc. from the agent programming community. One possibility, among admittedly many, is to adopt one of the extensions to UML proposed by the FIPA (Foundation for Intelligent Agents) and the OMG Agent Work group [14, 15]. For our example, let's concentrate on the *Buy Flowers* goal dependency, which might involve a detailed design of an *agent interaction protocol* (AIP). To define such a protocol, we use AUML - the Agent Unified Modeling Language [15], which supports templates and packages to represent the protocol as an object, but also in terms of sequence and collaborations diagrams. In AUML inter- and intra-agent dynamics are also described in terms of activity diagrams and state charts.

Figure 8 depicts a customisation of the FIPA Contract Net protocol to a particular scenario involving *Customer* and *Store Front* actors. Such a protocol describes a communication pattern among actors as an allowed sequence of messages, as well as constraints on the contents of those messages.

When a *Custome*r actor needs to buy some flowers, a request for proposal message (*Store Front-rfp*) is sent to the *Store Front* actor. The *Store Front* actor can then choose to respond to the *Customer* before a given deadline: by refusing to provide a proposal, submitting a proposal, or informing that it did

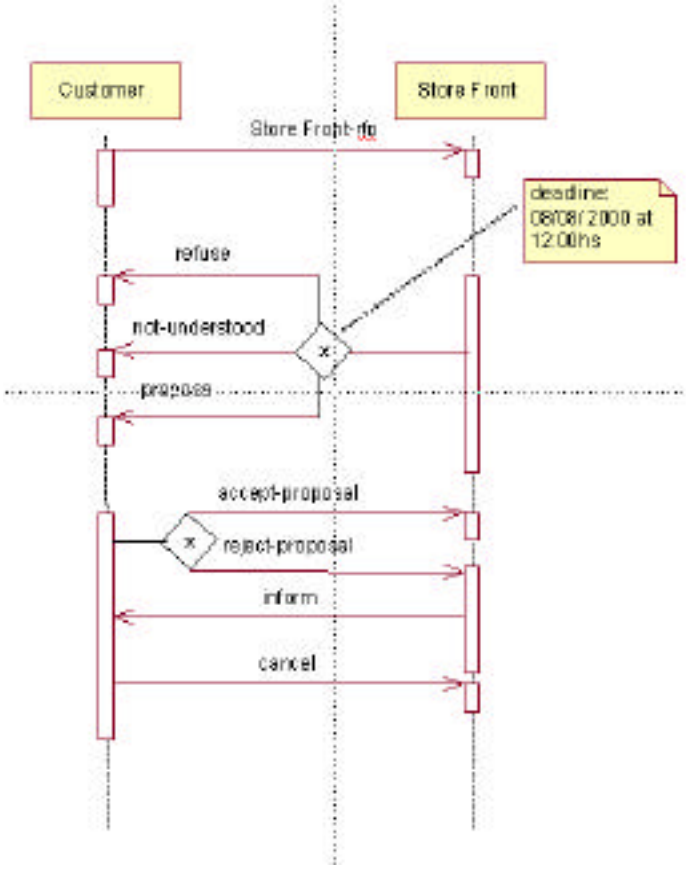

Figure 8. Agent Interaction Protocol for buying flowers

not understand (the diamond symbol indicates a decision that can result in zero or more communications being sent – depending on the conditions it contains; the "x" in the decision diamond indicates an *exclusive or* decision). If a proposal is offered, the *Customer* has a choice of either accepting or rejecting the proposal. When the *Store Front* actor receives proposal acceptance, it will inform the *Customer* about the proposal's execution. Additionally, the *Customer* actor can cancel the execution of the proposal at any time.

Of course the sequence diagram of figure 8 only provides a basic specification for an agent claim processing protocol. More processing details are required. For example, a *Customer* actor requests a call for proposals (CFP) from a *Store Front* actor. However, the diagram stipulates neither the procedure used by the *Customer* to produce the CFP request, nor the procedure employed by the *Store Front* actor to respond the CFP. Yet, these are clearly important details at this stage of the software development process.

Such details can be provided by using *levelling*, i.e., by introducing additional interaction and other diagrams which describe some of the primitive action shown on figure 8. Each additional level can express *intra-actor* or *inter-actor* activities. At the lowest level, specifications of an actor protocol requires spelling out the detailed processing that takes place within an actor in order to implement the protocol.

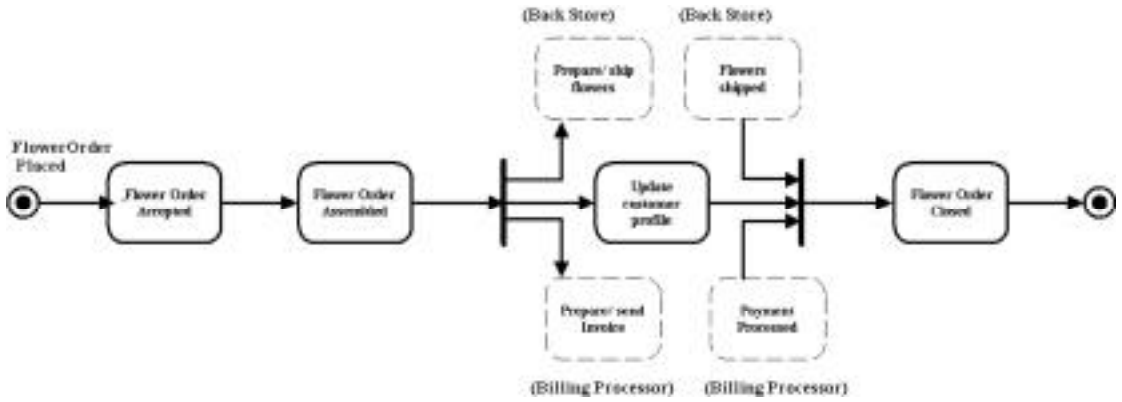

Figure 9: An activity diagram that specifies the order processing behaviour for *Store Front* Actor

State charts and activity diagrams can also be used to specify the internal processing of actors who are not aggregates. For example figure 9 depicts the detailed processing of orders by the *Store Front* actor. The diagram indicates that the actor process is triggered by placing a flower order and ends with the order completed. The internal processing is expressed as an activity diagram, where the *Store Front* actor is responsible for accepting, assembling, customer profiling and closing the flower order. The dotted operation boxes represent interfaces processes carried out by external actors. For example, the diagram indicates that when the flower order has been assembled, three actions are triggered concurrently. Furthermore, when both the payment has been accepted and the flowers shipped, the Close Order process can be invoked.

7. Conclusions and Discussion

We have argued in favour of a software development methodology which is founded on intentional concepts, such as those of actor, goal, (goal, task, resource, softgoal) dependency, etc. Our argument rests on the claim that enterprise software should be organized the same way enterprises are. Moreover, we have argued that current software development techniques lead to inflexible and non-generic software. This is the case because the elimination of goals during late requirements, freezes into the design of a software system a variety of assumptions which may or may not be true in its operational environment. Given the ever-growing demand for generic, component-ized software that can be downloaded and used in a variety of computing platforms around the world, we believe that the use of intentional concepts during late software development phases will become prevalent and should be further researched.

The Tropos project is only beginning and much remains to be done. We will be working towards a modeling framework which views software from four complementary perspectives:

- **Social** -- who are the relevant actors, what do they want? What are their obligations? What are their capabilities?…
- **Intentional** -- what are the relevant goals and how do they interrelate? How are they being met, and by whom?…
- **Process-oriented** -- what are the relevant business/computer processes? Who is responsible for what?…
- **Object-oriented** – what are the relevant objects and classes, along with their inter-relationships?

In this paper, we have focused the discussion on the social and intentional perspectives because they are novel. As hinted earlier, we propose to use UML-type modeling techniques for the others.

Of course, diagrams are not complete, nor formal as software specifications. To address this deficiency, we propose to offer three levels of software specification. The first is strictly diagrammatic, as discussed in this paper. The second involves formal annotations which complement diagrams. For example, annotations may specify that some obligation takes precedence over another. These could be used as a basis for simple forms of analysis. Finally, we propose to include within Tropos a formal specification language for all built-in constructs, to support deeper forms of analysis. Turning to the organization of Tropos models, the concepts of *i** will be embedded in a modeling framework which supports generalization, aggregation, classification, materialization and contextualization.

Acknowledgements

Many colleagues contributed to the ideas that led to this paper. Special thanks to Eric Yu, whose insights helped us focus our research on intentional and social concepts. The Tropos project includes as co-investigators Eric Yu (University of Toronto) and Yves Lesperance (York University); also Alex Borgida (Rutgers University), Matthias Jarke and Gerhard Lakemeyer (Technical University of Aachen.) The Canadian component of the project is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the CITO Centre of Excellence, funded by the Province of Ontario. This work was carried out while the first author was visiting the Department of Computer Science, University of Toronto (partially supported by the CNPq – Brazil grant 203262/86-7).

References

[1] Yu, E., *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1995.

[2] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.

[3] Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

[4] Wirfs-Brock, R., Wilkerson, B., Wiener, l., *Designing Object-Oriented Software.* Englewood Cliffs, NJ; Prentice-Hall, 1990.

[5] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[6] Conallen, J., *Building Web Applications with UML*, The Addison-Wesley Object Technology Series, Addison-Wesley, 2000.

[7] Dardenne, A., van Lamsweerde, A., and Fickas, S., "Goal–directed Requirements Acquisition," *Science of Computer Programming, 20*, 3-50, 1993.

[8] Cohen, P. and Levesque, H. "Intention is Choice with Commitment". *Artificial Intelligence, 32(3).*

[9] Yu, E., "Modeling Organizations for Information Systems Requirements Engineering," *Proceedings First IEEE International Symposium on Requirements Engineering*, San Jose, January 1993, pp. 34-41.

[10] Yu, E., and Mylopoulos, J., "Using Goals, Rules, and Methods to Support Reasoning in Business Process Reengineering", *International Journal of Intelligent Systems in Accounting, Finance and Management 5(1)*, January 1996.

[11] Yu, E. and Mylopoulos, J., "Understanding 'Why' in Software Process Modeling, Analysis and Design," *Proceedings Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[12] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993.

[13] Chung, L. K., Nixon, B. A., Yu, E., Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Publishing, 2000.

[14] Odell, J., Bock, C., *Suggested UML Extensions for Agents.* OMG document ad/99-12-01; Submitted to the OMG's Analysis and Design Task Force (ADTF) in response to the Request of Information (RFI) entitled "UML2.0 RFI". December 1999.

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