2 From Software Engineering to Project Management

This chapter explores the state of the art in the Software Engineering (SE) and Project Management (PM) fields. It covers the vertical and horizontal dimensions of software development life cycles and well known methodologies and techniques as the Rational Unified Process, the Unified Modelling Language, Agile Modelling and eXtreme Programming. It also exposes Software Project Management in general and studies Risk, Time and Quality Management in particular.

Section 2.1 defines the Software Engineering field, the four fundamental phases of software development, *analysis*, *design*, *implementation* and *test*, pointed out by [Leroy98] are then exposed in section 2.2. Those phases, defined here in a generic manner, are interpreted by most modern SE methodologies i.e. each methodology defines his own analysis, design, implementation and test phases in terms of scope, models to be build, roles involved, etc. A brief introduction to software development life cycles will be done afterwards in Section 2.3. Some representative existing SDLC will also be studied. Section 2.4 exposes the Rational Unified Process and section 2.5 agile modeling techniques. The chapter then turns on to project management, it overviews software project management in section 2.6; risk, time and quality management are respectively studied in sections 2.7, 2.8 and 2.9. Finally, the chapter takes some conclusions in Section 2.10.

2.1 Software Engineering: a Definition

Software Engineering (SE) is an engineering discipline concerned with all the aspects of software production. According to [AN02], a SE methodology is made of a *modeling language* and a *software development process*. On the one hand, the modeling language is the syntax made of concepts associated with visual icons used to build models (see for example the *Unified Modeling Language* [UML, RBJ99, BRJ99]). On the other hand, the software development process defines the who, what, when and how dimensions of software development. A software development process can generally be split up into four phases - analysis, design, implementation, and test - and follows a particular life cycle - sequential, incremental, iterative, spiral, etc.

2.2 Software Development Phases

This section overviews the four traditional phases of a software engineering process: analysis, design, implementation and test.

2.2.1 Analysis

Requirements are "*capabilities and conditions to which the system must conform*" [JBR99]. They can be functional - capture the intended behaviour of the system - or

non-functional – attributes, qualities of the system that cannot be directly implemented by technical means but result of it (such as security, availability, etc.).

The analysis phase has two main purposes:

- To model stakeholders' (end-users, managers, engineers, etc.) requirements i.e. to provide models allowing the software development teams and stakeholders to agree on what the system should do;
- To build a domain model, i.e., a representation of all the relevant concepts or real-world entities in a particular domain of interest.

The analysis phase is *problem-oriented*, focused on defining the problem (*what the problem is*) independently of any solution. Because of the existence of two complementary dimensions in software analysis, this phase is sometimes split by SE methodologies into two distinct ones.

2.2.2 Design

The design phase is concerned with the elaboration of a solution to the problem described during the analysis phase. It is a system-oriented phase focusing on *how to solve the problem*.

According to [Weyns04], the detailed design phase is in most SE methodologies split in two distinct levels:

- *Architectural design* aimed to build a system architecture that satisfies both functional and non-functional requirements;
- *Detailed design* aimed to build a model of detailed diagrams and descriptions of all the elements defined in the systems architecture. This model can directly be taken as input for the implementation phase.

2.2.3 Implementation

The implementation phase is concerned with the component building from scratch or by composition. On the basis of the models developed during the previous phases, the team should build exactly what has been requested even if there remains place for innovation and flexibility. For example components can be designed too narrowly for a particular system and it can be made more generic so that it can be reused.

Quality, performance, baselines, libraries, and debugging issues should be carefully considered at implementation level. The phase output is the executable product.

2.2.4 Test

The testing phase is concerned with the evaluation of software correctness, completeness, security and quality. Many approaches to software testing exist but effec-

tive testing of complex products is essentially a process of investigation rather than creating and following well defined procedure.

2.3 System Development Life Cycles

According to [Royce98], a System Development Life Cycle (SDLC) is "*a conceptual model used in project management to describe the stages involved in a system development project from an initial feasibility study through maintenance of the completed application*". Several SDLC such as the Sequential/Waterfall Model (see [Royce70]), the V-Model (see [Forsberg97]), the Incremental Model (see [Dorfman97]), the Evolutionary Model or the Spiral Model [Boehm88, Boehm94] have been proposed over the years. Depending on the project, some SDLC appear to be more adequate than others. For the development of huge and complex user-intensive enterprise information systems, the spiral model is more appropriate due to its iterative nature than the use of a traditional waterfall model. Indeed, for the reasons that will be exposed in this section, the former is much less risky than the later, especially in the later stages of the project. This section studies the evolution of SDLCs in SE history.

2.3.1 The Waterfall Model

The waterfall model [Royce70] describes a development process based on a series of phases, typically the four ones depicted previously into this chapter, fulfilled one after another in a linear/sequential way. This model, represented in figure 2-1, is the oldest project life cycle over SE history, it is however widely criticized (see for example [Boehm00a, Hanna95]); root causes are:

- Real life project does seldom follow the linear flow of the model. Since feed back loops are not envisaged by the model, ex-post modifications into the achieved disciplines causes confusions;
- As overviewed into the introduction, requirements can rarely be defined totally at the beginning of the software project;
- The first working version of the program is delivered at the very end of the project when the potential problems discovery makes them difficult to review.

Moreover as pointed out by [BPV94], the waterfall model can lead to *blocking states* in which members of the project team have to wait for other members to complete dependent tasks. This can lead to a considerable loss of productivity.

Despite its disadvantages, the waterfall model constitutes the fist attempt to build a framework for the proper incorporation of the traditional software development phases. In some specific cases, i.e., when all the assumptions defined in [Boehm00a] are met, this model can be optimal that is why it is then still in use today.

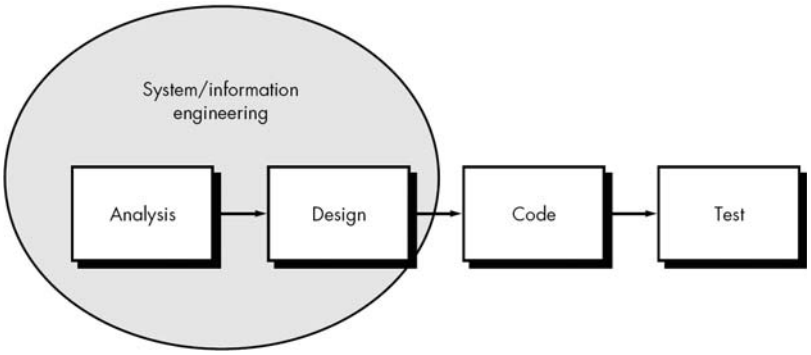


Figure 2-1. The Waterfall Model (from [Pressman01]).

2.3.2 The V-Model

The V-Model (or VEE model) is a software development model designed to simplify the understanding of the complexity associated with developing software. It is used to define a uniform procedure for product of project development. The V-model is a variation of the waterfall model. The essential advance of this model is that it supports development with testing. Steps are sequential and it is easy to define the control points and each sub-product that comes out after every step.

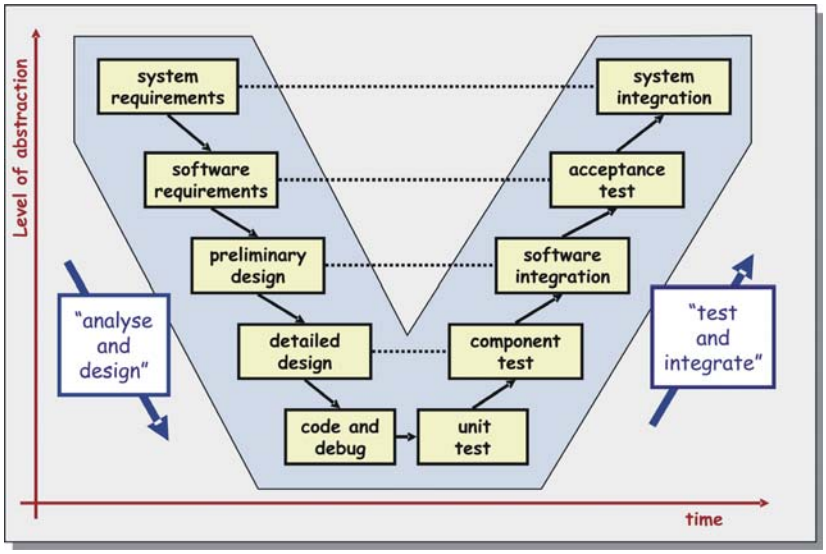


Figure 2-2. The V-Model (adapted from [Fotzberg97]).

In a V-model the software process has two branches: left branch includes the phases before implementation and right branch the phases after implementation (see Figure 2-2). Besides this the V-model defines three sub-models: User model, architectural model and implementation model. *User model* describes how customers point of view is a part of SW process and that is why this is the most important phase of the process. *Architectural model* creates a basis and framework for the next phases in the model. *Implementation model* defines the design of the software's internal structure, implementation phases and releases.

2.3.3 The Incremental Model

In the *incremental model* the phases of the waterfall model are repeated using an iterative prototyping philosophy. Figure 2-3 shows how the incremental model applies linear sequences in different stages while calendar time progresses. Each linear sequence produces a deliverable "increment" of the software [MDE93]. For example, when developing a web browser using the incremental model, the first increment can produce basic URL management, html editing and saving functions; more sophisticated editing (including third parties media browsing software) and navigation capabilities in the second increment; an html script editor in the third increment; and advanced navigation issues in the fourth.

Using incremental model, the first increment is often a *core product* [Pressman01]. At this stage, the basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is tested by the customer (or is reviewed in detail). After evaluation, next increment is planned. The plan addresses the modification of the core product to better meet the user requirements. Such a process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model is iterative by nature; it focuses on the delivery of an operational product with each increment. Early increments do provide capability that serves the user and also provide a basis for user evaluation.

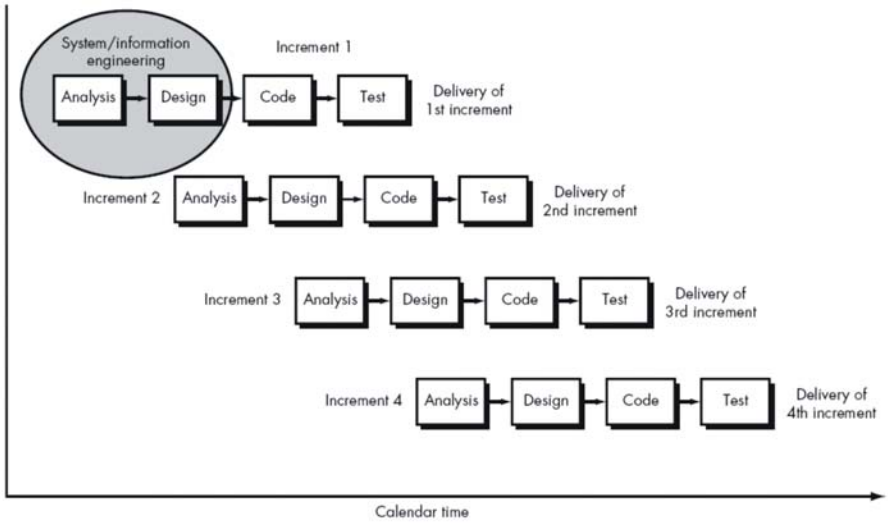


Figure 2-3. The Incremental Model (from [Pressman01]).

2.3.4 The Spiral Model

The spiral model is defined in [Boehm00a] as *"a **risk**-driven **process model generator** used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a **iterative cyclic** approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of **anchor point milestones** for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions"*.

The assumptions for an optimal waterfall SDLC are seldom met; causes were depicted in the introduction of this dissertation. That is why we need for a SDLC that includes continuous requirements acquisition and modelling: using an spiral SDLC implies that risk is considered during the early stages of the project not at the late ones as in a process fully driven by sequential activities.

The spiral model is composed of several activities called *task regions*. Their number varies from three to six. The spiral model depicted in Figure 2-4 (from [Pressman01]) contains six task regions:

- Customer communication encompasses the tasks required to establish effective communication between developer and customer;

- Planning encompasses the tasks required to define resources, timelines, and other project related information;

- Risk analysis encompasses the tasks required to assess both technical and management risks;
- Engineering encompasses the tasks required to build one or more representations of the application;
- Construction and release encompasses the tasks required to construct, test, install, and provide user support (e.g., documentation and training);
- Customer evaluation encompasses the tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each of those regions are fulfilled using work tasks, called a *task set,* tailored to the specificities of the undertaken project.

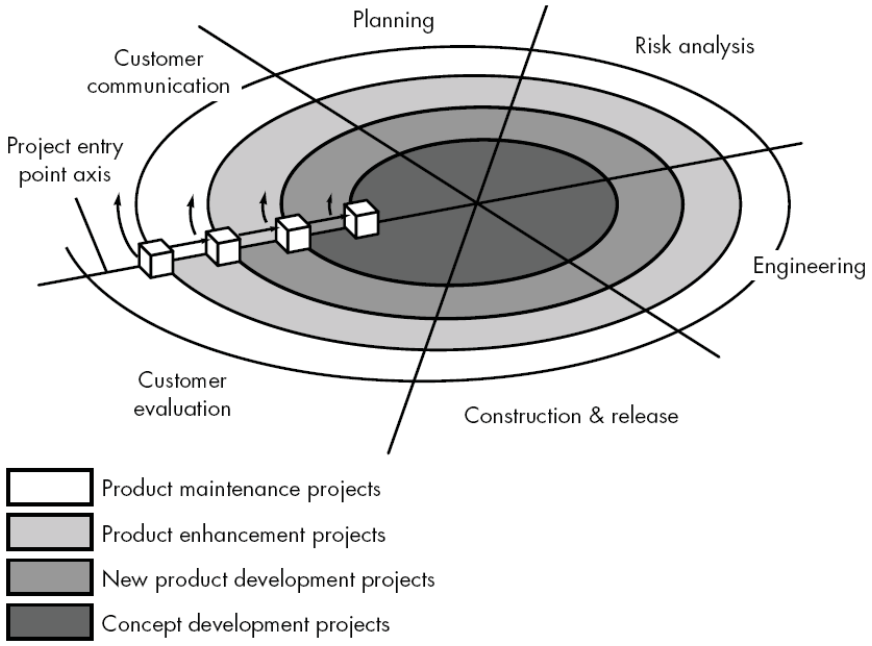


Figure 2-4. The Spiral Model (from [Pressman01]).

Facing this evolutionary process, the software engineering team runs into the spiral in a clockwise direction, beginning from the center. First circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback de-

rived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

2.3.5 The WIN-WIN Spiral Model

The idea of the WIN-WIN Spiral Model is based on the idea that communication with stakeholders is not trivial. Indeed, rather than a simple transcription of their requirements into a software product, the requirements engineering stage is often a matter of compromise. This means that stakeholders and software professionals often enter into a process of negotiation, where the first may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market.

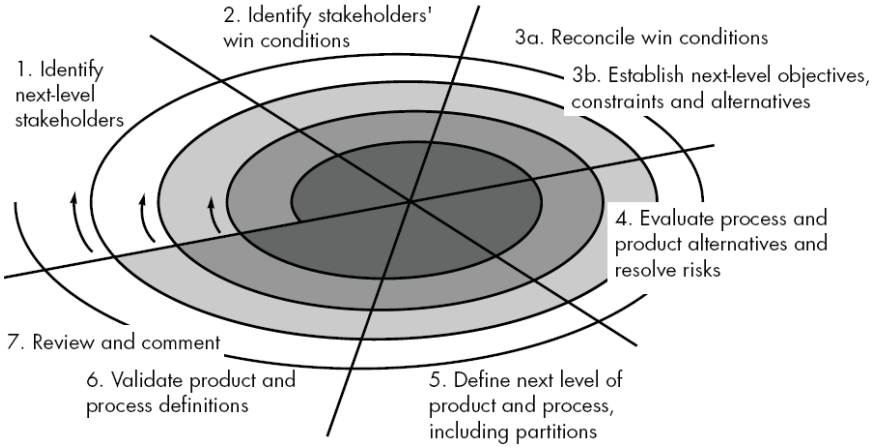


Figure 2-5. The WINWIN Spiral Model (from [Boehm98, Pressman01]).

Negotiations should be driven to achieve a "win-win" result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs while software professionals win by working to realistic and achievable budgets and deadlines.

The WINWIN Spiral Model as defined in [Boehm98] defines a series of negotiation activities when beginning a pass around the spiral. In the model, Customer communication is divided three distinct activities.

1. Identification of the system or subsystem's key "stakeholders".
2. Determination of the stakeholders' "win conditions".
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team).

A successful completion of these activities achieves a win-win result, becoming the key criterion for the software and system definition.

Another interesting contribution of this model is the introduction of three process milestones, called in [Boehm96] *anchor points*. These are used to evaluate the completion of a cycle around the spiral and provide decision milestones before the software project proceeds. Three anchor points representing three different views of the of progress are defined:

- *Life Cycle Objectives* (*LCO*) defines a set of objectives for each major software engineering activity;
- *Life Cycle Architecture* (*LCA*) establishes objectives that must be met as the system and software architecture is defined;
- *Initial Operational Capability* (IOC) represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.

The Rational Unified Process presented into the next section inherited those milestones.

2.4 The Rational Unified Process

As pointed out before, a SE methodology is made of a *modeling language* and a *software development process*. The Rational Unified Process (RUP) is a software development process using the Unified Modeling Language (UML) as modelling language.

2.4.1 The Modeling Language

The *Unified Modeling Language* or *UML* (see [UML]) is a modelling language to specify, visualize, construct and document software systems. UML has been normalized by the *Object Management Group* (*OMG*) in order to furnish a universal communication support because it is independent from application domain and programming languages. It is issued from the merging of the tree methodologies of Booch (OOT), Rumbaugh (OMT) and Jacobson (OOSE), even if its scope is much broader. UML has become an international standard, it has significantly matured since its version 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision, which was adopted by the OMG in 2003.

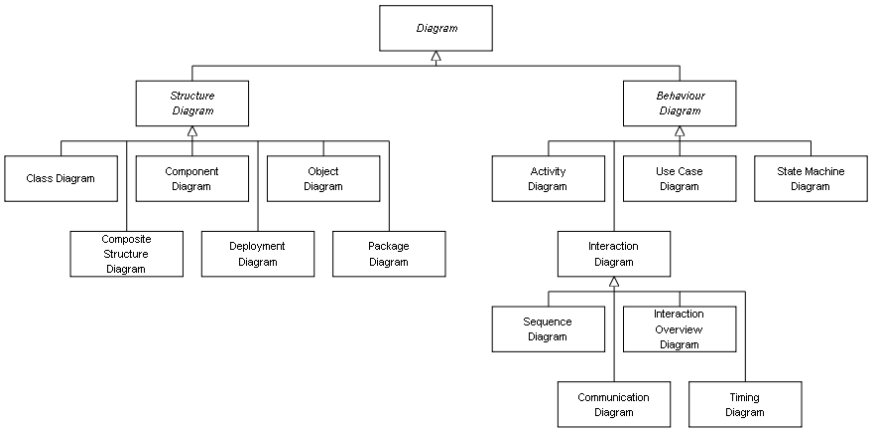


Figure 2-6. UML Diagrams Hierarchy.

The structure of the UML 2.0 diagrams is represented in Figure 2-6; it provides multiple system views as well as multiple layers of abstraction. These views are represented by different types of diagrams:

- Structure diagrams encompass
    - The class diagram which models the conceptual architecture of the system by describing the used classes and the relationship between those classes. The relationships posses cardinalities and/or special behaviour between classes as for example inheritance;
    - The object diagram which is a particular example of the class diagram, it is a class-instance. This diagram describes the objects and the relationship between those objects;
    - The component diagram which models the physical architecture of the application and describe the modules and the relationships between modules. A module models a source file or imported library. The component diagram highlight the dependencies between files, what allows to identify the future constraints linked to compilation;
    - The deployment diagram which models the involved material resources (PC, modem, server, etc.). A node characterizes a resource. The relationships between nodes describes who communicates with who and how by describing the communication support;
    - The composite structure diagram which shows the internal structure of a class and the *collaborations* that this structure makes possible. This can include internal *parts*, *ports* through which the parts interact

with each other or through which instances of the class interact with the parts and with the outside world, and *connectors* between parts or ports. A *composite structure* is a set of interconnected elements that collaborate at runtime to achieve some purpose. Each element has some defined *role* in the collaboration;

- The package diagram which depicts how a system is split up into logical groupings by showing the dependencies among these groupings. As a package is typically thought of as a directory, package diagrams provide a logical hierarchical decomposition of a system.

- Behavior diagrams encompass:

  - The use-cases diagram which models stakeholders' requirements by characterizing the interactions between the actors and the system. The actors are the different types of users that will use the system. The actors can visualize the information proposed by the system. They can modify the system state, in the case the system has to furnish an answer to this modification. This view is abstract, it allows to identify systems required functionalities so that users can fulfill their tasks. Use cases allow to understand the way the overall system operates;

  - The activity diagram which is the representation of a method behavior or a use-case process. This formalism allows to express sequence, synchronization, parallelism and conditions as "if – then – else";

  - The state machine diagram which models the different state transitions occurring during the object life cycle. The state is a period of time during which the object satisfies to a series of conditions on his own state or on what he is achieving. The different object states are linked and their transitions take place when events take place. Those events can be a message generation, temporal conditions, etc.

- Interaction diagrams encompass:

  - The communication diagram which models the interactions between objects and actors. This representation owns an equivalent sequentiality but focuses on the objects organization. Messages temporality is accessible only through the observation of the messages numbers;

  - The sequence diagram which models the actions between objects and actors in a temporal manner, the actions sequence is represented. The sequence diagrams illustrate the use cases by adding a temporal dimension (represented by a vertical line). The actions between objects and actors are called messages. The messages can, following their graphical representation, possess a different syntax;

- The timing diagrams which are used to explore objects behavior throughout a given period of time. A timing diagram is a special form of a sequence diagram. The differences between timing diagram and sequence diagram are the axes are reversed so that the time is increased from left to right and the lifelines are shown in separate compartments arranged vertically;

- The interaction overview diagram which is a form of activity diagram with nodes representing interaction diagrams (i.e., sequence, communication, interaction overview or timing diagrams). Most of the notation for interaction overview diagrams is the same for activity diagrams but interaction overview diagrams introduce two new elements: interaction occurrences and interaction elements.

2.4.2 The Software Development Process

The Unified Process (UP), the Rational Unified Process (RUP), the Agile Unified Process (AUP) and the Enterprise Unified Process (EUP) (see [Jacobson99, Jacobson00, Kruchten03, RUP, ANV05]) are a family of iterative software development processes allowing to produce quality information systems answering to users requirements. All of those based on the same engineering concepts (disciplines, iterations, phases, etc.) and using UML as modeling language but differing a little in terms the disciplines and phases scopes. UP and the RUP can be considered as the same processes since RUP is the commercial version of UP while the AUP is a subset of the RUP himself a subset of the EUP. Nowadays, the RUP is the most mature process from the UP family; AUP and EUP remain experimental variations of the former; that is why the RUP that will be presented here.

RUP is an agile modelling methodology; it respects the core principles depicted earlier. Furthermore, RUP is:

- Use-Case Driven: the process employs Use Cases to drive the development process from the beginning of the project to its end;

- Architecture Centric: the process seeks to understand the most significant static and dynamic aspects in terms of software architecture. The architecture is a function users need and is captured in the core Use Cases;

- Iterative and incremental: The process recognizes that it is practical to divide large projects into smaller ones or mini-projects. Each mini-project comprises an iteration that results in an increment. An iteration may encompass all of the workflows in the process. The iterations are planned using Use Cases.

The Rational Unified Process is driven by a two dimensional architecture. The vertical axis represents the process' workflows or disciplines.

The Rational Unified Process consists of cycles that may repeat over the long-term life of a system. A cycle consists of four phases: Inception, Elaboration, Construction and Transition. Each cycle is concluded with a release, there are also releases within a cycle. The four phases defined by the RUP are:

- **Inception Phase** - During the inception phase the core idea is developed into a product vision. In this phase, we review and confirm our understanding of the core business drivers. We want to understand the business case for why the project should be attempted. The inception phase establishes the product feasibility and delimits the project scope;

- **Elaboration Phase** - During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed. This phase focuses on the "Do-Ability" of the project. We identify significant risks and prepare a schedule, staff and cost profile for the entire project;

- **Construction Phase** - During the construction phase the product is moved from the architectural baseline to a system complete enough to transition to the user community. The architectural baseline grows to become the completed system as the design is refined into code;

- **Transition Phase** - In the transition phase the goal is to ensure that the requirements have been met to the satisfaction of the stakeholders. This phase is often initiated with a beta release of the application. Other activities include site preparation, manual completion, and defect identification and correction. The transition phase ends with a postmortem devoted to learning and recording lessons for future cycles.

2.5 Agile Modeling

Agile software development is a conceptual framework for undertaking software engineering projects. There are a number of agile software development methods, such as those espoused by the Agile Alliance, a non-profit organization that supports individuals and organizations using agile approaches in software development. Agile methods attempt to minimize risk by using iterative software development with short iterations lasting typically one to four weeks.

2.5.1 Agile Principles

According to [Ambler02], the core principles of agile modeling are:

- **Modeling With a Purpose**. Developers should continuously ask themselves *why* they are creating an artifact[1] as well as *who* they are creating it for. Does the artifact help to understand an aspect of your software better, to communicate the followed approach to senior management to justify your project, or to create documentation that describes the developed system to the people who will be operating and/or maintaining/evolving it over time? The first step is to identify a valid purpose for creating a model and the audience for that model, then based on that purpose and audience it has to be developed to the point where it is both sufficiently accurate and sufficiently detailed.

- **Maximize Stakeholder Investment**. Stakeholders must have the final word in how the resources (time, money, facilities, etc.) they are devoting to the project are invested or not.

- **Travel Light** Every artifact created that the project manager decides to keep, will need to be maintained over time. The more the number of artifacts kept is high, the more the resources spend to maintain them will be high. Similarly, the more complex/detailed the models are, the more likely it is that any given change will be harder to accomplish. Every time the project manager decides to keep a model he trades-off agility for the convenience of having that information available to the team in an abstract manner. A development team that decides to develop and maintain a detailed requirements document, a detailed collection of analysis models, a detailed collection of architectural models, and a detailed collection of design models is likely to spend the majority of his time updating documents instead of writing source code.

- **Multiple Models**. The team potentially needs to use multiple models to develop software because each model describes a single aspect of the software. Considering the complexity of modern information systems, a wide range of models are required to cover all those aspects. An important point is that all of these models do not have to be developed for any given system, but depending on the nature of the software developed, at least a subset of the models is required.

- **Rapid Feedback**. The time between an action and the feedback on that action is critical. By working with other people on a model, particularly with a shared modeling technology, analysts obtain a near-instant feedback on their ideas. Working closely with the customer, to better understand and analyze his requirements or to develop a user interface that meets his needs provides opportunities for rapid feedback.

- **Assume Simplicity**. Software development should be based on the assumption that the simplest solution is the best solution. Software should not be

[1] An artifact is a document such as a text document or a model taken as input or output in the SE activities.

overbuilt; no additional features not needed today should be added to models. In Agile Modeling models should be kept as simple as possible.

- **Embrace Change**. Requirements evolve over time. People's understanding of the requirements change over time. Stakeholders can change during the project, new people are added and existing ones can leave. Stakeholders can change their points of view as well, potentially changing the goals and success criteria of the project. Consequently, project's environment changes as efforts progress, the approach to development must reflect this reality.

- **Incremental Change**. In SE modeling models do not need to be right from the first time, in fact, it is very unlikely that they could be. Furthermore, every single detail do not need to be captured in the models, they just need to get it good enough at the time. Instead of futilely trying to develop an all encompassing model at the start, developers can put a stake in the ground by developing a small model, or perhaps a high-level model, and evolve it over time (or simply discard it when it is no longer needed) in an incremental manner.

- **Quality Work**. Work done by the development team has to be as good as possible.

- **Software is the Primary Goal**. The goal of software development is to produce software that meets stakeholders requirements in an effective manner. The primary goal is not to produce extraneous documentation, extraneous management artifacts, or even models. Any activity that does not directly contribute to this goal should be questioned and avoided if it cannot be justified in this light.

- **Enabling the Next Effort is the Secondary Goal**. The software project can be considered as a failure even if a working system has been furnished to the users – part of fulfilling the needs of the project stakeholders is to ensure that your system robust enough so that it can be extended over time. The next effort may be the development of the next major release of your system or it may simply be the operations and support of the current version being built. To enable it enough documentation and supporting materials need to be developed so that the next development effort can be effective.

- **Content is More Important Than Representation**. Any given model could have several ways to represent it. Consequently, a model does not need to be a document. Even a complex set of diagrams created using a CASE tool may not become part of a document, instead they are used as inputs into other artifacts, very likely source code, but never formalized as official documentation. The point is that you take advantage of the benefits of modeling without incurring the costs of creating and maintaining documentation.

- **Open and Honest Communications.** People need to perceive they are free, to offer suggestions. Open and honest communication enables people to make better decisions because the quality of the information that they are basing them on is more accurate.

2.5.2 eXtreme Programming

The eXtreme Programming (XP) [Beck00] is a software development process designed to manage projects in a simple and efficient manner. XP defines a whole of practices aimed to organize the development team's work. Those practices focus on software building, but do not cover the opportunity and feasibility studies. XP is one of the principal agile methodology, it is particularly well suited for small teams (lower than ten members) to develop at low cost.

User stories are similar to UML use cases and replace client specifications. They are written in natural language by clients. User stories allow the evaluation (by the designers) of the iteration's length and are then used to fulfill the tests.

An iteration starts by an iteration *release planning*, the meeting defines the stories to implement as well as the tests that should be accomplished. The planning is realized by the designers and the clients. The clients choose the scenarios to integrate in the next iteration and the designers evaluate iteration's length. The *spike solutions* are done by evaluating several potential solutions but are envisaged only for hard to solve problems. For each story, a planning is fixed: a story has to be implemented between one and three weeks, otherwise it has to be decomposed. A test represents a specific expected system behavior. A user story is validated when all unitary tests established for that story have been taken. Each developer takes a task responsibility (i.e. pass a test) and his pair (another person chosen by the developer and working with him as a pair) writes the corresponding unitary test. Developer's pair reorganizes/factorizes[2] the existing code to integrate the code during creation.

When the tests are successful for a particular story, the application's current version is updated with its last functionalities (*small releases*). Developer's pair integrates the last developments into the application by assuring that the non-regressing tests succeed.

XP also defines rules to associate to the process. Those rules are decomposed into four categories: planning (nine rules), coding (nine rules), conception (six rules) and tests (four rules). Those rules will not be detailed here but can be found in [Beck00].

[2] Factoring/Refactoring means improving the design of existing software code. Refactoring does not change the observable behavior of the software; it improves its internal structure. For example, if a programmer wants to add new functionality to a program, he may decide to refactor the program first to simplify the addition of new functionality in order to prevent software entropy (i.e. the tendency for software, over time, to become difficult and costly to maintain).

With exception to the user stories, XP does not produce documentation linked to analysis and design. The advantage of this method is its ease to put into practice, its low cost as well as its iterations high frequency and version delivery. Time is mostly spent to technical aspects: development and tests.

The method does however not cover the phases coming before and after development as users requirements capture, support and maintenance. The code refactoring step is not clearly explicated: what and how to factor? This approach is limited to little teams having a good agreement. Finally developers have to own enough skills to build evolutionary applications (otherwise they can waste their time restructuring the code). Most of the design decisions are taken by developers, another part are taken by the client when conceiving the user stories. Consequently, developers will often choose the easiest to implement solution.

2.6 Software Project Management

According to [RUP], "*Software Project Management is the art of balancing competing objectives, managing risks and overcoming constraints to successfully deliver a product which meets the needs of both customers (the payers of bills) and the users*".

Software Project Management can be cut into a series of disciplines, among those one can find:

- Risk management;
- Time management;
- Quality management.

Those disciplines will be studied into the next sections.

2.7 Risk Management

According to the Software Engineering Institute (SEI), "*Risk is the possibility of suffering loss*". A more precise definition is given by the [RUP], it considers risk as "*an ongoing or impending concern that has a significant probability of adversely affecting the success of major milestones*" while it defines success as "*meeting the entire set of all requirements and constraints held as project expectations by those in power*". Finally the IEEE Std 1540 defines the risk as "*the likelihood of an event, hazard, threat, or situation occurring and its undesirable consequences; a potential problem*".

Furthermore, risks can be categorized into direct and indirect risks. Direct risks are those on which the project has a large degree of control as for example wrong esti-

mates or poor quality. Indirect risks are those on which the project has little or no degree of control as loss of personnel or new competitors on the market.

According to [Kruchten03], the risk management process can be cut in a series of stages:

- Risk Identification consisting on the production and maintenance of a list of risks specific to the project;
- Risk Analysis consisting on the identification of what feature is affected by and what is the cause of the identified risks. It is also at that stage that a quantitative or qualitative value is allocated to risks, mostly in terms of impact and occurrence probability;
- Risk Prioritisation consisting on producing and diffusing a priority list for the identified risks;
- Risk Treatment consisting on defining the actions to take in order to deal with the risks.

Furthermore, as pointed out in [Kruchten03, RUP], risk management must be a continuous activity. The whole risks can seldom be identified once for all at the beginning of the project and new risks can be identified continuously. In the same way, the impact and likelihood of risks can evolve during the project so that they need to be re-evaluated continuously. Consequently, continuous effort must be devoted to risk identification and evaluation; this can be better achieved using an iterative life cycle.

2.8 Time Management

Facing time management the first feature to estimate is the size of the project. Units of measure can be of code size (number of lines of code) or of functional size (function points, use-case points, etc.). Once the size of the project has been estimated, tools to estimate the effort needed during the project phases are required. The *COnstructive COst MOdel* (*COCOMO*) existing in version 81 and II as well as *Software LIfecycle Management* (*SLIM*), are models allowing to estimate the effort required to complete a software project.

2.8.1 Sources Lines Of Code

The code size can be expressed as the amount of *Sources Lines Of Code* (*SLOC*), this approach offers an easy to measure and unambiguous method to evaluate the project. It also constitutes a simple indicator to monitor progress and a basis on which productivity can be estimated. Nevertheless the use of new technologies as genuine user interface builders, code generators, model-driven design, software patterns, code reuse, etc can seriously bias the SLOC estimations and the amount of work required.

2.8.2 Function Points

Function points were introduced by Allan Albrecht in 1979 (see [Albrecht79, AG83]), they provide an ordinal measure of software size independent from technology used. Function Point Analysis (FPA) is a structured technique of classifying components of a system. This method breaks systems into smaller components, so that they can be better understood and analyzed. It provides a structured technique for problem solving. In Function Point Analysis, systems are divided into five large classes and general system characteristics. The first three classes or components are *External Inputs*, *External Outputs* and *External Inquiries*. Each of these components transact against files that is why they are called *transactions*. The next two *Internal Logical Files* and *External Interface Files* are where data is stored that is combined to form logical information. The general system characteristics assess the general functionality of the system. The amount of SLOC per function point depends on the programming language used. Tables are widely available in literature.

2.8.3 Use Case Points

Use Case Points (see [SW01, ADSJ01]) is another method for project size and effort estimation using the project use cases as a basis for estimation. Use case based modelling is a well know and widely used technique to capture the software application's business processes and users requirements. Since they provide the functional scope of the application, analyzing their contents provides valuable insight into the effort and size needed to design and implement the application. Generally, applications with broad and complex use cases take more effort to design and implement than small applications with simpler use cases. Furthermore, the application development time is affected by:

- The amount of activities required to complete the use case (can be documented into activity diagrams);
- The amount and complexity of the involved actors;
- The technical implications of the use case realization (such as concurrency, security and performance);
- External factors such as developers experience and knowledge.

Gustav Karner defined in 1993 an estimation method that provides the ability to estimate an application's size and effort on the basis of use case called Use Case Points (UCP). This method analyzes the use case actors, scenarios as well as various technical and environmental factors and formalizes them into a mathematical equation. This equation is based on four variables, the *Technical Complexity Factor* (*TCF*), the *Environment Complexity Factor* (*ECF*), the Unadjusted Use Case Points (UUCP) and finally the *Productivity Factor* (*PF*). Those variables must be computed separately and integrated into the UCP equation:

$$UCP = TCP * ECF * UUCP * PF$$

2.8.4 COCOMO 81

COnstructive COst MOdel (*COCOMO*) an estimation model developed by Barry Boehm in 1981 (see [Boehm81]) used to determine the number of man-months required to develop a software project. COCOMO was designed on the basis of a sample of sixty projects at TWR, a Californian IT company. The model is a regression model i.e. on the basis or some identified variables, a regression on the database of the known projects was made and useful relationships on these variables were found and are in the model used to make predictions for other projects.

Three different modes exists in COCOMO depending on the type of project they are applied on:

- Organic: In this mode, relatively small software teams develop software in a highly familiar, in-house environment. It is suitable for small software projects (50 thousand delivered source instructions (KDSI)). People working on the project have extensive experience in working with related systems within the organization, and have a thorough understanding of how the system under development will contribute to the organizations objectives. Few constraints may appear during the software project;

- Semidetached: This mode of software development is suitable for medium sized projects (up to 300 KDSI), it is an intermediate mode. “Intermediate” may mean either of two things:
  - An intermediate level of project characteristic;
  - A mixture of the organic and embedded mode characteristics.

- Embedded: The embedded-mode deals with software projects operating within tight constraints. The product must operate within (is embedded in) a strongly coupled complex of hardware, software, regulations, and operational procedures, such as an electronic funds transfer system or an air traffic control system. Much innovation is also required and the project deals with complex interfaces.

Furthermore three levels of sophistications are depicted:

- Basic COCOMO is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code;

- Intermediate COCOMO computes software development effort as function of program size and a set of “cost drivers” that include subjective assessment of product, hardware personnel and project attributes;

- Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver’s impact on each step (analysis, design, etc.) of the software engineering process.

2.8.5 COCOMO II

The latest version of COCOMO called COCOMO II appeared in 1999 (see [Boehm00c]). The oldest version was not suited for the development life cycles used today; the latest version addresses these deficiencies by allowing dealing with iterative life cycles. COCOMO II also aimed to develop software cost database and tool support capabilities for continuous model improvement.

COCOMO II has three different models:

- The Application Composition Model is suitable for projects using modern GUI-builder tools for rapid application development;
- The Early Design Model is used to make rough estimates of a project's cost and duration before it is entire architecture is not determined. It uses a small set of new Cost Drivers, and new estimating equations.
- The Post-Architecture Model is the most detailed COCOMO II model. It is used after overall architecture has been decided. It has new cost drivers, new line counting rules, and new equations.

The main differences between COCOMO81 and COCOMO II are:

- The exponent value *b* in the effort equation is replaced with a variable value based on five scale factors rather then constants;
- Project size can be determined using object points, function points or source lines of code (SLOC);
- EAF is calculated from seventeen cost drivers better fitting today's methods, COCOMO81 only had fifteen;
- A breakage rating has been added to address volatility of system.

2.8.7 Iterative Planning

According to [Kruchten03], the Project Plan is "*a coarse-grained plan, and there is only one per development project. It captures the overall envelope of the project for one cycle*". Project planning includes three different aspects, the first is the definition of the major milestones (Lifecycle Objective, Lifecycle Architecture, Initial Operational Capability and Product Release), the second concerns the staffing profile, i.e. the resources required over time and third the dates of the minor milestones which occur at the end of each iteration.

2.8.7.1 Phases Length and Resources Allocation

In RUP based projects, iterations and phases' duration and HR allocation vary from one another. Figure 2-7 illustrates a typical project time and effort repartition

over a project. However no universal repartition exists; variations appear due to environmental factors, for example:

- if the project deals with many risks and a lot of uncertainty, the elaboration phase will be longer;
- if the project faces no risks and the software architecture is in place, the elaboration phase will be shorter;
- if the project outputs in a commercial project, the transition phase will be longer;
- if the project deals with certification issues, the transition phase will be very long;
- etc.

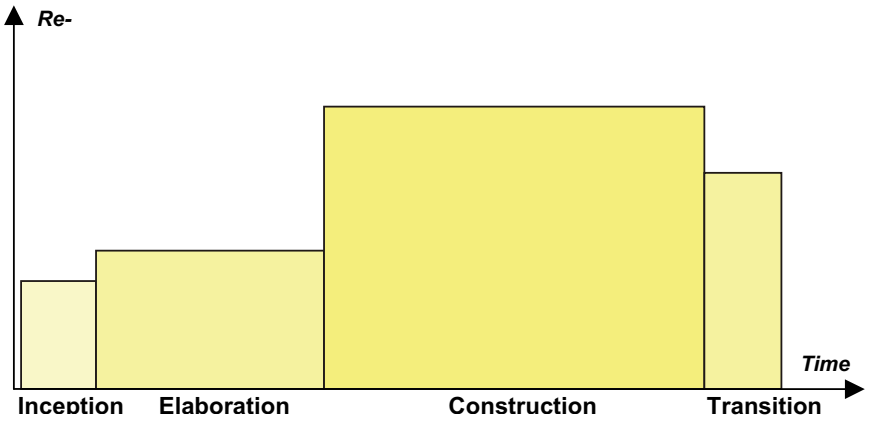


Figure 2-7. Typical RUP-based Project Effort Repartition (from [Kruchten03])

2.8.7.2 Amount of Iterations

Determining the number of iterations for the project is also an empirical issue. The number of iterations varies mostly in function of the project size but also in function of the staff experience with such processes. Each of the following phases may contain a variable number of iterations:

- The inception phase:
    - 0..1 iteration for a very simple project to produce a proof-of-concept prototype or user-interface mock-up;
    - 1 iteration for a more substantial project, to produce a prototype;
    - 1..2 iterations for a large project with many unknowns to eventually allow for more prototyping.
- The elaboration phase:
    - 1 iteration for a very simple project to produce a prototype;
    - 2 iterations for a more substantial project, the first to develop an architectural prototype and the second to finalize the architectural baseline;
    - 3 iterations for a large project with many unknowns, the additional iteration to allow different technologies to be explored, or to phase in the introduction of architectural solutions;
- The construction phase:
    - 1 iteration for a very simple project to build the product;
    - 2 iterations for a more substantial project, the first to expose a partial system and the second to mature it for beta test;
    - 3 iterations for a large project with many unknowns, the additional iteration because of the sheer size of the project;
- The transition phase:
    - 1 iteration for a very simple project to finish the product;
    - 1 iteration for a more substantial project to go from beta to full product release;
    - 2 iterations for a large project with many unknowns, the additional iteration to allow for operational feedback.

Summarizing, the number of iterations of a typical RUP project is 6 more or less 3 to 4 mostly in function of the project size. Most development cycles require six to eight iterations.

2.8.7.4 Iteration Plan

An *Iteration Plan* is a plan focusing on a particular iteration of the software project. Each iteration owns such a plan who defines the iteration scope, its evaluations criteria the activities performed in each discipline of the iteration and finally the human resources responsibilities. Two iterations are usually active at any time of the project:

- The current iteration plan to track progress for the iteration that is being performed;
- The next iteration build toward the second half of the current iteration and is ready at the end of it.

The project manager will consequently be writing the next iteration plan while the software development team is fulfilling the current iteration plan. The second plan can evolve later in the iteration so that the software engineering team receives a viable plan for the subsequent iteration. Typically, an iteration plan encompasses the following properties:

- The *Iteration Scope* is the intent, what will be accomplished during the iteration. During the Inception and Elaboration phases, the iteration objectives will principally be risk-driven, in the sense that risks will determine which use cases, scenarios, algorithms, and so on will be developed during the iteration. During the Construction and Transition phases, the objectives will be determined by completion objectives, and achievement of a set level of quality;
- The *Iteration Evaluation Criteria* determines how to objectively evaluate the accomplishments at the end of the iteration, it specifies the artifacts (or *WorkProducts*) that will be worked on;
- The *Iteration Activities* establish what works need to be done and on which artifacts. A generic description of the activities that can be performed during a discipline is available for software development processes as the RUP, these can be instantiated on each iteration of each project. The activities performed during each phase are planned here;
- The *Roles Assignation* aims to assign to each human resource to furnish boundaries to execute the activities.

2.9 Quality Management

According to [Pressman04], "*software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance)*".

Quality of design refers to the characteristics specified by designers for a specific item. Performance metrics, ergonomic issues, tolerance to fault are factors involved into the quality of design. Those can be improved using more sophisticated techniques raising the development cost higher.

Quality of conformance represents the degree to which design specifications are followed during manufacturing. The greater this degree is, the higher the quality of conformance level is.

Facing the software development stages, quality of the design encompasses requirements, specifications and system design while quality of conformance mostly focuses on implementation issues. When the implementation adequately follows the design then the resulting system meets its requirements and performance objectives, then is the quality of conformance high.

Another more intuitive measure of software quality through user satisfaction has been proposed in [Glass98], which is:

User satisfaction = compliant product + good quality
+ delivery within budged and schedule.

2.9.1 Quality Control

Quality control represents the means allowing measuring (quantitatively ou qualitatively) quality. [Pressman04] quotes that quality control "*involves the series of inspections, reviews and tests used throughout the software process to ensure each work product meets the requirements placed upon it.*" It includes the feedback loop to the process that created the work product. Measurement and feedback allow adapting the process if the work products created do not meet its specifications. Quality control is a part of the manufacturing process.

2.9.2 Quality Assurance

According to [PM91], "*quality assurance provides management with the necessary data to be informed about product quality, to gain insight and confidence that quality goals will be met*". It also includes the identification of problems affecting quality, if some are detected, management should take measures.

2.9.3 Quality Control Management

To better understand the activities of companies in a distributed network it is useful to separate the business systems into a series of value generating processes referred to as the value chain. The idea of the value chain is based on the process view of organisations, the idea of seeing a manufacturing (or service) organisation as a system, made up of subsystems each with inputs, transformation processes and outputs. Inputs, transformation processes, and outputs involve the acquisition and consumption of resources - money, labour, materials, equipment, buildings, land, administration and management. Since every process with its inputs and outputs contributes

to the added value, each process can be considered as a value chain process within the distributed network. This approach, using the idea of a value chain, provides a transparent insight into needed resources of a network. Tracing, monitoring of resources and controlling of performance in a network become easier since costs and value drivers are identified for each value activity and its ultimate goal is to maximize value creation while minimizing costs.

The QCM approach supports an over all performance measurement for components with value adding processes, path recording, tracing of progress and information. The following lines describe the main functionalities of QCM to be provided in order to ensure performance measurement in distributed networks. To generally enable the tracing of a distributed network progress (value chain progress) and performance within a distributed production and collaboration network there is a need for a method to generalize a project, transform and map it into a process based model. To provide standardization and common vocabulary in distributed networks this approach uses a pool of micro processes which are able to cover all possible end users activities in the network. To support this modeling for each participating partner of any kind of project with network character a pool of micro processes adapted from a process reference model has been integrated as the core of the approach. This allows the description of any partner's contribution to a project on a process basis, as a collaboration project consists of processes and each process is a sequence of partner's activities. As most reference models are covering supply chains this approach goes further and considers also research, design, supply and customer services. Therefore it is foreseen to base the approach on a pool of standardised micro processes which are more service oriented than manufacturing driven.

2.10 Chapter Summary

For as long as complex software systems have been developed, the software engineering field has arisen. At early stages the need to decompose software production into separate disciplines appeared, that is why software production is traditionally divided into an analysis stage to capture the application requirements, a design stage to translate user requirements into a software design and an implementation stage to implement the software in a given computer language. Empirical experience revealed the need for testing the developed software solution. Unfortunately when proceeding sequentially, testing is left for the latest stages of the project when corrective measures are hard to establish. That is why software life cycles progressively evolved to incremental, iterative and finally spiral life cycles allowing a more realistic approach to problems involving users in an intensive manner. Nowadays such models have been included into broader methodologies as the Rational Unified Process or eXtreme Programming.

Using those modern development methodologies is not trivial and the need for support activities regrouped by the term software project management. Among those activities risk management is one of the most fundamental. Indeed, RUP and XP where fundamentally created for reducing the risks taken into the software project.

Time management allows project managers to evaluate the overall amount of work that should be performed onto the project, allocate resources and create a detailed iteration planning. Finally, quality management is the field where quality benchmarks are set, quantified and cyclically evaluated. Having overviewed the general software engineering specificities, we can turn on to the epistemological foundations in Chapter 3.

References

[Albrecht79] A.J. Albrecht. “Measuring application development productivity”. In *Proceedings of the IBM Applications Development Joint SHARE/GUIDE/IBM Symposium*, CA, USA, pages 83-92, 1979.

[AG83] A.J. Albrecht and J.E. Gaffney. “Software function, source lines code, and development effort prediction: a software science validation”. *IEEE Transaction of Software Engineering*, 9(6):639-648, 1983.

[Ambler02] S.W. Ambler, “Agile Modelling – Effective Practices for extreme programming and the unified process”, *John Wiley & Sons*, 2002.

[ANV05] S.W. Ambler, J. Nalbone and M. J. Vizdos, “The Enterprise Unified Process: Extending the Rational Unified Process”, *Prentice Hall PTR*, 2005.

[ADSJ01] B. Anda, H. Dreiem, D.I.K Sjoberg, M. Jorgensen. “Estimation Software Development Effort Based on Use Cases-Experiences from Industry”. In *Proceedings of the Fourth Internation Conference on the Unified Modeling Language*. Toronto, Canada, 2001.

[AN02] J. Arlow and I. Neustadt. “UML and the Unified Process”. *The Object Technology Series*, Addison Wesley, 2002.

[Beck00]: K. Beck. “Extreme Programming Explained: Embrace Change”. *Addison-Wesley*, 2000.

[Boehm81] B. Boehm. “Software Engineering Economics”. Prentice Hall, 1981.

[Boehm88] B. Boehm. “A Spiral Model of Software Development and Enhancement”. *Computer*, May 1988, pp. 61-72.

[Boehm94] B. Boehm. “A Collaborative Spiral Software Process Model Based on Theory W”. *IEEE CS Press*, Los Alamitos, Calif., 1994, pp. 59-68.

[Boehm96] B. Boehm. “Anchoring the Software Process”. *IEEE Software*, Volume 13, No 4, pp. 73-82, July 1996.

[Boehm98] B. Boehm. “Using the WINWIN Spiral Model: A Case Study”. *Computer,* vol. 31, no. 7, pp. 33–44, July 1998.

[Boehm00a] B. Boehm edited by Wilfred J. Hansen. “Spiral Development: Experience, Principles, and Refinements”. *Carnegie Mellon University*, *Special Report CMU/SEI-00-SR-08*, *ESC-SR-00-08*, June 2000.

[Boehm00b] B. Boehm. “Requirements that Handle IKIWISI, COTS, and Rapid Change”. *Computer,* IEEE, pp. 99-102, July 2000.

[Boehm00c] B. Boehm et al. “Software cost estimation with COCOMO II”. *Prentice Hall*, 2000.

[BRJ99] G. Booch, J. Rumbaugh and I. Jacobson. "The Unified Modeling Language User Guide". *Addison-Wesley Object Technology Series*, 1999.

[BPV94] M. Bradac, D. Perry, and L. Votta. "Prototyping a Process Monitoring Experiment". *IEEE Trans. Software Engineering,* vol. SE-20, no. 10, pp. 774–784, October 1994.

[Dorfman97] M. Dorfman. "Requirements Engineering". In *R. H. Thayer and M. Dorfman (eds) "Software Requirements Engineering, Second Edition"*, *IEEE Computer Society Press*, pp 7-22, 1997.

[Forsberg97] K. Forsberg and H. Mooz. "System engineering overview". In *R. H. Thayer and M. Dorfman (eds.) "Software Requirements Engineering, Second Edition"*, *IEEE Computer Society Press*, 1997, pp.44-72.

[Glass98] R. L. Glass, "Defining Quality Intuitively". *IEEE Software*, vol. 15, no. 3, pp. 103-104,107, 1998.

[Hanna95] M. Hanna. "Farewell to Waterfalls". *Software Magazine,* pp.38–46, May 1995.

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh. "The Unified Software Development Process". Addision-Wesley, 1999.

[Kruchten03] P. Kruchten. "The Rational Unified Process: An Introduction". *Addison-Wesley*, 2003.

[Leroy98] R. Leroy Burback, "Software Engineering Methodology: The Watersluice", *PhD thesis*, *Stanford University*, *CompSci department*, USA, 1998.

[MDE93] J McDermid. and P. Rook. "Software Development Process Models". In *Software Engineer's Reference Book, CRC Press*, pp. 15/26–15/28, 1993.

[Pressman01] R. S. Pressman. "Software Engineering, A practitioner's approach" – Fifth edition. McGraw Hill, 2001.

[Pressman04] R. S. Pressman. "Software Engineering, A practitioner's approach" – Sixth edition. McGraw Hill, 2004.

[PM91] L. Putnam and W. Myers. "Measures for Excellence: Reliable Software on Time, Within Budget". Prentice Hall, 1991.

[Royce70] W. Royce. "Managing the Development of Large Software Systems". In *Proceedings of the IEEE WESCON*, pp. 1-9, August 1970.

[Royce98] W. Royce. "Software Project Management. A Unified Framework". *Addison-Wesley*, 1998.

[RBJ99] J. Rumbaugh, G. Booch, and I. Jacobson. "The Unified Modeling Language Reference Manual". *Addison-Wesley Object Technology Series*, 1999.

[RUP] IBM. "The Rational Unified Process. Version 2003.06.00.65". *Rational Software Corporation*, 2003.

[SW01] G. Schneider and J. Winters. “Applying Use Cases - A Practical Guide” – Second Edition. *Addison Wesley*, Boston, 2001.

[UML] OMG. “OMG Unified Modeling Language Specification. Version 2.1.2”. November 2007.

[Weyns04] D. Weyns, A. Helleboogh, E. Steegmans, T. De Wolf, K. Mertens, B. Boucké and T. Holvoet. “Agents are not part of the problem, agents can solve the problem”. In *Proceedings of the OOPSLA Workshop on Agent-Oriented Methodologies*, 2004.

3 Epistemological Foundations

Information systems are deeply linked to human activities. Unfortunately, development methodologies have been traditionally inspired by programming concepts and not by organizational and human ones. This leads to ontological and semantic gaps between the systems and their environments. The adoption of agent orientation and Multi-Agent Systems (MAS) helps to reduce these gaps by offering modeling tools based on organizational concepts (actors, agents, goals, objectives, responsibilities, social dependencies, etc.) as fundamentals to conceive systems through all the development process. Moreover, software development is becoming increasingly complex. Stakeholders' expectations are growing higher while the development agendas have to be as short as possible. Project managers, business analysts and software developers need adequate processes and models to specify the organizational context, capture requirements and build efficient and flexible systems.

In this chapter we propose a modern epistemological reading of SE as an evolving science as well as software project knowledge evolution; project actors' reasoning context is also studied. Popperian theories have been used in literature to describe the evolution of knowledge within a software project (at micro level). Due to the nature of SE belonging as well to social sciences we will show that this vision is abusive at both micro level (knowledge within the project) and macro level (knowledge on development frameworks). The Lakatosian falsification principle will be used to demonstrate the complexity of getting a crucial experiment in SE. Actors involved in a software project have a limited vision inherent to their bounded rationality. They can subsequently learn more about users' expectations by proceeding iteratively. This will be shown into this chapter.

We also propose a modern epistemological validation of the emergence of Object-Orientation (OO) and Agent-Orientation (AO). The latter will be put into perspective through the Lakatosian approach. Related work and contributions to the epistemological position of OO and AO will first be explicated. The emerging context of the conceptual frameworks of OO and AO, the software crisis, is then briefly described. The validation of our epistemological reading will be done on the basis of OO and AO operationalization of some critical theoretical concepts derived from the Kuhnian and Lakatosian theories. We finally discuss the adoption of the Lakatosian research programme concept to characterize both OO and AO. Implications of this epistemological position on everyday work have been distinguished both for software engineering researchers and practitioners. For researchers, it mostly has an implication on how agent ontologies are built and for practitioners it has an implication on how software problems are envisaged.

This chapter is organized as follows. Section 3.1 presents the context of the study. It encompasses the main contributions as well as some words about the software crisis. Section 3.2 envisages software engineering as a social science, knowledge

evolution is studied in Poperian terms and nuanced in Lakatosian ones. Simon's bounded rationality principle is also used to describe the project actors' rationale. In section 3.3, we point out the emergence of OO and AO as a consequence of the software crisis. AO is successively considered as a paradigm and a research programme. On the basis of how some relevant concepts of the Kuhnian and Lakatosian frameworks are operationalized by OO and AO, we provide a Lakatosian reading of those modeling concepts. The chapter is summarized in Section 3.4.

3.1 Context

This section presents the context of the research. Main contributions of an epistemological reading for computer science researchers are exposed as well as the software crisis.

3.1.1 Related Work and Contributions

[Basili92] defines Software Engineering (SE) as "*the disciplined development and evolution of software systems based upon a set of principles, technologies and processes*". These theoretical frameworks are expected to solve practical problems by proposing software solutions. SE is a practice-oriented field (where empiricism often plays an important role) and constantly evolving; however, one must dispose of a framework to build common (and preferably best) practices improvement. Kaisler [Kaisler05] points out that "*We develop more experience, we not only continue to learn new practices, but we refine and hone the practices that we have already learned*". SE is the genuine discipline that emerged from this interconnection between practices and software solutions. Today's software development has become a very complex task and no one has the required skills or time to resolve a sophisticated problem on his or her own. Software development phases need the input from lots of people having to use concepts and ideas for which they share a common understanding. This can be referred as SE's key role: providing some common theoretical entities to allow specialists to develop software solutions.

Few papers in specialized literature point to an in depth questioning of SE knowledge evolution. As [Kaisler05] emphasizes, the literature is mainly technical or practical and focused on the software design processes. Research methodologies, however, need to be conscientiously built to favour the development and improvement of software solutions. To this end, an epistemological analysis is of primary importance as pointed out by [Basili92]: "*The goal is to develop the conceptual scientific foundations of software engineering upon which future researchers can build*".

In this chapter, we mostly focus on two specific aspects of SE: knowledge evolution in SE at both micro (knowledge into a software project) and macro levels (knowledge about software engineering frameworks) as well as the evolution from object to agent orientation.

3.1.1.1 The Evolution of Software Development Life Cycles

The study of the evolution of knowledge in SE tends to demonstrate that the roots of software development especially those involving users intensively (where requirements engineering and organizational modelling play a dominant role) can be found in social sciences rather than in exact ones with a profound consequence on the most adequate approach during the requirements elicitation process. The proof of concept is based on the study of knowledge and knowledge evolution of software professionals and users during the requirements elicitation process. This will allow us to show that an iterative SDLC is more adapted than a traditional waterfall one by studying the epistemological foundations of software development processes with Herbert Simon's bounded rationality principle.

According to [TD99], the iterative nature of the requirements elicitation process is based on Simon's *bounded rationality* principle and on Karl Popper *knowledge theory*. Our study takes those two concepts as a first basis, we however propose a more profound study based on an enriched epistemological literature, specifically we envisage knowledge in SE at the light of:

- bounded rationality but also perfect rationality and the Lakatosian adaptive rationality;
- the poperian knowledge growth model but criticize it on the basis of the falsification principle.

3.1.1.2 From Object-Orientation to Agent-Orientation

OO and AO which are modelling and programming ontologies rather than development processes. A few papers have discussed the evolution of knowledge in SE but encompass a broader range of aspects of this discipline than the current paper.

In the book titled *Software Paradigms* (see [Kaisler05]) published in 2005, Kaisler uses the notion of a paradigm to characterize the way of solving problems in software development. Though he explicitly quotes Kuhn's work to define the concept of paradigm, he explains that he includes in this definition the "*concepts of law, theory, application and instrumentation: in, effect, both the theoretical and the practical*". The practical dimension being a key issue in computer science, it must be integrated into the paradigm when this concept is used in engineering software. Kaisler's work is based on an empirical process: *"We are going to apply the notion of paradigm to the investigation of programming languages and software architectures to determine how well we can solve different types of problems*". On the basis of this study, he proposes a problem typology which would determine the programming approach: *"For a given problem class, we'd like to be able to create software that solves the problem or parts of it efficiently. There are two aspects to creating such software: the software's architecture and the choice of programming language*." Kaisler uses the term paradigm in a very large sense since he applies the concept to the whole "top-down analysis." The problem typology defines a software typology which finally determines a specific implementation.

Even if we do not assign the same meaning as Kaisler to the paradigm concept, our analysis can be related to his work. By proposing an epistemological reading of the evolution toward AO, we only focus on the last step of Kaisler's view. Indeed, the emergence of AO can be seen as a broadening of the unit of implementation and consequently to the amount of problems that can be solved. This is in line with the vision developed in [JW01]. AO is the best suited to solve complex problems because it refers to a higher abstraction level and it provides some advantages for solution programming. Jennings and Wooldridge use the term "paradigm" to characterize the evolution from OO to AO. They paradoxically emphasize the continuity between these two paradigms. Indeed, as far as the Kuhnian discussion is concerned, if two theoretical frameworks can be compared they cannot be quoted as paradigms; this will be explained in section 3.4.

Finally, [GA04] points out that: "*One of the most recent (and widely accepted) examples to a "rescuer new paradigm" in software engineering is the object-orientation paradigm,…*" and recalls that the evolution from OO toward AO is often presented in terms of paradigm shift [JW01, Basili92, WJ95]. A very important point discussed by Göktürk et al. [GA04] is that the choice of a paradigm is similar to the choice of a conceptualization/communication language; consequently mixing two different paradigms could be counter productive. What Göktürk et al. emphasize indirectly is what epistemologists call the "incommensurability thesis". Following this line of argument, two paradigms are totally incomparable since they represent two different set of knowledge.

Based on these related works, our contribution is multiple:

- We emphasize on the epistemological reasons why the paradigm concept is inappropriate to explain the differences between OO and AO. To be considered as paradigm, two theoretical frameworks need to be incomparable and "incommensurable";

- We propose a new epistemological analysis of the evolution toward AO by using a Lakatosian framework which can explain and justify the effectiveness of this framework. In this paper, AO will be presented as a new research programme widening and improving the knowledge in SE. This improvement cannot be seen as discontinuity of knowledge but rather as continuity since AO could encapsulate object technology;

- By proposing an epistemological reading of the emergence of the AO, we try to reduce what Jennings and Woolridge [JW01] call the "gap" between knowledge and applications. We prove that this evolution can be explained in line with the conventional standards to justify the scientific status of the SE discipline. The paradigm-concept cannot justify the scientific status whereas the research programme-concept can.

3.1.2 The Software Crisis

Although constant progress is being made in software development, complex information systems often imperfectly match users' requirements. A "software engineering crisis" diagnosis was first made in the late sixties. Considerable work has been done in conceptual modeling and other engineering methodologies and processes so that the crisis can be mitigated. However, the diagnosis still remains partially to date. Indeed, structured methods are not systematically used in software projects and computers expect responsible users, but most of the time the problems encountered are the result of human decisions or methodological insufficiencies.

Software engineering (SE) can be defined as the methodological processes and the relevant artifacts used for the development of software on a large scale. The software engineering crisis has different reasons:

- the increasing complexity of modern software: continuous technological progress allows us to develop larger and larger applications while the increasing demand exceeds the productivity gains obtained by methodological, techniques and tool improvements;

- the common under-estimation of the difficulty and consequently the software development cost (the methods, widely empiric, of individual programming are not applicable to the development of large and complex systems), conflicting software development process and inadequate software products;

- lack of reliability and maturity of software compared to hardware and its relative importance in the realization of the complex system functions;

- substantial delay in the diffusion of best practices within the software industry (see [Davis95, Meyer97, Sommerville01]) due to the complexity and cost of education of software developers in a constantly evolving discipline;

- inherent complexity:

    - the problems that a software has to deal with can be arbitrarily complex; for example, the limits of system functionalities are often much less clear than those of tangible products;

    - the sequential decomposition of the development phases is not so natural for SE than it is for other engineering disciplines (mechanical engineering for example).

We argue that advances in SE conceptual modeling such as OO and AO are part of the effort to address the crisis; this point becomes important in the context of the epistemological reading, as discussed in this chapter.

3.2 Software Engineering: Lakatosian Reading of a Social Science

In this section we will present the Popperian knowledge growth model and nuance its application to SE with the Lakatosian falsification principle.

3.2.1 The Knowledge Growth Theory

According to Popper [Popper72], scientific knowledge evolves thanks to a continuous and infinite problem resolution process. Knowledge increases by confronting theories with reality. This confrontation, called *falsification* by Popper himself, often generates new problems that theoreticians have to define, theorize and solve. The empirical refutation of a proposed theory provides requiring new theories subject to empirical refutation providing other problems and so on. The knowledge growth process is ad infinitum because, according to Popper, the resolution of new problems requires new knowledge (i.e. new theories and new conceptual tools). Popper calls "crucial experiment" [Popper59] any theory refutation allowing the emergence of new problems. This refutation constitutes a real source of inspiration for the development of new theories supposed to solve the new problems.

According to [TD99], the Popperian knowledge growth model can be applied to iterative software development. Indeed, by disturbing the established relations in a particular organizational context, each software solution generates new problems which are specific to this organization. It is impossible to predict exactly how a new software solution will disturb the organizational context. This context is often very complex because a lot of interactions between its components exist and, moreover, an iterative process leads to a moving evolution of this context. At first time, the stakeholders, by using a software solution, modify their needs to solve the new problems created by this system and then improve their use of the system. Stakeholders requirements cannot be fully described and specified before the implementation because they evolve when they approach the software system. Consequently their requirements evolve iteratively during the software development life cycle. [TD99] explains that this evolution can be characterized by a Popperian lecture. In the following section, the use of the Popperian knowledge growth theory applied to software development will be reconsidered.

3.2.2 Falsification Principle: a Critical View

In his work about the mathematical formalisms, Irmé Lakatos proposes two categories of theories: Euclidean and quasi-empirical theories. Whereas the first are connected with an axiomatic approach (imposing a purely deductive logic from a priori axioms), the seconds are determined by the empiricism because they evolve through a continuous feed-back between the theory and its empirical results. As we will show in this section, this point of view is different from the Popperian one because in his epistemology, theoreticians have to change the whole theory if it is falsified. We think that the Lakatosian epistemology can be useful in SE because the process of theories adjustment he proposed is directly in line with the quasi-empirical logic observed in software development.

3. Epistemological foundations

The aim of this section is to moderate the use of Popperian epistemology in SE with respect to what we exposed in the previous one. At first sight, the Popperian knowledge growth model is very close to what Lakatos calls the "quasi-empirical theories". However, the two visions are different and, according to us, only Lakatos is adequate for an epistemological analysis of SE. We present our arguments in this section.

Popper developed his epistemology for "hard sciences", especially for physics. Science is mainly considered in terms of "representation of the world" rather than in terms of "efficient answers" as it would be in the field of applied sciences. This detail is of primary importance since, at the opposite of physical sciences, an universal law true for every software development cannot exist. As will be shown in the second part of this study by using the bounded rationality principle, there is no universal and optimal software solution: no development is identical to one another since each organization is different. The practical experiment supposed "to falsify" the developed solution appears to be extremely complex and multiple. In this context, it is impossible to dispose of what Popper calls a "crucial experiment". Indeed, in software development, each new solution is a new experiment based on various theoretical concepts and trying to satisfy various requirements. Each software solution is developed for a particular organization employing various people pursuing various goals and it is very difficult to compare with one another so that the reasons of an empirical failure are hard to determine. Moreover, in software development it is very hard to correctly identify the nature of the error [Priestley05]. The reason(s) of an empirical failure can be conceptual or result from a bad representation of the organizational logic but it could simply be the consequence of a programming mistake or to a technical problem in the physical layer. The variety of errors strongly complicates the idea of a "crucial experiment". As we mentioned earlier, software development is not a pure empirical science but rather a "quasi empirical applied discipline", therefore, we cannot, in case of empirical failure, reject all the applied SE theoretical bases (like requirements elicitation process used during the project). Theoretical tools can be very powerful and particularly well indicated in the context of the "failed" development but the reason of failure is exterior to them. In the same way and on a micro-basis we cannot in case of empirical failure reject the identified requirements. Indeed, as exposed earlier in this dissertation, requirements appear in an iterative manner during the project life cycle. In case of empirical failure of a software solution the identified requirements cannot be simply rejected. They can be relevant and accurately specified but the reason of the failure is exterior to them.

Consequently the falsification principle specific to the Popperian epistemology cannot be applied in the strict sense to software engineering, a quasi-empirical discipline as it would be to hard sciences ones. This subscribes the fact that the nature of SE can also be found in social sciences rather than only in hard ones.

For sure, the empirical failure of a software solution has to lead to calling into question the SE theoretical foundations in general and the identified requirements in particular but they cannot be directly and purely rejected. The Popperian model of knowledge (also called "dogmatic falsificationnism" in philosophy of science) ap-

pears to be too radical for an epistemological reading of SE. This Popperian radicality has already been recognized and caused a plentiful literature in philosophy of science (see e.g., [Kuhn96] or [Lakatos77]). In line with the opposition to the radicalism of falsification, Lakatos developed its methodology of "research programs" (see [Lakatos77] for the theoretical basis and section 3.4 for an application on Object and Agent Orientation).

The Lakatosian epistemology (often "called methodological falsificationnism") is less radical than the Popperian one. Indeed, Lakatos tries to explain why scientific theories continue to exist in spite of the existence of empirical refutations. Even if empiricism plays a very important role in the evolution of the knowledge, Lakatos recognizes that all the theories born, are refuted and die refuted. The Hungarian philosopher strongly criticizes the concept of "crucial experiment" and argues it is impossible to explain the evolution of science only in Popperian terms. Lakatos recognizes the importance of empiricism since it is precisely through the confrontation to the real world that sciences evolve, however, the philosopher explains why the "ad hoc"[1] characteristic often observed in empirical studies can be useful to improve the theory. In the Lakatosian vision of science, it is not necessary to change the whole theory in case of refutation. The empirical failure just represents a sign of a problem which must be solved by improving the tested theory. However, as long as the tested theory does not face a continuous empirical failure (which would mark the degenerative character of the research program in which the theory has been developed), nothing obliges the theorists to modify their theoretical framework[2]. This vision is particularly well adapted to an epistemological reading of SE.

In this section, we showed that the Lakatosian epistemology would be better adapted than the Popperian one to characterize the knowledge evolution in SE. Indeed, the Lakatosian epistemology, by rejecting the concept of "crucial experiment" is less radical with the importance of the empiricism and allows a feed-back process between the empirical and the theoretical levels. Moreover, as shown previously, the "quasi-empirical" character of software development and the complexity of organizational entities require flexibility in the interpretation of problems encountered during the implementation of a software solution.

3.2.3 Bounded Rationality

Herbert Simon (Nobel Prize in economics, 1978) proposes the concept of *bounded rationality* [Simon83] to characterize the human reasoning and behavior in uncertain situations. This concept illustrates the idea that human beings have limited abilities to

[1] *Ad hoc* is a latin expression which means "*for this purpose*". In philosophy of science *ad hoc* often means the addition of corollary hypotheses to a scientific theory to prevent this theory from being falsified by anomalies not anticipated by its initial theory. Popper rejects this "ad hoc" character in science and, following him, this should be considered as a kind of intellectual dishonesty (see [Popper59]).

[2] [Lakatos77] reminds that science is a human practice and that it is sometimes very difficult for scientists to reject a theoretical framework when they have devoted several years of their life working on and developing it.

analyze all the parameters of the solutions they face. According to Simon, the rationality of human beings is related to the present and to the psychological biases. The rationality suggested by Simon is much broader than the perfect rationality which would allow one to be able to evaluate all the present and future parameters of a particular solution. This idea of perfect rationality is often used in economics and made Simon win a Nobel Prize for his work.

This vision of rationality can be used to characterize an organization behavior as a software engineering team. Indeed, given the uncertainty and the complexity of the real world and the technological progress, there is no universal and optimal software solution which would be rationally superior to the other possible solutions. Because software engineering is a human practice, the perfect software solution driven by a perfect rationality does not exist. The illustration on Figure 3-1 shows the multiple actors' visions of a software product. When building a software solution several actors' (bounded) rationalities intervene, for example:

- when modeling the organizational setting:
    - stakeholders (mainly engineers) explain their vision of the organization and other business processes;
    - software modelers represent their understanding of those in the form of models containing their own limitations.
- when eliciting and modeling user requirements:
    - users explain their vision of what the system should do;
    - software modelers represent their understanding of those in the form of models containing their own limitations.
- when designing the software application, software designers represent their understanding of the models and other artifacts produced by analysts building design models containing their own limitations.
- when implementing the system, developers implement the system on the basis of their understanding of the design models and other artifacts.

Each actor involved in the software development process has to deal with his own rationality which is by essence bounded.

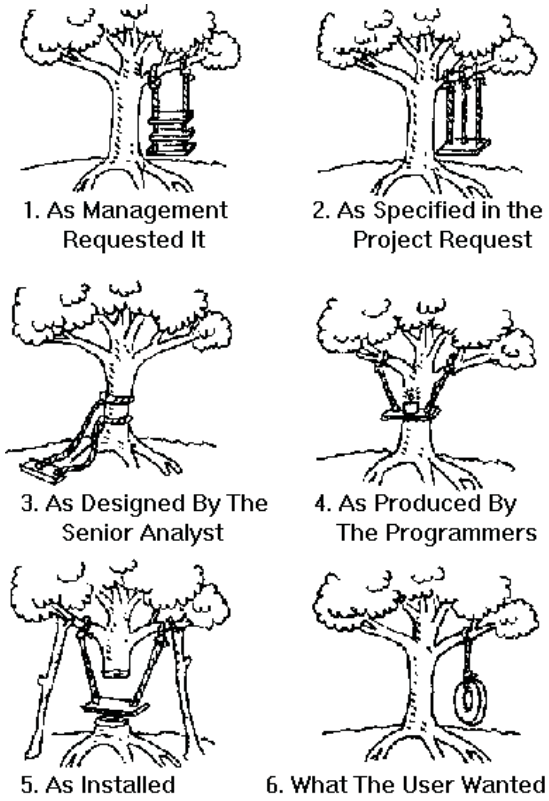


Figure 3-1. a Multi Vision Representation of Software Development.

The bounded rationality principle is in accordance with the rationality used in social sciences. This concept has allowed the emergence of behavioral economics which tend to be more and more dominant[3]. Using such a vision implies to take into account that human beings cannot evaluate perfectly the moving contexts and that this misunderstanding of the situation leads them to seek a *satisfactory solution*. This satisfactory solution depends directly on each individual and the quest for a satisfactory solution allows people to meet their fundamental needs and to develop themselves. In this human reasoning model, no way exists to determine if this "satisfactory solution" maximizes a specific utility function or to know if another choice would have provided an higher level of satisfaction.

[3] Daniel Kahneman psychology professor at Princeton won, in 2002, the Nobel Price in economics for having integrated insights from psychological research into economic science, especially concerning human judgment and decision-making under uncertainty (see [TK79, Kahneman02]).

The concept of perfect rationality is extremely useful to solve some theoretical problems [Arthur94]. Perfect rationality presupposes that individuals have much higher computational abilities than they have in the real world. Indeed, beyond a certain level of complexity, the logical abilities of people are not sufficient to evaluate the situation. Moreover, in complex situations implying interactions between people, an individual, who would have perfect rationality, would have to guess the behavior of other people (leads by bounded rationality). So human behavior is always determined by bounded rationality and this rationality does not depend only on the ability to evaluate a situation but also on the interactions with other (non-perfectly rational) people. Let us remind that human relations are partly driven by subjective beliefs which complicate the rational evaluation of a particular situation. We could say that the complex interactive situations can be seen as undefined problems. To face such problems, Arthur [Arthur94], in accordance with the behavioral economics (and with the bounded rationality principle), proposes to think the human mind as a problems simplification machine. This simplification is possible thanks to internal patterns developed by people. Each actor has a collection of internal beliefs which materialize in a particular evolving behavior. Indeed, actors improve their behaviors and learn which of their internal belief is adapted to face a complicated and undefined situation. We can then observe a kind of iterative improvement of human behavior. The principle of bounded rationality results from a particular *conservative incrementalism* materialized in a careful attitude always concerned by feedbacks provided by human interactions [KK82].

The principle of bounded rationality is an interesting concept for the theoretical analysis of software development. It allows characterizing the complexity observed in software solutions to give an efficient response to organizational problems by nature hard to define. Organizational interactions allow project stakeholders to improve their knowledge in an iterative manner and to improve their requirements comprehension. When using iterative, incremental or spiral software development requirements are collected and revised progressively on the basis of early testing to get the most advanced vision of stakeholders rationality so that users' perceived quality of the software solution can be improved.

3.2.4 Implications

Everyday work implications of the study of knowledge evolution in SE as well as the bounded rationality principle applied to SE processes gave several interesting conclusions important in the daily practice of researchers and other practicians.

The most important aspect that can be deduced from our work is that SE does not belong to hard sciences but can rather be considered as a human discipline. This conclusion is mostly based on two epistemological lectures:

- the Lakatosian principles for a more adequate study of the evolution of knowledge into the SE processes than traditional Popperian ones;

- the bounded rationality principle applied to the process itself depicting the involved actors rationale.

At the light of those two lectures, we claim for the adoption of SDLCs adopting iterative principles because:

- knowledge grows within human organizations. When developing software solutions, knowledge about users requirements and further expectations grow during the project life cycle and previously identified requirements should not be refuted but rather refined continuously at the light of users feedbacks;

- the ability to express requirements and to understand their nature is by essence limited since actors are dealing with bounded rationality so that this process cannot be proceeded once for all into the software project but should rather be continuous.

Thanks to its inherent feedback loop, iterative development allows to better deal with the knowledge evolution into the software project and with actors' bounded rationality.

Furthermore, due to the fact that with a Lakatosian reading of SE knowledge evolution no crucial experiment can reasonably be established, empirical testing of the developed SE theories should be realized on case studies but the quantitative burden of the proof of their better performance can hardly be established. This consequence is notably useful for researchers working on the elaboration of new SE methodologies.

3.3 A Modern Epistemological Reading of Agent Orientation

We argue that a Lakatosian vision should be adapted for an epistemological reading of the emergence of AO. To develop our argumentation, we first consider its use in the literature dealing with the "paradigm" concept to characterize the evolution from OO to AO. We then explain why the Lakatosian "research programme" concept is more adequate than the Kuhnian "paradigm" concept to illustrate the evolution of modeling and programming concepts.

3.3.1 The Traditional Kuhnian Perspective of Software Engineering

This section defines the paradigm concept following the Kuhnian epistemology and envisages SE as an evolution from one paradigm to another.

3.3.1.1 "Paradigm" Concept: a Definition

The word "paradigm" comes directly from philosophy where its meaning remains surprisingly rather vague. Plato and Aristotle were the first authors to introduce this concept. According to them, the paradigm is a kind of explanatory model, which

allows people to understand, in terms of causality, the changes imposed by Nature. However, the paradigm is not, strictly speaking, a logic. For Aristotle, the "paradigm" was "different from both deduction, which goes from universal to particular, and induction, which goes from particular to universal, in the sense that the paradigm goes from particular to particular." [GA04].

The term "paradigm" has not really been used before the 20th century when Thomas Kuhn developed a specific epistemology based on this concept. The paradigm is defined as "*a constellation of concepts, values, perceptions and practices shared by a community and which forms a particular vision of reality that is the basis of the way a community organizes itself* " [Kuhn96]. Nevertheless, Kuhn himself admits that the use of the word remains rather vague: it is possible to identify twenty-two different meanings of the "paradigm" concept used in Kuhnian epistemology [Masterman70]. In the last edition of his book, Kuhn even recognized that the "paradigm" concept is vague but he explained that it is close to what he calls a "disciplinary matrix".

This leads us to consider in this paper the "paradigm" as *a way of representing the world, which necessarily includes conceptual tools and methods (the conjunction of these two elements forming what Kuhn called a disciplinary matrix), such that an observer can create models*. Each paradigm refers to a particular ontology and represents a subset of "what is representable". The representation abilities of a paradigm are basically related to the conceptual tools, to the modeling methodology and the use of these two elements by theoreticians.

3.3.1.2 Paradigms and Software Engineering

The first paradigm to be introduced in SE was the **procedural paradigm** [GA04]. It was based on the use of algorithms to execute particular tasks. The second paradigm was the **data-hiding paradigm**, which focused on the data's organization and introduced the concept of modules (to hide the data's). This paradigm was followed by the **data-abstraction paradigm,** which concentrated on the types and on the operations defined on these types. Next was the **object-oriented paradigm**, "built upon the data-abstraction" paradigm but introducing new concepts like inheritance and polymorphism. Finally, using the flexibility of the component-oriented logic, the **agent-oriented paradigm** has divided software into independent and communicating entities called "agents" [WJ95]. This last paradigm has been described in detail in Chapter 3.

Programming languages and modeling paradigms are interdependent. The "chicken and egg" metaphor could be used to characterize their reciprocal relationship [GA04]: sometimes, specific needs for a programming language lead to a better implementation of a modeling paradigm and sometimes, the evolution of the modeling paradigm influences and improves the development of a specific programming language. However, even if programming languages and modeling paradigms are interdependent, the agent-oriented paradigm differs in the sense that it is not formally related to specific programming languages. The concepts used in AO have been inspired from the organizational structures found in the real world. In the beginning, agent-oriented models were implemented in object-oriented languages but further

evolutions allow to support and directly implement multi-agent systems in terms of full-fledged agent concepts such as *Beliefs*, *Desires* and *Intentions* (*BDI*) (see [Jack, Jadex]).

3.3.2 A Lakatosian Perspective of Software Engineering

In the following sections, we will propose to review the Kuhnian vision of OO and AO to demonstrate that it is not best suited to describe the evolution in SE. With respect to the research programme concept developed by Lakatos, we will explain why a Lakatosian understanding of the evolution to AO is more appropriate than a Kuhnian one.

3.3.2.1 "Research Program" Concept: a Definition

In the continuity of the Popperian philosophy (which will be briefly presented in the following section), Imre Lakatos has developed in 1974 an original approach of science. He considers scientific theories as general structures he calls "research programmes". A Lakatosian research programme is a kind of scientific construction, a theoretical framework which guides future research (in a specific field) in a positive or negative way. Each research programme is constituted by a *hard core*, a *protective belt of auxiliary hypotheses*, a *positive* and a *negative heuristic*.

The **hard core** is composed of general theoretical assumptions which constitute the basic knowledge for the programme development. In other words, these axioms are the assumptions the theorists will not challenge in their research. This hard core is surrounded with a **protective belt** composed of the auxiliary hypotheses, which complete the hard core and with assumptions related to the description of the initial conditions of a specific problem. These auxiliary hypotheses will be thoroughly studied again, widened and completed by theorists in their further studies within the programme. This widening of the protective belt hypotheses contributes to the evolution of the research programme without calling into question the basic knowledge shared by a scientific community.

The **positive heuristic** represents the agreement among the theoreticians over the scientific evolution of the research programme. It is a kind of "problem solving machinery" composed by proposals and indications on the way to widen and enrich the research programme. The **negative heuristic** is the opposite of the positive one. Within each research programme, it is important to maintain the basic assumptions unchanged. It means that all the questions or methodologies that are not in accordance with the basic knowledge must be rejected. All doubts appearing about the basic knowledge of the main theoretical framework become a kind of negative heuristic of the research programme. When the negative heuristic becomes more and more important, a research programme can become "degenerative" (i.e. it has more and more empirical anomalies). This means that theoreticians have to reconsider the basic knowledge of the programme, which can lead to the creation of another research programme. Let us mention that this revision is always a very slow process.

According to Lakatos, we can characterize the evolution of knowledge as a series of “problems shifts” which allow the scientific theories to evolve without rejecting the basic axioms shared by theorists within a specific research programme. The concept of “research programme” represents a descriptive and minimal unit of knowledge, which allows for a rational reconstruction of the history of science.

At first glance, the “research programme” concept seems rather close to the “paradigm” concept. Indeed, it is, in both cases, a matter of “disciplinary matrix” used to describe a particular ontology of the external world. However, differences exist between these two concepts especially in the evolution of science and knowledge in a large sense.

According to Kuhn, the evolution of science does not follow a straight line and does not converge towards something, which would be the « truth ». In the Kuhnian vision, the evolution of science could be represented by a broken line where discontinuity would mark the passage from one paradigm to another. From this point of view, different paradigms cannot be compared. Moreover, Kuhn specifies that a paradigm always emerges within a discipline facing a methodological crisis (characterized by the absence of a dominating theoretical framework) [Kuhn96]. Following a crisis undergone by a previously dominating paradigm, a new paradigm emerges with a new language and a new rationality. This new way of thinking does not allow a comparison between the old and the new paradigm. Given that a new paradigm is a new way of thinking about the world, there is no basis for comparison. The “paradigms incommensurability” thesis has become a very well known issue in the philosophy of sciences [Sankey94].

Lakatos decomposes the evolution of science into successive methodological and epistemological steps. These steps form a kind of vertical structure built with a multitude of “layers of knowledge” and where each layer represents a particular research programme. In the Lakatosian vision, the emergence of a new research programme is induced by an empirical degeneration of a previously dominating research programme. The new research programme will constitute a superior layer of knowledge, which will integrate the same conceptual tools as the former but which would be able to solve its empirical anomalies through what Lakatos calls a “problem shift”. The latter is characterized by an extension or a redefinition of the protective belt of the preceding programme. In this vision, research programmes remain comparable to each other (in both conceptual and empirical terms). The language and rationality of the new research programme result from a progressive evolution of knowledge and from the resolution of the empirical anomalies of the previous research programme. In contrast to the kuhnian vision, Lakatos explains that there is no discontinuity between the different research programmes.

3.3.2.2 Agent-Orientation: Paradigm Vs Research Program

In this section, we present three main arguments for the use of the Lakatosian research programme to understand the shift from OO to AO.

Kuhnian Crisis or Lakatosian Problem Shift?

As discussed in the first part of this paper, a SE-crisis has been observed due to the fact that few software projects successfully manage to fully satisfy users' requirements. In the Kuhnian vision, this crisis could be considered as a favourable argument for the emergence of a new paradigm. In this perspective, a crucial question raises: "*does the current crisis characterize the end of a dominating paradigm or is it simply the result of a pre-paradigmatic step specific to "young sciences" which have not found a dominating paradigm yet?*" Using the Kuhnian rhetoric to analyse this crisis, we can consider the situation as a paradigm evolution. Indeed, the pre-paradigmatic step was rather characterized by the procedural framework (which was defined by an algorithmic and sequential, i.e., a strictly computer/mathematical logic) as well as the data-hiding and the data-abstraction frameworks. This pre-paradigmatic period was essential to the evolution process towards OO. However, the crisis situation observed in SE must be carefully analysed. Even if the "SE-crisis" diagnosis has been noted for several years, we think that the context in which AO has emerged cannot be considered as a crisis in the Kuhnian sense. Indeed, most of the methodological rules existed before AO and the current software development process does not seem to be so chaotic: IT specialists dispose of analysis methods and methodological tools with a high level of abstraction (see for example [Kruchten03, CKM02]). These elements tend to show that what looks like a crisis is rather an (animated but normal) evolution of knowledge in SE (see [Skarmeas99, Odell02]), which could be interpreted as a "problem shift" in the Lakatosian vision.

Kuhnian Discontinuity or Lakatosian Continuity?

We consider that the Kuhnian discontinuity between paradigms is not appropriate to explain the emergence of AO because there is no real "fracture" between OO and AO. Indeed, in some software solutions, communicating objects are used and are completely relevant and sufficient. In software problems where no learning skills are valuable, the use of agents would not bring crucial advantages: they would just transfer messages and would not behave like learning and collaborating agents pursuing goals [Odell02]. In this special case, there is no contribution of the agent concept to the software solution in comparison to object technology. We can see that the cohabitation within the same application between modules exploiting object technology and others exploiting agent technology can thus be an "optimal" solution (see [Odell02]). In a Lakatosian vision, this cohabitation represents a progressive evolution of knowledge in SE. Indeed, the Lakatosian epistemology implies that the transition between research programmes is not clear and depends on the specific aspects of the experiment conducted (the software solution is the experiment in our case). A hybrid solution between modules developed on the basis of objects and others on the basis of agents can thus be explained by the continuity between research programmes inherent to the Lakatosian vision of knowledge applied to SE.

Kuhnian Incommensurability or Lakatosian Commensurability ?

Another drawback of the adoption of the Kuhnian epistemology in SE is the incommensurability between paradigms. OO and AO can be compared since collaborating agents can be used as communicating objects and, more important, agents can be implemented using object-oriented languages (see for example [BB97,

be implemented using object-oriented languages (see for example [BB97, BPR00[4]]). In this perspective, agents can be considered as "super objects" i.e. objects possessing skills as collaboration, intentionality, learning, autonomy, reasoning, etc. If we consider SE development as a history of "raising the level of abstraction", AO can be seen as an evolution of OO because it raises that level a little higher (see [Skarmeas99]). In this perspective, research programmes preceding OO can also be considered as lower layered than the later. This vision perfectly matches with the Lakatosian concept of layers of knowledge introduced earlier. Indeed, considering SE evolution, each new research programme raises the abstraction level and constitutes a higher layer of knowledge. These layers are comparable so that OO and AO are said to be commensurable.

3.3.3 Lessons Learned

AO is based on the basic knowledge that existed before its emergence. We could say that agent-oriented modelling and programming has a **hard core** composed of the concepts defined in the previous research programmes (procedural, data hiding and data abstraction) on the one hand, and the artificial intelligence field [WJ95] on the other hand. The **protective belt** of the AO research programme would be characterized by the evolution towards a widely used SE methodology allowing the development of large projects.

Table 3-1 summarizes the contrast between the Kuhnian and Lakatosian epistemologies applied to the evolution from OO to AO. The lessons learned are:

- The context in which AO has emerged cannot be considered as a Kuhnian crisis because of the existence of strong methodological rules that emerged within OO preceding the emergence of AO. We claim there is a problem shift;
- AO seems to be based on the evolution of the previous methodological rules so that we point to continuity between OO and AO;
- OO and AO are directly commensurable since the latter can be conceptualized as a knowledge layer upon the first.

In the light of the arguments presented above, we contend that the Kuhnian epistemology often referred to in the literature is not appropriate to provide a correct epistemological analysis of the knowledge in SE. We rather propose to use the Lakatosian epistemology to characterize the emergence of AO.

We argue that the Lakatosian epistemology is directly in line with the idea of computer knowledge depicted as a "structure in layers". We have represented this architecture in the following Figure 3-2.

[4] Jade is an extension of the object-oriented programming language Java.

		Kuhnian paradigm	Lakatosian research programme
Evolution Steps	**Knowledge Emergence**	Crisis	Problem shift
	Knowledge Evolution	Discontinuity	Continuity
	Knowledge Maturity	Incommensurability	Commensurability

Table 3-1. Kuhnian and Lakatosian Visions of AO Emergence.

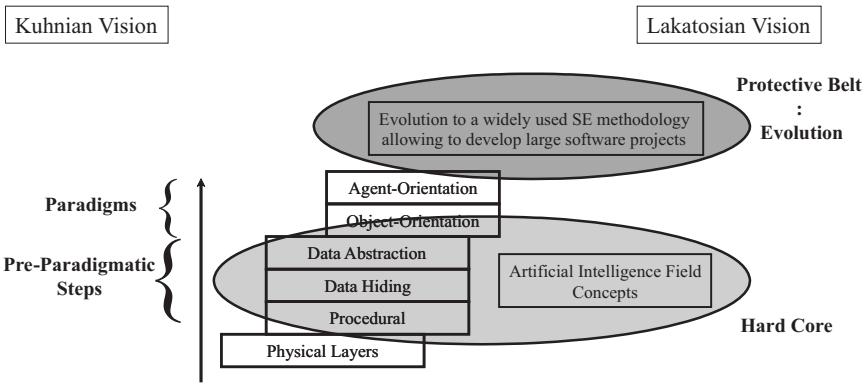


Figure 3-2. Evolution of Knowledge in SE

3.3.4 Implications

In this section, we briefly discuss the implications of using the concept of research program rather than paradigm to characterize OO and AO for both software engineering researchers and practitioners.

Our work provides SE researchers a complementary view and the specific nature of the research area they are working on. Moreover, the impact of this nuance is important due to the fact that, as we have pointed out, no fundamental departure was observed when evolving from OO to AO. Therefore, concepts developed in OO and other related research areas can be adapted to AO with some consequence on the process of building agent ontologies.

Managers and other SE practitioners will learn that AO should be envisaged as a natural evolution rather than as a complete revolution. AO can be seen as complementary to OO and leads to the fact that the modularity of software solutions resulting from several techniques can be utilized when developing new systems. Software modeling techniques can then be considered as problem oriented, i.e., modeling, design and implementation techniques are driven by the problem specificities, rather than solution oriented (where those should be driven by an ex ante methodological/technical choice). This conclusion can be taken at different levels:

- At the analysis stage where organizational modeling can be better performed using models representing collaborative agents (see for example the i* diagrams in [Yu95]) while purely functional requirements can be simply modeled using object modeling languages such as UML use cases;

- At the design stage, entities required to collaborate and learn can be designed as agents while others requiring less sophisticated behavior can be designed as objects;

- At the implementation level, modules can be implemented using agent technologies that facilitates communication with modules implemented in object technology.

Finally, our new conceptualization has a profound impact on the SE development life cycles. Indeed, the literature on the evolution of software engineering life cycles is rich and the adoption of mature development life cycles such as the spiral model, the Rational Unified Process, or agile methodologies traditionally operationalized by OO technologies can be adapted to AO development without revising the fundamental concepts.

3.4 Chapter Summary

User requirements that have been poorly taken into account as well as modeling and development languages inspired by programming concepts as opposed to organization and enterprise has led to a software crisis. Solutions can be found at different levels, and among those, iterative development has been one of the natural software processes evolution. Indeed, proceeding iteratively allows, thanks to continuous feedback loops, to develop a software product having a perceived quality in better accordance to what users expect.

This chapter has presented an epistemological analysis of SE at both macro and micro levels. At macro level, the paper concludes SE is a human science; indeed due to the impossibility of a crucial experiment, upcoming innovations can be based on refinements of existing theories and do not induce a complete refutation of existing techniques. At micro level, we studied knowledge evolution in the software process itself which is comparable to the Lakatosian approach used at macro level. Further-

more, at the same level we used Herbert Simon's bounded rationality principle to characterize project actors' rationale. The paper finally advocates that on the basis of the epistemological lecture we had, the use of iterative development is better adapted to the processes of SE particularly due to how knowledge evolve and how actors are bounded rational.

OO at the first level and AO at the second level also constituted definite progress to fight the software crisis. AO furnishes concepts to model the organization more precisely, thus enabling analysts to create more accurate models. This chapter has also presented an epistemological analysis of these improvements. AO is a promising innovation in tools allowing analysts to model the organization and user requirements. We have shown in this paper that the emergence of this innovation can be better studied using Lakatosian ideas rather than Kuhnian ones. In this vision, AO would be considered as a research programme rather than a paradigm. Even if it seems to be just a matter of terminology, the differences between these two epistemological concepts is clear and profound so that they should not be confused. The chapter also advocates that the approaches taken by both researchers and practitioners to software problems and their possible solutions is influenced by the vision of OO and AO as a research program.

More work should be carried out in SE fundamentals by studying other methodological frameworks and epistemological foundations to provide the computer science researcher a more accurate conscience of the research field that he or she is working on.

References

[Arthur94] W.B. Arthur. “Inductive Reasoning and Bounded Rationality”. *American Economic Association Annual Meetings*, 1994.

[Basili92] V.R. Basili. “The Experimental Paradigm in Software Engineering”. In *Experimental Software Engineering Issues : Critical Assessment and Futures Directives*, *H.D. Rombach, V.R. Basili, R.W. Selby Editors*, Springer-Verlag, pp. 3-12, 1992.

[BPR00] F. Bellifemine, A. Poggi and G. Rimassa. “Developing Multi-agent Systems with JADE”. In *Proceedings Seventh International Workshop on Agent Theories, Architectures, and Languages* (*ATAL 2000*), 2000.

[BB97] J.P. Bigus and J. Bigus. “Constructing Intelligent Agents with Java: A Programmer’s Guide to Smarter Applications”. *John Wiley & Sons*, 1997.

[CKM02] J. Castro, M. Kolp and J. Mylopoulos. “Towards A Requirements-Driven Development Methodology : The Tropos Project”. In *Information Systems*, Elsevier, 2002.

[Davis95] A. Davis. “201 principles of software development”. *McGraw-Hill*, 1995.

[GA04] E. Göktürk and N. Akkok “Paradigm and Software Engineering”, In *Proceedings of Impact of Software Process on Quality*, May 2004.

[Jack] JACK Intelligent Agents. http://www.agent-software.com/

[Jadex] JADEX BDI Agent System. http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

[JW01] N.R. Jennings and M. Wooldridge. “Agent-Oriented Software Engineering”. In *Handbook of Agent Technology*, J. Bradshaw, AAAI/MIT Press, 2001.

[Kaisler05] S.H. Kaisler. “Software Paradigms”. *John Wiley & Son*, 2005.

[KK82] S. Kaplan and R. Kaplan. “Cognition and Environment: Functioning in an Uncertain World”. *Praeger*, 1982.

[Kruchten03] P. Kruchten. “The Rational Unified Process: An Introduction”. *Addison-Wesley*, 2003.

[Kuhn96] T. Kuhn. “The structure of Scientific Revolutions” – Third Edition. *University of Chicago Press*, 1996.

[Lakatos77] I. Lakatos. “The Methodology of Scientific Research Programmes: Philosophical Papers Volume 1”. *Cambridge University Press*, 1977.

[Masterman70] M. Masterman. “The Nature of Paradigm”. In *I. Lakatos and A.Musgrave, Criticism and the Growth of Knowledge*, Cambridge University Press, 1970.

[Meyer97] B. Meyer. “Object-oriented Software Construction. Second Edition”. *Prentice Hall*, 1997.

[Odell02] J.J. Odell. “Objects and Agents Compared”. In *Journal of Object Technology*, vol. 1, no. 1, pp. 41-53, May-June 2002.

[Popper59] K.R. Popper. “The Logic of Scientific Discovery”. Hutchinson, London, 1959.

[Popper72] K.R. Popper. “Objective Knowledge, An Evolutionary Approach”. *Oxford University Press*, 1972.

[Priestley05] M. Priestley. “The Logic of Correctness in Software Engineering”. *CAiSE Workshops*, 2005.

[Sankey94] H. Sankey. “The Incommensurability Thesis”. Ashgate, Sydney, 1994.

[Simon83] H.A. Simon. “Models of Bounded Rationality”. *MIT Press*, Cambridge, 1983

[Skarmeas99] N. Skarmeas. “Agents as Objects with Knowledge Base State”. *Imperial College Press*, 1999.

[Sommerville01] I. Sommerville. “Software Engineering” – Sixth Edition. *Addison-Wesley*, 2001.

[TD99] C. Toffolon and S. Dakhili. “Requirements Elicitation Process: An Iterative Approach based on the Spiral Model”. In *XVIth International Symposium on Information and Computer Sciences* (*ISCIS'99*), Izmir, Turkey, October 1999.

[TK79] A. Tversky, D. Kahneman. “Prospect Theory: an analysis of decisions under risk”. *Econometrica*, Vol. 47, Issue 2, 263-292, March 1979.

[WJ95] M. Wooldridge and N.R Jennings. “Intelligent agents: Theory and practice”. *The knowledge Engineering Review*, 10(2): 115-152, 1995.

[Yu95] E. Yu. “Modeling Strategic Relationships for Process Reengineering”. *PhD thesis*, *University of Toronto*, *Department of Computer Science*, Canada, 1995.

4 Towards an Iterative MAS Development Methodology

As overviewed into the previous chapter, software development methodologies can use plenty of different life cycles. In this chapter we will, between others, study the following question: *facing a particular software development methodology what features must be studied to evaluate its life cycle?* Indeed, purely waterfall development methodologies can advise their users to repeat their phases "iteratively" during the project but no theoretical framework nor any template to support this kind of development has been developed or furnished (see [BS02, BGPG02]). Consequently each waterfall development methodology can easily add this advice to its definition and pretend to be iterative. We believe this is widely abusive and claim that a SE methodology using an iterative SDLC has to offer a custom theoretical framework often called project management to support, assist and monitor iterative development. This chapter will be dedicated to the description and justification of the I-Tropos methodology building process. I-Tropos is a multi-agent software development methodology using an iterative life cycle being the focus of this dissertation. It offers disciplines of different nature: SE as well as PM ones; nature determination will be justified in this chapter.

This chapter firstly introduces agent-oriented software engineering (AOSE) and overviews its multiple aspects. It then distinguishes software engineering (SE) from project management (PM) and applies the principles on I-Tropos disciplines to raise borders between the two fields. A framework of elements allowing to study the MAS development methodologies is then introduced and applied on four main processes: Tropos, MaSE, ADELPHE and MASSIVE. The building process of I-Tropos is finally overviewed, it encompasses the theoretical foundations of the process, the study of a mean to formalize (document) the process, the way the process has been designed, a comparison with the traditional Tropos development method and finally points to further works and developments.

4.1 Agent-Oriented Software Engineering

New technologies have raised the need for software that is robust, reactive to change in the environment, evolve over time and can support dynamic architecture. In applications such as eBusiness, peer to peer networks or ubiquitous computing, software components must be able to compose services dynamically, establish and drop partnership with other component in order to achieve their goals. Learning, planning, negotiating and communication are essential in this kind of software.

[Bradshaw97] defines the software agent as "*a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes*". Autonomy and continuity derives from the need of agents that are responsive to change in the environment without requiring human intervention. It

should ideally learn from its experience when it stays in an environment for a long time. We also expect agent that are in the same environment to cooperate to achieve their goals. In order to fulfill such behavior, agents are required to be:

- Autonomous: capable of acting without external intervention;
- Interactive: communicates with environment and other agents;
- Adaptive: capable of responding to environment or other agents, and is able to modify its behavior on the basis of its experience;
- Mobile: able to transport itself from one environment to another;
- Proactive: acting in order to meet its goals. Not just reacting to its environment;
- Intelligent: state is formalized by knowledge, and is able to choose an action based on that knowledge;
- Unpredictable: his actions are unpredictable, even knowing its state. Nondeterministic behavior;
- Continuous: a continuous running process;
- Robust: able to deal with incomplete data and errors;
- Transparent: can be transparent when needed, but can provide a log of his actions when needed.

4.1.1 Multi-Agent Systems

Most of the time, an agent exists in an environment inhabited by other agents, called a Multi-Agent System (MAS). It is then generally conceived as a society of autonomous, collaborative, and goal-driven software components (agents), much like a social organization [Do05].

A MAS is a system composed of several agents capable of reaching goals, possibly commons, that are difficult to achieve by an individual system. The global behavior of a MAS derives from the interactions among the constituents agents: they cooperate, coordinate or negotiate with each other.

4.1.2 Benefits of AOSE

No evidence yet exists to suggest that the agent-oriented software engineering will become widely used in every day enterprise developments. There are, however, convincing arguments for believing that it will be of benefit for engineering some complex software systems [JW99].

According to Jennings and Wooldridge [JW99], the main arguments are:

- the effectiveness of agent-oriented decompositions in partitioning the problem space of a complex system. A complex computerized system can be decomposed into smaller components, the same way that a multiagent system can be decomposed into the elements that constitute the system (software agents);

- the suitability of the key agent-oriented abstractions, such as agents, (social) interactions and organizations, in modeling complex systems;

- the appropriateness of the agent-oriented philosophy for dealing with the dependencies and the interactions that exist in a complex system.

The factor that really makes agent-oriented software engineering distinct from any other software engineering paradigm is the higher level of abstraction employed in the development of software systems. The idea of modeling a system in terms of autonomous entities with characteristics similar to humans introduces a close-to-real-life modeling of the system, and therefore makes the development of the software system natural.

However, as mentioned by Kinny et al. [KGR96], "*if agent-oriented software engineering is to become widely accepted as a paradigm for the development of large-scale applications, adequate agent-oriented methodologies and modeling techniques will be essential. This is not just to ensure that systems are reliable, maintainable, and conformant, but to allow their design, implementation, and maintenance to be carried out by software analysts and engineers rather than researchers*".

This argument summarizes what is well known within the agent research community: the existence of mature and complete methodologies, to help developers to model software systems by taking into account the unique characteristics that agent orientation introduces, is an important issue for the success and wide acceptance of agent-oriented software engineering.

4.1.3 MAS Development Methodologies

Agent-oriented software engineering methodologies have been mainly divided into two principal categories: those inspired by object-oriented methodologies and those that have been developed with agent orientation in mind. More insight into a genealogy of agent-oriented methodology is presented by Brian Henderson-Sellers in [GHW04].

Several reasons can be cited that justify the extension of current object-oriented methodologies for the development of systems with agent orientation in mind [IGG99]. These include the similarities that can be found between the two main concepts, namely object and agent [WC01, KGR96], the common usage of object-oriented languages, such as JAVA, for the implementation of agent-oriented systems,

and the familiarity of many software engineers with object-oriented methodologies [IGG99]. This question is studied in depth in Chapter 4.

Unfortunately, many shortcomings can be identified on the extension of current object-oriented methodologies for the development of software systems with agent orientation in mind. The different characteristics of agent must be taken into consideration when developing an agent-oriented system. However, the level of abstraction and the models provided by object-oriented software engineering methodologies are not adequate to model these characteristics (e.g., agents are autonomous, and have intentions). Because of this, it has been argued [WJK00], that "*if agents are to realize their potential as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored to them*".

For all of these reasons, specific agent-oriented software engineering methodologies developed with agent concepts have emerged. Efforts towards this direction have grown rapidly the last few years, and as a result many methodologies based on agent-orientation have been developed (see [IGG99] for a review). The main advantage of these methodologies is the inclusion of models and notations to capture the unique characteristics that agent orientation introduces. In the next section we will study four of these methodologies with a strong focus on their development life cycle by using the framework depicted earlier.

4.2 Software Engineering versus Project Management

Previous section focused on AOSE; Software Engineering is, by definition, *the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software* [IEEE04]. On the other side, Software Project Management has been defined by [RUP] as *the art of balancing competing objectives, managing risks and overcoming constraints to successfully deliver a product which meets the needs of both customers (the payers of bills) and the users*.

For sure, there is a gap between the two fields and they have to be positioned. **Engineering** is the art of applying specific knowledge and utilizing natural laws and physical resources in order to design and implement materials, structures, machines, devices systems and processes that realize a desired objective and meet specific criteria. **Management** is the art of planning, organizing, resourcing, leading or directing, controlling an organization (a group of individuals or entities) for the purpose of accomplishing a desired goal. Intuitively the two terms seem to be complementary but have a different scope. Following [Brandon06], the two fields considerably overlap; in Figure 4-1, their vision of SE and PM scope is depicted. Even if their vision is not clearly exposed and argued, we summarize it as follows:

- SE encompasses the traditional analysis, design, implementation, test and deployment disciplines;

- PM encompasses project scheduling, costs management, procurement, communications and human resources management;

- Defining the project scope, change, quality and risk management are part of the two disciplines.

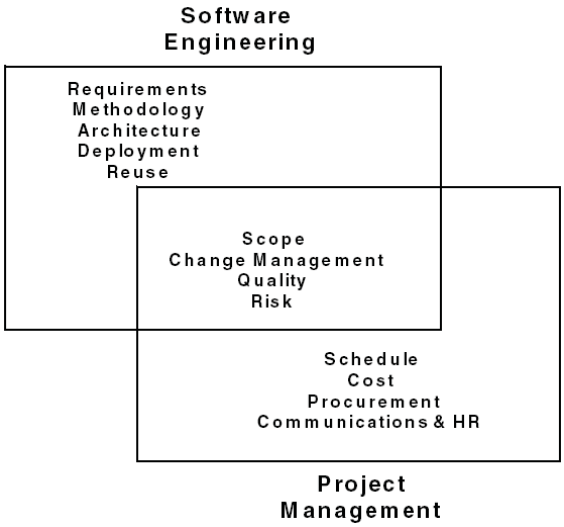


Figure 4-1. Software Engineering versus Project Management (from [Brandon06]).

Brandon's vision remains unfortunately rather vague. We will partially adopt their vision but nuance and adapt the ideas to the software process we develop. We will consider that, in software development, engineering refers to the ontologies[1], models, guidelines, programming languages and other artifacts used when building software allowing to formalize, represent and develop the desired system. On the other hand, the art of balancing human resources onto specific tasks to meet user requirements with respect to available calendar time and financial resources, identified risks, expected quality levels, etc. following a defined life cycle belongs to management. However, risk and quality management do refer to the two disciplines since:

- risk and quality factors are taken into account from a managerial perspective (at process level) to allocate more resources to deal with a risk or meet the desired quality level, prioritize requirements elicitation of the most risky issues etc.

[1] In both computer science and information science, an ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain.

- risk and quality factors are subject to technical choices, solutions, designs and implementations so that they are part of the software engineering field.

As far as we are concerned, we will consider the software engineering field when refeering to the I-Tropos core disciplines (*Organizational Modeling*, *Requirements Engineering*, *Architectural Design*, *Detailed Design*, *Implementation*, *Test* and *Deployment*) because they refer to practical technical choices to build the software product. The I-Tropos support disciplines (*Risk Management*, *Quality Management*, *Time Management* and *Process Management*) will be referred to as project management since they are mostly concerned with the management of resources to specific tasks. As pointed out earlier, risk and quality management could also reefer to software engineering but since, in this dissertation, we mostly focus on the "process" level, we reefer to them as project management. Finally, since PM regroups the process support disciplines we sometimes, in the rest of this dissertation, use the words *support framework*, *process support disciplines* as a synonym to PM.

We can now consider the position of the development methodologies such as Tropos. Tropos as defined in [CKM02] is only a set of phases into which one or several models are built with variable scope. With respect to the above definitions, all the phases described reefer to software engineering, none to project management. One can consider that the process is described in a generic or at least flexible way to be included into a defined software development life cycle (possibly iterative). This statement has, as far as we know, never been introduced by the methodology authors and we argue that:

- the use of an elaborated development life cycle (as iterative one) requires a specific framework including project management;

- a project management framework can hardly be defined in a completely generic manner particularly if it is model driven. Indeed, the later requires a study of the analysis models and its components to find the adequate scope elements to drive the software process. A couple of principles to define such elements can however be determined.

The project management framework described in this dissertation is applied to Tropos to build I-Tropos methodology. It constitutes a first attempt to include a mature development life cycle into MAS development. We could envisage, with some adaptations, to apply to other development methodologies which could be agent but also object oriented or even other research programmes[2]. The key feature to adapt the framework is the adequate identification and definition of scope elements into analysis models for model driven software project management.

Finally, as pointed out in [HSMGD04, Sousa08], the process, method and methodology terms are often considered as synonyms in SE literature. We consider here

[2] In the Lakatosian sense.

that method and methodology are synonyms, we will however have a contextual reading of the term *process* as exposed in this section. Indeed, in [BB01], a process is described "*as a set of work items, scheduled according to constraints, which all participate in fulfilling a common purpose*". On the basis of this definition, the process concept can be considered at several levels. Indeed, as pointed out by [HSMGD04], the term tends to include human resource and technological management, lifecycle issues and other elements included into software development phases. [BB01] however emphasizes by noting that the term may refer to a smaller scale as for example "software maintenance process". [Brinkkemper90] describes a method as "*an approach to perform a systems development project, based on a specific way of thinking, consisting of directions and rules, structured in a systematic way in development activities with corresponding development products*". According to us, this conception of the method term corresponds to the process "macro-level" described above. We will consequently consider, in this dissertation, the terms method and process as synonyms when the later reefers to "macro-level" processes, i.e., the whole of I-Tropos could be indifferently quoted as process or method. When referring to a "micro-level" process as, for example, a single SPEM discipline, the two terms cannot be interchanged. The process term is thus envisaged in a flexible manner and requires to be contextualized to evaluate if it can be used as a synonym of method/methodology or not.

4.3 A Framework for Evaluating SE Methodologies Life Cycle

At this stage, we need to study the existing MAS development methodologies into further details. To achieve this, we need to determine specific criteria to evaluate them. The goal of this dissertation is to create an operational MAS development methodology for huge software projects, evaluation elements will be selected for this purpose.

Modern software development methodologies as the Rational Unified Process [Kruchten03] dispose of a "two dimensional" lifecycle in the sense that a series of sequential disciplines are repeated iteratively. That is why we have to evaluate both the vertical and the horizontal coverage. The five factors on the basis of which we will evaluate this process are regrouped under "***Engineering concepts***".

First of all, we will evaluate the vertical life cycle coverage. In that sense we firstly evaluate how the process is divided into multiple "highest level" stages, this element will be called the ***Process cutting***. At second, we will evaluate whether these stages cover the traditional phases of software development or more, this element will be called ***Covered stages***. The process can encompass the traditional stages in a superficial manner, that is why we next evaluate the ***Stages scope***. Through these three elements we will be able to evaluate the vertical dimension of a particular process at various levels of granularity.

Next we will overview the horizontal dimension of the process through the ***Process lifecycle*** element. This element will evaluate if the process is purely waterfall or if

a template or, better, a framework[3] for the use of more advanced development lifecycles.

As pointed out previously, software engineering and project management are two different but complementary disciplines. The remaining elements that will be evaluated on each envisaged MAS methodology are regrouped under "***Project management concepts***".

An advanced software development method requires a framework to support its successful application. First of all, the process should be documented adequately, preferably using an adequate method engineering standard (for more information see section 4.5.2 formalizing the process), that is why we will evaluate the availability of a ***Process description***. At second stage we will evaluate how the method takes different aspects of software project management into account. We will evaluate the availability of a ***Risk management***, a ***Quality management*** and ***Time management*** framework because these are the one that will be developed in this dissertation in the context of I-Tropos[4].

Finally, two operational aspects of the process will be evaluated. At first, the presence of ***Software metrics*** to evaluate the process advancement and performance. At second, the availability of a ***CASE tool*** to support the methodology's developments.

The framework includes an evaluation in terms of:

- **Engineering concepts**. How does the process organize its life cycle? this criterion can be evaluated in terms of:
    - **Process cutting**. Is the process divided into phases, views, disciplines?
    - **Covered stages**. Does the methodology cover all software development stages (analysis, design, implementation, test, etc.)?
    - **Stages scope**. Is the analysis stage rather requirements-driven, technology-driven or both? Does it focus on both stakeholders requirements and business modeling? Does it encompass functional and non-functional requirements? Is the design phase divided into architectural and detailed design?

[3] In the context of this dissertation when we refer to an iterative template, we only refer an architectural pattern as the RUP profile (see [Kruchten03]), when we refer to an iterative framework, we refer to a whole set of tools to support iterative development.

[4] Note that we could have evaluated the availability of a *Process Management* framework since this is also overviewed in I-Tropos but we think this feature as too specific to be evaluated here.

  - **Process lifecycle**. Is the lifecycle waterfall, incremental, iterative, spiral,…?

- **Project management concepts**. Does the methodology offer a project management perspective to drive software development? This criterion can be evaluated in terms of:

  - **Process description**. To what extend is the process described: is the phase description limited to its objective and models or is there a complete phase description using process meta-concepts? Is there a formal role/activities/input or output models description using process meta-concepts?

  - **Risk management**. Does the methodology define a risk management framework designed to evaluate threats?

  - **Quality management**. Does the methodology define a quality management framework designed to evaluate quality factors?

  - **Time management**. Does the methodology offer a method to plan software development in terms of time and resources allocation?

  - **Software metrics**. Does the methodology furnish explicit milestones/measures allowing to evaluate the project advancement/performance?

  - **CASE tool**. Is there a case tool to support software development?

4.4 AOSE Methodologies Evaluation

This section overviews and evaluates four main AOSE methodologies. Those methodologies are rather different from each other and have been selected for multiple reasons. It has not been possible to be totally exhaustive in the existing methodologies overview, four where selected focusing on the following characteristics:

- The methodology's analysis models to evaluate the potential for model driven development;

- The methodology's announced lifecycle to evaluate the potential for iterative development;

- The methodology's process cutting to evaluate the potential for inclusion in an existing iterative template such as the Rational Unified Process (see [Kruchten03]).

On the basis of a rough evaluation of those criteria we selected the following methodologies for further evaluation:

- Tropos uses the i* strategic dependency and strategic rationale models at analysis stages; those models define and use organizational elements (actors, agents, goals, objectives, responsibilities, social dependencies, etc.) as fundamentals to conceive systems through all the development process. The study of this method is consequently of particular interest;

- MaSE views MAS as a further abstraction of object-orientation where agents are a specialization of object. It both uses goals capture and use cases at analysis stages, it has consequently been selected;

- ADELPHE claims following the Rational Unified Process template and uses use-case models at analysis stage, that is why we also choose to study this method in further details;

- MASSIVE is view-oriented rather than decomposed into disciplines, it also uses an iterative life cycle; its study is complementary to the one of the previous methodologies.

The rest of this section is devoted to the study and comparison of these methods following the criteria exposed in the previous one.

4.4.1 Tropos

Tropos [CKM02] is a MAS software development methodology founded on intentional and social concepts, inspired by early requirements analysis.

Tropos is based on the fact that existing software development methodologies have been inspired by programming concepts, not by organizational ones, leading to a semantic gap between software and its environment [CKM02]. Tropos uses i*[5] organizational modeling framework, which offers the notion of actor, goal and dependencies. The other models are often made using Unified Modeling Language and Agent UML [AUML].

This methodology is founded on social and intentional concepts, and inspired by early requirement analysis. Software is viewed from five complementary perspectives:

- Social: who the actors are, what they want to achieve, what responsibilities they have, what they can do.

- Intentional: what the goals are, how the goals interact, how the goals are met.

[5] i* : i star, or distributed intentionality

- Communicational: how the actors can interact with each others.
- Process-oriented: what the business and computer process are, what they are responsible of.
- Object-oriented: what the object and classes are, what they relationships are.

Following [GKMP04], Tropos is based on two key ideas:

- The notion of agent and the related mentalistic notions, such as goals and plans, are used in all phases of software development, from early analysis down to the actual implementation;
- The methodology covers the very early phases of the requirements analysis, thus allowing for a deeper understanding of the environment where the software-to-be will eventually operate.

4.4.1.1 Phases

This section overviews the five Tropos development phases: early requirements, late requirements, architectural design, detailed design and implementation.

Early Requirements Analysis is concerned with the understanding of a problem by studying an existing organizational setting. The emphasis is put on understanding the motivation and rationale that underly system requirements. During this phase, requirements engineers model the target domain in terms of (social) actors and intentions. This analysis addresses directly the Object-Oriented deficiencies in modeling the problem domain. Each goal intention is analyzed from the point of view of the actors resulting in a set of social dependencies between actors.

Requirements Analysis phases in *Tropos* are influenced by Eric Yu's *i** modeling framework [Yu95]. This framework includes two models formalized with the intentional concepts from the BDI model:

- the Strategic Dependency Model (SD) describes the network of social relationships among actors;
- the Strategic Rationale Model (SR) describes and supports the reasoning through which each actor goes with respect to its relationships with other actors.

Late Requirements Analysis extends the models created in the previous step with the target system in its environment. The system 'to-be' is modeled as one or more actors. Its interdependencies with other actors contribute to the accomplishment of stakeholder goals. Therefore, these dependencies define the target system's functional

and non-functional requirements. The same process (*i.e.*: SD and SR analysis), is operated again but with the focus on the intended software system.

The objective of **Architectural Design** is to organize the dependencies between the various sub-actors identified in the previous phases in order to meet functional and non-functional requirements of the system.

A Multi Agent System can be seen as a social organization of autonomous software entities (agents) that can flexibly achieve agreed-upon intentions through their interactions. Following [BCM03], "*A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together.*"

During Architectural Design, models will be refined with respect to:

- the inclusion of new actors due to the delegation of sub-goals upon analysis of system's goals;
- the inclusion of new actors according to the choice of a specific architectural style [KGM06];
- the inclusion of new actors positively contributing to the fulfilment of non functional requirements.

During **Detailed Design**, the behavior of each architectural component is defined in further detail. This discipline is concerned with the specification of the agent micro level taking into account the implementation platforms. The objective is to perform a design that will map directly to the code.

During the **Implementation** phase, the system implementation is carried out consistently with detailed design.

4.4.1.2 Scope

[BS02] points out about a requirement model that "*(…) iterative steps of incremental refinement of the model have to be performed. (…), this way of proceeding is a typical feature of the Tropos methodology. The final setting of each Tropos phase may be reached possibly after several refinements, in each of which not only new details may be added, but, also, already present elements and dependencies can be revised or even deleted. This iterative process not only may require intra-phase refinements, but, possibly, also revisions of artifacts produced during earlier phases (inter-phases refinements). The importance of retrotraceability is, here, evident.*" This statement highlights the need for an iterative perspective in Tropos software development methodology. Nevertheless we argue that Tropos cannot be considered as a development process using an iterative development life cycle because:

- In terms of engineering concepts:
    - Only a series of sequential activities are defined;
    - Phases only encompass Analysis, Design and Implementation; the Testing phase is not covered.
- In terms of project management:
    - Process description remains basic;
    - No project schedule and risk management frameworks are defined;
    - No software metrics are defined.

In other words, there is no project management framework to support iterative development is introduced and Tropos cannot be considered as iterative.

4.4.1.3 Methodology Evaluation: Summary

In this section the methodology's is evaluated on the basis of the criteria defined framework earlier in this chapter.

- Engineering concepts:
    - Process cutting. Process divided into 5 phases: early requirements, late requirements, architectural design, detailed design and implementation;
    - Process lifecycle. Waterfall process.
    - Covered stages. Tropos does not cover the test stage;
    - Stages scope.
        - Analysis stages cover both organisational environmental and user requirements aspects;
        - The design stage is divided in both architectural and detailed design.
- Project management concepts:
    - Process description. No formalization of the process concept on the basis of any method engineering standard;

- Risk management. Secure Tropos [GMMZ06, MMMDHG08], an extension of the Tropos, is a formal framework and methodology for modelling and analysing security requirements. The framework is adapted at micro level, i.e. at the level of i* modelling and is not used for project management.

- Quality management. The i* framework used in Tropos allows the representation of softgoals, The framework is however adapted at micro level, i.e. at the level of i* modelling and is not used for project management.

- Time management. No framework for time management is offered.

- Software metrics. No milestone or metric are furnished.

- CASE-Tool. Several CASE-tools are available and allow different levels of representation, for a complete list and description see [i*wiki].

4.4.2 MaSE

The primary focus of *Multi-agent Systems Engineering* (*MaSE*) is to help a designer take an initial set of requirements and analyze, design, and implement a working multi-agent system [DWS01]. It is viewed as a further abstraction of object-orientation where agents are a specialization of object.

4.4.2.1 Phases

The MaSE life-cycle follows the phases and steps shown in Figure 4-2. The rounded rectangles show which model is used in each phases, while arrows show dependencies between these models. One of the strength of MaSE is that changes can be tracked forward or backward trough the different steps.

While is it shown as a single top-down flow, the method is actually iterative. The intent is that the analyst or designer be allowed to move between steps and phases freely such that with each successive pass, additional detail is added and, eventually, a complete and consistent system design is produced [DWS01].

The purpose of the Analysis phase is to produce a set of roles whose tasks describe what the system has to do to meet its overall requirements. Each role is responsible for achieving or helping to achieve goals of the system. Analysis phase consists of three steps:

- Capturing Goals: this phase is used to transform initial system specifications into a set of system goals. This step is divided into two sub-steps, identifying goals and structuring goals into a Goal Hierarchy Diagram;

- Applying Use Cases: the aim of this step is to capture a set of use case from the system context, and capture a set of sequence diagrams;

- Refining Roles: this last step of the analysis phase consists in transforming goals and sequence diagrams into roles and their associated tasks (which are more suitable for designing multiagent systems), and represented in a MaSE Role Model.

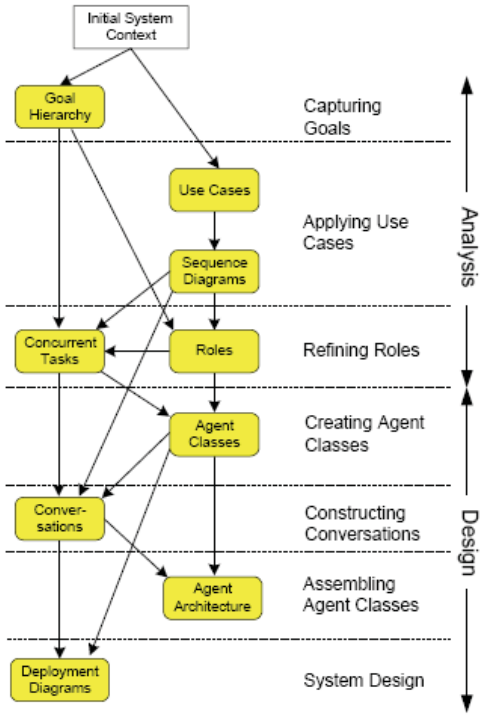


Figure 4-2. The MaSE Methodology Workflow.

The purpose of the Design phase is to define overall system organization by transforming roles and tasks into agent types and conversations. The Design phase is divided into four steps:

- Creating Agent Classes: using roles defined in the Analysis phase, agent classes are created, using Agent Class Diagram.

- Constructing Conversations: up to this point, no communication between agents has been defined. This is the purpose of this step, using Communication Class Diagrams.

- Assembling Agent Classes: internal of agents is being created, using Agent Architecture Diagram. This step is divided into two sub-steps: defining the agent architecture and defining the components that make up the architecture.

- System Design: in this final step of the methodology, actual agents are instantiated from previous defined agent classes. A Deployment Diagram is used to show the location, type and number of agents within the system.

4.4.2.2 Scope

MaSE is a sequential methodology, using seven steps. This methodology reuses some of the UML diagrams and concepts, as it defines agents as specialization of objects. The method organization is fairly simple, and no roles are defined in the methodology. The strength of this methodology is its simplicity, but there is a lack in the earliest phases of the methodology, as gathering specifications is not covered by the methodology.

4.4.2.3 Methodology Evaluation: Summary

In this section the methodology's is evaluated on the basis of the criteria defined framework earlier in this chapter.

- Engineering concepts:
    - Process cutting. Process divided into 4 phases: requirements, design, implementation and test;
    - Process lifecycle. The methodology advises to proceed with phases iteratively but no real framework is furnished.
    - Covered stages. All the classical phases of software development are at least partially covered;
    - Stages scope.
        - MaSE lacks in requirement phase because the methodology supposes that the development starts off with all requirements gathered. On the other hand, as a goal hierarchy is established, some part of the requirement phase is still covered;
        - A single design stage.

- Project management concepts:
    - Process description. No formalization of the process concept on the basis of any method engineering standard;
    - Risk management. No framework for risk management is offered.
    - Quality management. No framework for quality management is offered.
    - Time management. No framework for time management is offered.
    - Software metrics. No milestone or metric are furnished.
    - CASE-Tool. MaSE is supported by a CASE-tool called *Agent Tool*.

4.4.3 ADELFE

The ADELFE[6] methodology is based on the AMAS (Adaptive Multi-Agent Systems) theory (this theory gives local agent design criteria so as to enable the emergence of an organization within the system and thus, of the global function of the system), uses notations coming from UML and AUML, and advise to follow the Rational Unified Process (RUP) template.

4.4.3.1 Phases

The process is divided into three phases: Requirements, Analysis and Design. Each of these phases is divided into a certain number of steps, as shown in Figure 4-3, and produces some model, as shown in Figure 4-4.

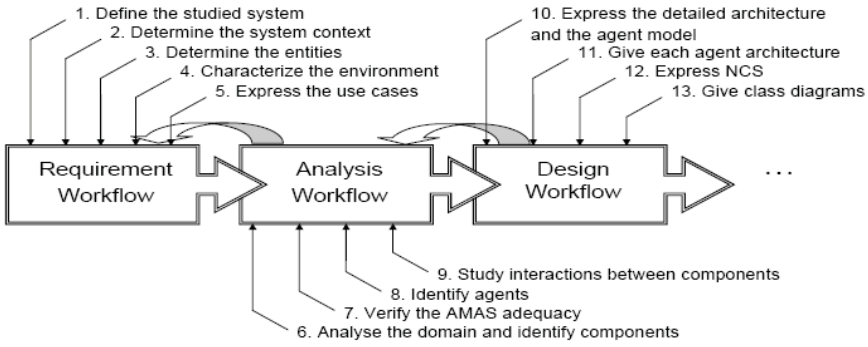


Figure 4-3. ADELFE Methodology.

[6] *ADELFE* means *Atelier de Développement de Logiciels à Fonctionnalité Emergente*

- Requirements: During this first phase, a definition of the system to be is made and is then transformed into a use case model. Organization and management of the functional and non-functional requirements is also one of the objectives. This phase produces the Environment Model;

- Analysis: There are two goals in this phase: identify the agents and verify the AMAS adequacy. This phase produces the Identification of the Agent Model;

- Design: In this last phase, agents are designed and their interaction languages are designed too. The system architectures and software components are defined. This phase produces the Agent and the Non-Cooperative Situation[7] (NCS) Models.

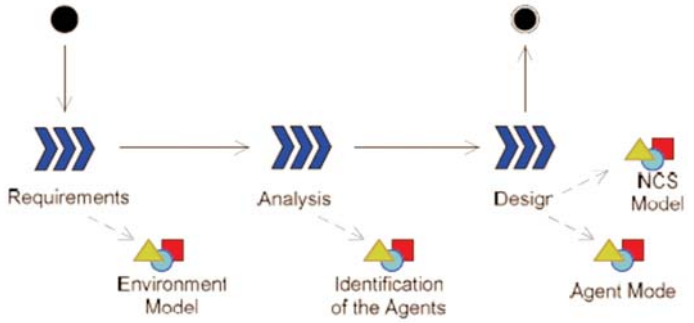


Figure 4-4. The ADELFE Process: Phases View.

4.4.3.2 Scope

The ADELFE methodology is sequential using three main phases. It defines roles and outputs for each phase and is precisely defined in [CS05] using Software Process Engineering Metamodel [OMG05] notation.

4.4.3.3 Methodology Evaluation: Summary

In this section the methodology's is evaluated on the basis of the criteria defined framework earlier in this chapter.

- Engineering concepts:

[7] "Non Cooperative Situation are unusual situations in which the agent must face an unpredictable environment. There are different kinds of NCS: incomprehension, ambiguity, incompetence, unproductiveness, concurrency, conflict, uselessness." [Adelfe05]

  - Process cutting. Process divided into 3 phases: requirements, analysis, design;
  - Process lifecycle. The methodology advices to follow the RUP template, no further information about how to follow it is furnished, no project management framework is described;
  - Covered stages. Implementation and test stages are not covered;
  - Stages scope.
    - The requirements stage models the environment, the analysis stage focuses on the user perspective;
    - A single design stage.

- Project management concepts:
  - Process description. A formalization of the process concept on the basis of SPEM-concepts is proposed in [CS05, GMP03];
  - Risk management. No framework for risk management is offered.
  - Quality management. No framework for quality management is offered.
  - Time management. No framework for time management is offered.
  - Software metrics. No milestone or metric are furnished.
  - CASE-Tool. No CASE-tool has specifically been developed for ADELFE.

4.4.4 MASSIVE

The ***Multiagent Systems Iterative View Engineering*** (*MASSIVE*) development method [Lind01] is a pragmatic framework for the development of multiagent systems. It is:

- Requirements-driven: in this type of approach a task hierarchy is done without assumption on any technology. Technological issues are considered after the task hierarchy's building and based on objective decision functions in order to find the most appropriate technologies;
- View-oriented: this type of method deals with the system as a whole and uses different perspectives on the entire system as fundamental abstraction.

The system is not decomposed but rather viewed from different angles with different focus on particular aspects;

- Iterative: the different stages of the software development are repeated iteratively.

4.4.4.1 Phases

Since MASSIVE is view-oriented rather than model-oriented, it is decomposed in different complementary views rather than in sequential phases. Each of theses views are depicted in this section.

In the **task view**, functional requirements of the target system are analyzed and a task hierarchy used to determine the basic problem solving capabilities of the system's entities is generated. Non-functional requirements of the target system are also defined and quantified as much as possible. MASSIVE is view-oriented so that this view does not assume agent technology and models remain high level.

In the **environment view**, the environment of the target system is analyzed from the developers' perspective and from the systems perspective. Their vision usually greatly differ since the developer has usually global knowledge whereas the system only has local knowledge.

The **role view** determines the functional aggregation of the basic problem solving capabilities according to the physical constraints of the target system. A role is an abstraction that links the domain dependent part of the application with the agent technology that is used to solve the problem under consideration. In this view, an agent consists of one or more role descriptions and an architecture that is capable of executing these role models which makes it important to aggregate the basic capabilities according to physical constraints.

Interaction is the fundamental concept within a system that is comprised of multiple independent entities that must coordinate themselves in order to achieve their individual as well as their joint goals. In the **interaction view**, interaction within the target system is seen as a generalized form of conflict resolution that is not limited to a particular form such as communication. Instead, several generic forms of interaction that can be instantiated in a wide variety of contexts are discussed and the developer is encouraged to analyze the target problem with respect to the applicability of these generic forms before designing new forms.

A society is a structured collection of entities that pursue a common goal. The goal of the **society view** is to classify the society that either pre-exists within the organizational context of the system or that is desirable from the point-of-view of the system developer. According to this classification and well defined quality measures for the performance of the target society that depend on application specific aspects, a society model is developed that is consistent with the roles within the society and that achieves the defined goals.

The **architectural view** is a projection of the target system onto the fundamental structural attributes with respect to the system design. The major aspects that are dealt with in this view are the system architecture as a whole and – due to the size and complexity of this particular aspect – the agent architecture. The system architecture is described according to various aspects and includes things such as the agent management or database integration. The required agent architecture is characterized according to the requirements of the problem to be solved; it is strongly recommended that the system developer should at first try to select one of the numerous existing architectures before trying to develop a new architecture from scratch.

The **system view**, finally, deals with aspects that affect several of the other views at the same time or even the system as a whole. The system view, for example, handles the user interface that controls the interaction between the system and the user(s) on a task specific as well as the task independent level. Generally speaking, the task specific aspects are usually the input specification and the output presentation whereas the task independent aspects deal with the visualization of the system activities in order to enable the user to follow the ongoing computations and interactions. Other aspects that are treated in this view are the system-wide error-handling strategy, performance engineering and the system deployment once it has been developed.

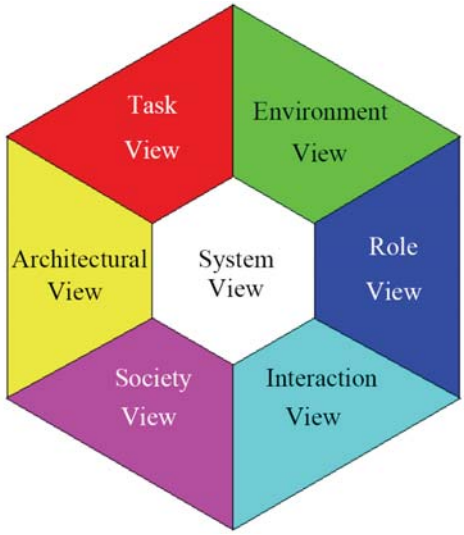


Figure 4-5. MASSIVE Views.

4.4.4.2 Scope

Massive is the most interesting MAS software development methodology in an iterative perspective because it uses an *Iterative View Engineering* approach itself based on *Iterative Enhancement* and *Round-trip Engineering*. The overall process

model contains itself several small micro models that are used for individual tasks within the design process for a particular view. The advantages of this life-cycle model are that it can deal with an incomplete problem specification because of the Iterative Enhancement approach. The model is also useful for early risk detection because parts of the model are incrementally implemented and can direct the project managers view to critical regions.

The micro models as well as the overall process model are in not fixed for the entire lifetime of the project model. Just like the views discussed earlier, they are also subject to changes and refinements during the course of time. In order to preserve these adaptations and to make them accessible to others, the process model and the product model are both embedded into a larger organizational structure called the *Experience Factory*. The Experience Factory provides the formal framework for a permanent learning process over project boundaries that eventually captures the multiagent experience of organization in terms of specific product and process models for various domains.

4.4.4.3 Methodology Evaluation: Summary

In this section the methodology's is evaluated on the basis of the criteria defined framework earlier in this chapter.

- Engineering concepts:
    - Process cutting. Process divided in views: the task, environment, role, interaction, society, architectural and system views. Note that each of these views could be related to a classical SE discipline;
    - Process lifecycle. Iterative lifecycle depicted in [Lind01], a basic support framework is provided: the system view.
    - Covered stages. Deployment stage is not covered;
    - Stages scope.
        - The environment view is used for organizational modelling, the task view for user requirements representation;
        - The role, interaction, society and architectural views are concerned with system design.
- Project management concepts:
    - Process description. [Lind01] furnishes a process description using textual and formal definitions rather than using a method engineering standard;

- Risk management. No framework for risk management is offered. About that topic, [Lind01] quotes "*that this topic is closely related to the optimistic assumption that everything will go all right*".

- Quality management. Quality evaluation and measuring is a process step.

- Time management. The system view can be considered as a basic framework for time management since it manages the contextual changes and the knowledge evolutions among iterations.

- Software metrics. No milestone or metric are furnished.

- CASE-Tool. No CASE-tool has specifically been developed for MASSIVE.

4.4.5 Methodologies Comparison

Table 4-1 summarizes the three methodologies main characteristics. Lessons taken from that comparison are:

- Massive is view oriented, however each of its views can be seen as part of a discipline, more precisely:

  - The task and environment views belong to analysis;

  - The role, interaction, social and architectural views belong to design;

  - Following [Lind01], the system view belongs to both analysis and design. However on the basis of the view scope we suggest to consider it as an independent support discipline since it manages aspects of the other views as well as strategic project issues.

- MaSE has lowest analysis coverage since it assumes that development starts off with all requirements gathered, other methodologies study both the organizational environment and user requirements at analysis stages;

- The most advanced agent-oriented analysis framework is used by the Tropos methodology and are the i* strategic dependency and strategic rationale diagrams. Other methods are based on use-case models or very basic custom models;

	Tropos	MaSE	ADELPHE	MASSIVE
Engineering concepts				
Process cutting	Disciplines	Disciplines	Disciplines	Views
Covered stages				
Analysis	Yes	Yes	Yes	Yes
Design	Yes	Yes	Yes	Yes
Implementation	Yes	Yes	No	Yes
Test	No	No	No	Yes
Deployment	No	No	No	No
Stages scope				
Analysis	Full	Basic	Full	Full
Design	Full	Full	Full	Full
Analysis Models				
Custom Agent-Oriented	i* SD/SRM	Goal model	No	Task tree
UML Use Cases	No	Yes	Yes	No
Process lifecycle	Waterfall	Waterfall	Iterative	Iterative
Project management concepts				
Process description	Basic	Average	Advanced	Basic
Risk management				
At model level	Yes	No	No	No
At process level	No	No	No	No
Quality management				
At model level	Yes	No	No	No
At process level	No	No	No	Yes
Time management				
At process level	No	No	No	Yes
Software metrics	No	No	No	No
CASE tool	Yes	Yes	No	No

Table 4-1. MAS Development Methodologies Comparison.

- Tropos and MaSE are purely waterfall, ADELPHE is said to use the RUP template but does not offer a framework to manage such development, MASSIVE is iterative and uses round trip engineering with a basic support framework called the system view;

- ADELPHE has the most advanced process description since it is the only making use of a method engineering standard (which is SPEM), other methodologies descriptions remain textual with possibly formal definitions;

- MASSIVE is the only methodology offering a framework to support iterative development, even if this framework remains basic.

4.5 Towards an Iterative AOSE Methodology

The previous section showed that most of the existing AOSE methodologies only define engineering stages and required development artefacts without formally or semi-formally defining a way to manage the software process. The goal of this dissertation is to fill up this gap with the I-Tropos methodology. This section exposes the choice of the method to be extended (Tropos), the I-Tropos design process, the selected method engineering standard, a comparison between Tropos and I-Tropos and finally the further possible evolutions of the process.

4.5.1 Theoretical foundations

The process developed into this thesis is based onto the Tropos methodology. It was chosen to extend Tropos rather than other MAS development methodologies because:

- Tropos already offers the most complete coverage of both analysis and design stages even if no test discipline is defined as for example in MaSE;

- Tropos uses the i* strategic dependency (SDM) and strategic rationale models (SRM) at analysis stages; those models define and use organizational elements (actors, agents, goals, objectives, responsibilities, social dependencies, etc.) as fundamentals to conceive systems through all the development process. The SDM and SRM are consequently particularly well indicated for analysis in MAS development (better than traditional UML use case models as in MaSE) and the use of their elements to drive the software process is of particular interest;

- Lots of work has been done to develop the two design disciplines in further details as for example in [DKP03, FKWA06]; furthermore organizational styles and design patterns enrich the methodology.

4.5.2 Designing the Process

Since I-Tropos is an extension of Tropos, designing the I-Tropos process requires documenting the existing disciplines, making them evolve for iterative practicing, adding new disciplines to cover the whole vertical life cycle and determine the phases determining the horizontal dimension.

First of all, the vertical life cycle is made of 7 traditional SE disciplines and 4 support (PM) ones, they have been built using the following criteria:

- The Organisational Modelling and Requirements Engineering workflows are largely inspired by Tropos Early and Late Requirements. The disciplines were designed on the basis of the information found in [CKM02];

- The Architectural Design workflow is based on the equivalent Tropos discipline. This discipline was designed on the basis of the information found in [Faulkner04];

- The Detailed Design workflow is based on the equivalent Tropos discipline. This discipline was designed on the basis of the information found in [Do05];

- The Implementation workflow is based on the equivalent Tropos discipline. This discipline was designed on the basis of the information found in [CKM02] and on the RUP knowledge base [IBM03];

- The Test and Deployment workflows had to be designed from scratch. At highest abstraction level (the WorkDefinition level, see [OMG05]) they follow the same pattern: *defining what to do*, *applying the policy* and *evaluating the results*. Further documentation has also been found in the RUP knowledge base [IBM03, Kruchten03];

- The Risk, Quality and Time Management workflows had to be designed from scratch. At highest abstraction level (the WorkDefinition level, see [OMG05]) they follow the same pattern: *defining/identifying elements*, *acting to deal with those elements* and *evaluating the results*. Further documentation has also been found in the RUP knowledge base [IBM03, Kruchten03];

- Finally, the Process Management workflow had to be designed from scratch. Since this discipline is concerned with the software process itself it is not applied on the same level as the previously described. It has been split in Software Process Tailoring and Software Process Improvement. The first only defines the tailoring rules at two levels of granularity, the second is in charge with the identification of improvement elements and their integration into the software process.

The I-Tropos phases (horizontal dimension) are largely inspired by the phases defined by the Rational Unified Process since it takes over its iterative template.

4.5.3 Documenting the Process

Designing a software process requires a language/formalism to provide a unified readable documentation. While designing I-Tropos, a method or set of models to represent the software process into different dimensions is required. Among the eventualities for the software process description, we envisaged:

- The i* framework itself. This framework is unfortunately not designed for method engineering; we nevertheless could envisage, to extend the proposition depicted in [WAK08] (see Appendix A) with inheritance of traditional elements to method engineering elements. The framework could then be used for such purpose since it would provide graphical representation at process level, offer static and dynamic views and be flexible enough to provide multiple abstraction levels;

- Method engineering approaches. Brian Henderson-Sellers listed the major method engineering methodologies in [Henderson-Sellers06], those methodologies are:

    - The Software Process Engineering Meta-Model (SPEM) "*is a meta-model for defining software development processes and their components*" [OMG05]. SPEM has been designed for documenting methodologies and software processes, its elements are defined in a generic way rather than for particular domains.

    - The Object-Oriented Process, Environment and Notation (OPEN) Process Framework (OPF) [FH02] provides a process environment to create and configure software processes according to the project requirements.

    - ISO/IEC 12207 is an international standard on software lifecycle processes designed to establish a common international framework to acquire, supply, develop, operate, and maintain software [Singh95].

    - ISO/IEC 15504 is a framework for assessment of software processes. Process assessment leads to process improvement and capability determination, it identifies changes to processes and capability determination identifies risks involved in applying the selected process in a project.

    - The Software Process Improvement and Capability dEtermination for Object-Orientation (OOSPICE) is an approach to integrate component-based development with ISO 15504. This approach is

related to process improvement and not directly to process definition, it however provide a process reverence model which defines the process that will be assessed.

The method engineering standards defined above are completely documented in [Sousa08] which also introduces a comparison of the frameworks in terms of:

- Presentability: the meta-model specification has a graphical representation for its elements that is expressive and mapped with the target domain;
- Reusability: the meta-model supports the creation of a knowledge base with many alternatives for selection according to specific needs of organizations and projects;
- Static and Dynamic Views: the meta-model allows users to represent a process both as a static and dynamic structure;
- Flexibility: the meta-model has a good level of abstraction that addresses the definition of method for verying purpose, scope and complexity;
- Standardization: the meta-model is compliant with relevant standards as XML and UML which facilitates integration with other;
- User-Centered: the approach considers the importance of performing activities focused on the user interaction.

[Sousa08] evaluates each of these elements with all the evoked method engineering standards, conclusions are summarized in Table 4-2.

	SPEM	OPF	ISO/IEC 12207	ISO/IEC 15504	OOSPICE
Presentability	Yes	Yes	No	No	No
Reusability	Yes	Yes	Yes	Yes	Yes
Static and Dynamic	Yes	Yes	No	No	Yes
Flexibility	Yes	Yes	Yes	Yes	No
Standardization	Yes	Yes	Yes	Yes	Yes
User-Centered	No	No	No	No	No

Table 4-2. Development Approaches Comparison (adapted from [Sousa08]).

On the basis of the study of the different eventualities, we decided not to define a custom framework to describe the I-Tropos process. Indeed, this would lead to a set of models in adequacy with our process description requirements but would require

defining a set of elements that have already been formalised and depicted in detail into the different method engineering approaches. We rather propose to select one of the method engineering approaches depicted above; for this purpose we evaluate the relative importance of each method evaluation element with respect to our approach:

- Presentability is highly important to furnish a visual documentation of the software process. Since one of our goals is to build a MAS development methodology that could be adopted for everyday software developments;
- Reusability is also highly important since the process must be adopted for developments in various domains;
- Allowing to represent static and dynamic views is also highly important to dispose of multiple complementary views of the proposed process;
- Flexibility is highly important for the development of the software process because it has to be adapted for MAS development;
- Standardization is highly important for the process adoption;
- To be user-centred is marginally important since the developed process is based on a goal driven approach rather that on focusing directly on user interaction.

On the basis of table 4-2, one can determine that only the SPEM and the OPF meet all the highly important criteria for the process developed in this dissertation. The SPEM has been chosen to describe our process because:

- it is developed by OMG and has been subject to continuous versions and adaptation. The process has been described using version 1.1 [OMG05], the latest available version is, however, version 2.0 [OMG08];
- it is the standard that is closest to the one used by Rational when describing the RUP so that its adoption by practitioners would be easier;
- it has been used to document other MAS development methodologies as in [CS05, GMP03] so that it constitutes a little bit of a usage in MAS development method description.

4.5.4 Tropos versus I-Tropos: Evolution of a MAS Development Process

I-Tropos represents an evolution of the Tropos process. It constitutes an operationalization of the Tropos methodology in order to be used in large software developments. I-Tropos mainly fills up the gap of project management which is, for now, seldom approached in MAS literature. The main contributions are:

- a meta-level process documentation. Tropos disciplines had been described in terms of models produced without going in further details;

- a full SE life cycle coverage:

    - Early and late requirements disciplines have been rebuilt into, respectively, organizational modelling and requirements engineering;

    - An implementation, test and deployment discipline have been introduced in the process to cover the identified gaps in life cycle coverage;

    - Risk, quality, time and process management disciplines have been introduced to cover the project management aspects of the software process.

- A project management framework for the inclusion of a MAS development methodology in an iterative process template. To better evaluate this contribution, we suggest comparing it with the MASSIVE system view, the only known support framework in MAS development methodologies. I-Tropos PM framework is characterized as follows:

    - Software project development and management are model-driven since they use the the goeals issued from i* SD and SR models as fundamental scope elements. This can hardly be compared with MASSIVE system view since it only reports an iteration experience to the next one and does not introduce a model-driven management approach;

    - Threats and quality factors are used for prioritizing goals' realization so that we dispose of risk and quality management at process level. The MASSIVE system view only focuses on managing its analysis and design stages in an had hoc manner;

    - Time management provides a multi-iterations vision refined at milestones (when iteration ends) while MASSIVE only offers a single iterations management perspective.

Empirical evaluation of the benefits of AOSE iterative development is hard to measure. This could be done by achieving a similar case study with I-Tropos on the one side and with Tropos or other waterfall AOSE methodologies on the other using development teams with similar experience. Such an experience is practically unachievable. However, if it was, we could measure both PM and SE metrics to evaluate the impact of iterative development on each aspect.

First of all, the PM metrics that could be interesting to evaluate are the following:

- The general users' satisfaction towards the developed application;
- Risk management: for each of the I-Tropos identified risks, we could measure quantifiably users' satisfaction on the functions and properties concerned with the particular threat; measure the threat impact in case of occurence;
- Quality management: for each of the I-Tropos identified quality factors, we could measure quantifiably users' satisfaction on the functions and properties concerned with the particular quality factor;
- Time management: on a global basis we could compare:
    - Development time;
    - Development costs.

In a more general manner, we could test the I-Tropos process performance by comparing every estimated factor and comparing it to the effectively measured so that a general process performance level could be computed. This would allow to calibrate and to refine the process notably in the process management discipline.

Due to the poorness of literature concerning agent-oriented software metrics evaluating structural complexity, we point to the use of existing object-oriented ones. As a preliminary tests suite, we claim for the use of the metrics proposed by Chidamber and Kemerer in [CK94]. Those include:

- The Depth of Inheritance Tree (DIT). This metric measures the maximum level of the inheritance hierarchy of a class. The root of the inheritance tree inherits from no class and has a DIT count of 0. Chidamber and Kemerer suggest that DIT can be used to indicate the complexity of the design, potential for reuse;
- The Number Of Children (NOC). This metric counts the number of immediate subclasses belonging to a class. NOC was intended to indicate the level of reuse in a system and a possible indicator of the level of testing required;
- The Lack of Cohesion in Methods (LCOM). This metric is intended to measure the lack of cohesion in the methods of a class. It is based on the principle that different attributes occurring in different methods of a class causes that class to be less cohesive than one where the same attribute is used in few methods of the class. It is viewed that a lack of cohesiveness as undesirable as it is against encapsulation. Lack of cohesion could imply that the class should probably be split into two or more subclasses;

- The Weighted Methods per Class (WMC). The sum of the complexities of the methods in a class;

- The Coupling between objects (CBO). The number of other classes whose methods or instance attribute(s) are used by methods of this class;

- The Response for a Class (RFC). The sum of the number of methods in the class and the number of methods called by each of these methods, where each called method is counted once.

4.5.5 Process Evolution and Future Work

I-Tropos lays the foundations of a mature software development process methodology; it covers in depth the whole traditional engineering life cycle and introduces project and process management disciplines. The process could however still be extended at different levels, among those we find:

- A higher level of integration: risk and quality management frameworks evaluate respectively threats and quality factors at process level for iterative development planning. Work has also been done for risk and quality management within the software process i.e. to deal with threats and quality issues at operational/technical level. A systematic approach would cover both aspects on the basis of a unique model or set of models. We suggest this could be achieved by integrating the framework proposed in [WAK08] into the process;

- Dealing with organizational change management. Nowadays organizations are constantly evolving: reorganisations, merges, acquisitions, etc. lead to a need to constantly manage change. I-Tropos in its actual version is better able to deal with such issues than traditional waterfall methodologies because of its iterative nature and organizational modelling capabilities. Nevertheless we suggest that to adequately manage change, one could include a dedicated discipline for a systemic approach of this topic. The interest of such a discipline is of particular interest in our MAS modelling and development method since it uses the i* framework which incorporates organizational elements as fundamentals. Such an approach would be particularity interesting in the context of ERP systems which are software systems continuously evolving with the organizational changes of the environment they are deployed in;

- As explained in this dissertation, the iterative character of a software process allows to make the organizational modelling and requirements engineering stages continuous along the whole software process to better understand the organizational setting and take into account users needs. The iterative nature of the process could, however, also be exploited to deal with the non-deterministic character of software agents. Since agents do not behave deterministically they induce a contextual change during an iteration. Since

this aspect could hardly be forecasted, this could be taken into account thanks to the use of an iterative process allowing to determine a contextual change during an iteration and taking it into account during the next one;

- Since the PM framework offers a certain degree of genericity, it could be adapted for model-driven PM in other MAS development methodologies. This could be easily done in methodologies using use-case modelling as MaSE or ADELPHE but also for traditional object-oriented UML software projects.

4.6 Chapter Summary

This chapter overviewed the multiple aspects of the I-Tropos development process. AOSE has firstly been overviewed; the distinction between SE and PM has then been made. A multiple-criteria framework has been built to evaluate some existing AOSE methodologies in the perspective of iterative software development. Tropos, MaSE, ADELPHE and MASSIVE have then been evaluated on the basis of the framework. Even if most of them advise to proceed iteratively, only MASSIVE furnishes a basic framework to support this type of development. Such a framework, often called project management, is nevertheless mandatory for effective iterative software project development. After that, the building process of I-Tropos is exposed, this encompasses the process' theoretical foundations, design, documenting, a comparison with I-Tropos and finally overviews the evolutions and future work.

Now that the foundations of the process developed in this dissertation have been exposed and that it has been positioned between the other methodologies, I-Tropos can be documented. This will be done into the next chapter.

References

[AUML] FIPA. “Agent UML”. http://www.auml.org/.

[BCM03] L. Bastos, J. Castro and J. Mylopoulos. “Integrating Organizational Requirements and Socip-Intentional Architectural Styles”. In *Proceedings of Second International Workshop From Software Requirements to Architectures* (*STRAW’03*), Portland, USA, May 9, 2003.

[BS02] P. Bresciani and F. Sannicol. “Applying Tropos Early Requirements Analysis for defining a Tropos tool”. In *Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002) at CAiSE'02*, *Toronto, Ontario, Canada*, may 2002.

[BGPG02] C. Bernon, M.P. Gleizes, G. Picard and P. Glize. “The Adelfe Methodology for an Intranet System Design”. In *Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002) at CAiSE'02*, *Toronto, Ontario, Canada*, may 2002.

[Bradshaw97] J.M. Bradshaw. “Software Agents”. *AAAI Press/The MIT Press*, 1997.

[Brandon06] D. Brandon. “Project Management for Modern Information Systems”. *IRM Press*, 2006.

[BB01] E. Breton, J. Bézivin. “Model Driven Process Engineering”. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, pp 225-230, IEEE Computer Society, 2001.

[Brinkkemper90] S. Brinkkemper. “Formalization of Information Systems Modelling”. PhD Thesis, University of Nijmengen, Thesis Publishers, Amsterdam, 1990.

[CKM02] J. Castro, M. Kolp and J. Mylopoulos. “Towards A Requirements-Driven Development Methodology : The Tropos Project”. In *Information Systems*, Elsevier, 2002.

[CK94] S.R. Chidamber and C.F. Kemerer.. A Metrics Suite for Object-Oriented Design. In IEEE Transactions on Software Engineering, 20(6), 476-493, 1994.

[CS05] M. Cossentino and V. Seidita. “SPEM description of the ADELFE process”. *Technical report*, *RT-ICAR-PA-05-07*. July 2005.

[DWS01] S. DeLoach, M. Wood and C. Sparkman, “Multiagent Systems Engineering”. In *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, no.3. pp. 231-258, 2001.

[Do05] T. T. Do. “A Framework for Multi-Agent Systems Detailed Design”. Ph.D. Thesis, Université catholique de Louvain, Institut d’Administration et de Gestion (IAG), Louvain-la-Neuve, Belgium, July 2005.

[DKP03] T. T. Do, M. Kolp and A. Pirotte. "Social Patterns for Designing Multi-Agent Systems". In *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, San Francisco, USA, July 2003.

[Faulkner04] S. Faulkner. "An Architectural Framework for Describing BDI Multi-Agent Information Systems". Ph.D. Thesis, Université catholique de Louvain, Institut d'Administration et de Gestion (IAG), Louvain-la-Neuve, Belgium, May 2004.

[FKWA07] S. Faulkner, M. Kolp, Y. Wautelet and Y. Achbany. "A Formal Description Language for Multi-Agent Architectures". In *Proceedings of the 17th International Workshop on Agent-Oriented Information Systems*, Hakodate, Japan, 2006.

[FH02] D.G. Firesmith and B. Henderson-Sellers. "The OPEN Process Framework. An Introduction", *Addison-Wesley*, 2002.

[GHW04] P. Giorgini, B. Henderson-Sellers and M. Winikoff. "Agent-Oriented Information Systems". In *Proceedings of the 5th International Bi-Conference Workshop* (*AOIS*), Australia, July 14, 2003 and Chacago, IL, USA, Octover 13th, 2003, Revised Selected Papers Springer, 2004.

[GKMP04] P. Giorgini, M. Kolp, J. Mylopoulos and M. Pistore, "The Tropos Methodology: An Overview". In *Methodologies and Software Engineering for Agent Systems*, Kluwer, 2004.

[GMMZ06] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. "Requirements Engineering for Trust Management: Model, Methodology, and Reasoning". In *International Journal of Information Security,* 5(4):257-274, October 2006.

[GMP03] M.-P. Gleizes, T. Millan and G. Picard. "ADELFE: Using SPEM Notation to Unify Agent Engineering Process and Methodology". *Rapport interne IRIT n° IRIT/2003-10-R*, June 2003.

[Henderson-Sellers06] B. Henderson-Sellers. Method Engineering: Theory and Practice. ISTA'2006, 2006 , pp.13-23.

[HSMGD04] B. Henderson-Sellers, M. Serour, T. McBride, C. Gonzalez-Perez, L. Dagher. "Process Construction and Customization". In *Journal of Universal Computer Science*, vol. 10, no. 4, 2004.

[i*wiki] i* Wiki. "i* Available Tools". Available at http://istar.rwth-aachen.de/tiki-index.php?page_ref_id=21

[IBM03] IBM. "The Rational Unified Process. Version 2003.06.00.65". *Rational Software Corporation*, 2003.

[IGG99] C. Iglesias, M. Garijo and J. Gonzales. "A survey of agent-oriented methodologies". *Intelligent Agents IV*, *Lecture Notes in Computer Science*, Springer-Verlag 1555, 1999.

[JW99] N. R. Jennings and M. Wooldridge, "Agent–Oriented Software Engineering". In *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi Agent System Engineering* (*MAAMAW-99*), Valencia, Spain, 1999

[KGM06] M. Kolp, P. Giorgini and J. Mylopoulos, "Multi-Agent Architectures as Organizational Structures", In *International Journal of Autonomous Agents and Multi-Agent Systems* (*JAAMAS*), Kluwer, 2006.

[Kruchten03] P. Kruchten. "The Rational Unified Process: An Introduction". *Addison-Wesley*, 2003.

[Lind01] J. Lind. "Iterative Software Engineering for Multiagent Systems: the MASSIVE Method". *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*, Vol. 1994, Springer-Verlag, 2001.

[MMMDHG08] R. Matulevicuis, N. Mayer, H. Mouratidis, E. Dubous, P. Heymans and N. Genon. "Adapting Secure Tropos for Security Risk Management in the Early Phases of Information Systems Development". In *Proceedings of the International Conference on Advanced Information Systems Engineering* (*CAiSE'08*), Lecture Notes in Computer Schience, 541-555, 2008.

[OMG05] OMG, "The Software Process Engineering Meta-model Specification. Version 1.1", January 2005.

[OMG08] OMG, "Software and Systems Process Engineering Meta-model Specification. Version 2.0", April 2008.

[Singh95] R. Singh. "Introduction to International Standard ISO/IEC 12207: Software Life Cycle Processes", tutorial slides, 1995.

[Sousa] K. S. Sousa. "Towards Method Engineering of Model-Driven User Interface Development", DEA Thesis, Université catholique de Louvain (UCL), Louvain School of Management (LSM), Louvain-la-Neuve, Belgium, 2008.

[WAK08] Y. Wautelet, Y. Achbany and M. Kolp. "A Service-Oriented Framework for MAS Modeling". In *Proceedings of the the 10th International Conference on Enterprise Information Systems* (*ICEIS 2008*), Barcelona, 2008.

[WJK00] M. Wooldridge, N.R. Jennings and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design". In *Autonomous Agents and Multi-Agent Systems*, 3(3): 285-312, 2000.

[Yu95] E. Yu. "Modeling Strategic Relationships for Process Reengineering". *PhD thesis*, *University of Toronto*, *Department of Computer Science*, Canada, 1995.

5 I-Tropos: Software Process Description

Thanks to the latest developments of agent-oriented methodologies the whole waterfall software development life cycle (SDLC) of an agent-based application can today be covered. These approaches highlight the fact that the agent-oriented research program fills up the conceptual gaps between previous research programs such as object orientation and the organization being modeled. It also offers tools for organization modeling and requirements engineering and allows the analyst to deal with the inherent and increasing complexity of software applications. Nevertheless, classical *Tropos* and other agent-oriented methodologies do not cover yet all aspects of the software engineering lifecycle depicted in iterative methodologies such as the *Unified Process*. Obviously, the advantages of an iterative software engineering process, such as efficient software project management, continuous organizational modeling and requirements acquisition, early implementation, continuous testing and modularity should be applied in the development of agent-oriented software. In this context, the aim of our research is to extend the waterfall SDLC of *Tropos* into an iterative one in order to offer these advantages to MAS development. This new methodology will be called *I-Tropos*, its process is overviewed in this chapter. Section 5.1 studies i* elements and refines some of their definitions in order to find the most appropriate scope element to drive the I-Tropos software process. Section 5.2 overviews the process characteristics; finally, Sections 5.3 and 5.4 describe respectively the core and support disciplines in a generic way.

5.1 Model-Driven Development: an i* Approach

Some modern software development methodologies are defined as *model driven*. In such methods, the entire development process is deduced from the features modeled in highest level diagrams. Object-oriented development methodologies such as the Unified Process [IBM03, JBR99, JB00, Kruchten03] are for example said to be *use case driven*. This means that the entire development process is driven and planned by use cases identified in the analysis disciplines. Moreover, techniques like *Use-Case Points* [SW01] directly use those high level diagram elements as fundamentals for effort estimation.

Such model elements, called *scope elements*, are consequently useful not only to share a common high level vision between stakeholders, but also to estimate the project effort on a non redundant basis, to evaluate related risks, to fix an acceptable level of quality, etc. Tropos developments using traditional Strategic Dependency (SD) and Strategic Rationale (SR) diagrams [Yu95] for organizational modeling do not unfortunately possess the ideal elements to drive the development process. This section overviews each i*/Tropos element to find the most suitable element to drive the software process. To cope with deficiencies, an extended goal definition is proposed in section 5.1.3.

5.1.1 i* Weaknesses: Related Work

As pointed out earlier, considerable work has been done in i*/Tropos and other agent-oriented software development methodologies. The traditional i* modeling framework has however been poorly criticized and weaknesses as the goals adequate abstraction level or the alternative modeling possibilities have only been recently pointed out in literature [ERMP06, PEM08]. Similarly, considerations on software project management using Tropos or other MAS methodologies do not constitute an overwhelmed aspect of SE literature.

Methodological foundations and revised i* rules have been based on the weaknesses of the i* approach addressed in [ERPM06, PEM08]. Those papers define a series of features for evaluating the i* framework of case studies. The empirical study addresses the weaknesses of the approach on the basis of an evaluation framework including aspects as refinement, modularity, repeatability, complexity management, expressiveness, traceability, reusability, scalability and domain applicability. To address the identified weaknesses they point to the use of higher level abstractions called business services into i* developments. This approach is taken as starting point in [WAK08] (the paper is available in Appendix A). Indeed, Wautelet et al. use the idea of a service level diagram with non-overlapping services but supplement it to become a three diagram framework furnishing three complementary views for MAS analysis. This framework has the advantage of offering a high level aggregate view in which the elements can be used as fundamental for driving the software process. Those elements are refined in two views, one static made of the traditional i* strategic dependency and strategic rationale diagrams and another dynamic showing the atomic steps for service realization. This framework could have been adopted by I-Tropos, however in this dissertation we point to the refinement of the goal definition and i* modeling method to incorporate the advantages of the service concept as depicted in [ERPM06, PEM07, WAK08] to traditional SD/SR modeling.

5.1.2 i* Elements Evaluation

This section studies each possible element in i* SD and SR diagrams to seek for an existing fitting scope element.

The Tropos software development methodology uses the i* framework for organizational and requirements modeling. Two kinds of diagrams are envisaged for such modeling activities:

- The Strategic Dependency Model (SD) describes a network of dependency relationships among various actors in an organizational context. Actors are usually identified within the context of the model. This model shows who an actor is and who depends on the work of an actor. An SD consists of a set of nodes and links connecting actors. Nodes represent actors and each link represents a dependency between two actors. The depending actor is called *Depender* and the actor who is depended upon is called the *Dependee*.

- The Strategic Rationale Model (SR) provides an intentional description of processes in terms of process elements and the rationale behind them. While the Strategic Dependency (SD) model maintains a level of abstraction by modeling only the external relationships among actors, the SR model forgoes that abstraction in order to allow a deeper understanding about strategic actors' reasoning about processes to be explicitly expressed. The SR model describes the intentional relationships that are "internal" to actors, such as means-ends relationships that relate process elements, providing explicit representation of "why" and "how" and alternatives. The rationales are at strategic level, so that the process alternatives being reasoned about are strategic relationships, i.e., SD configurations. Using knowledge represented in and organized by these modelling concepts, process alternatives can be systematically generated and explored, so as to help actors find new process designs that better address their interests, needs, and concerns. The SR model is a graph, with several types of nodes and links that work together to provide a representational structure for expressing the rationale behind processes.

Actors of the SD and SR models are involved in dependencies of different intentional elements; we will briefly describe each element and explain why none of these are good candidates for driving the development process.

- The Resource dependency.
    - In a Resource dependency, one actor (the depender) depends on the other (the dependee) for the availability of an entity (physical or informational). By establishing this dependency, the depender gains the ability to use this entity as a resource. At the same time, the depender becomes vulnerable if the entity turns out to be unavailable. Under resource dependency, the issue of decisions (who takes decision for “achieving” the dependum) does not come up. A resource is the finished product of some deliberation-action process. It is assumed that there are no open issues or decision to be addressed;
    - Resources represent entities handled by actors, they do not represent user requirements or organizational processes, but just intervene during their achievement. Resources do consequently not represent adequate elements for driving the software process;
- The Softgoal dependency.
    - In a Softgoal dependency, a depender depends on the dependee to perform some task that meets a softgoal. The meaning of the softgoal is specified in terms of the methods that are chosen in the course of pursuing the goal. As in the goal dependency, a depender gains the ability of having the goal condition brought about, but be-

comes vulnerable in case the dependee fails to bring on that condition. The difference here is that the conditions to be attained are elaborated as the task is performed;

- Softgoals represent non-functional requirements, they are consequently not directly implemented but do have an influence on the way functional requirements are. Softgoals do consequently not represent adequate elements for driving the software process;

- The Task dependency.
  - In a Task dependency, the depender depends on the dependee to carry out an activity. A Task dependency specifies how the task is to be performed, but not why. The depender is vulnerable since the dependee may fail to perform the task. Task specifications should be viewed as constrains rather than as complete (and therefore adequate) know-how for performing the task. This is one reason why a dependee may fail in performing the task. Another reason may be that the dependee decides not to perform the task even when it is able to, e.g. if it decides there are more important things to do (which may be due to other commitments). Under task dependency, the depender makes the decisions. The depender's goal are not given to the dependee;
  - Tasks represent functional activities agents perform. Using this element as scope for driving the software process is problematic since it should represent a rather atomic set of actions and the same features can be modeled as goals by some analysts or as tasks by others. Moreover, tasks are commonly represented in the context of a goal which has a higher level of abstraction. Tasks do consequently not represent the ideal elements for driving the software process;

- The Goal dependency.
  - In a Goal dependency, the depender depends on the dependee to bring about a certain state in the world. The dependee is given the freedom to choose how to do it. With a goal dependency, the depender gains the ability to assume that the condition or state of the world will hold, but becomes vulnerable since the dependee may fail to bring about that condition. Under goal dependency, the dependee is free to, and is expected to, make whatever decisions are necessary to achieve the goal (the dependum);
  - The fulfillment of a goal supposes a functional behavior of an agent. A goal is the best candidate for driving the software process. Indeed it represents the highest level feature of the SD and SR dia-

grams so that it has a more strategic dimension so that they can raise a higher level of abstraction and aggregation. However, the goal as defined in [Yu95] addresses problems to be the element for driving the software process since:

- The atomicity of the goal is not clearly defined; moreover, goal decomposition can involve goals included into others so that:
  - a great amount of goals – too much for keeping an easily understandable vision of the software project – can be represented in i* diagrams of huge software projects;
  - goals are often at different levels of atomicity so that they can be totally or partially overlapping which is a major drawback to keep the simple goal-based vision.
- The distinction between a task and a goal is not trivial, so that different persons can model similar behavior by a goal or by a task.

5.1.3 Extending the i* Goals and Tasks Definition

On the basis of the discussion of the previous section, two alternatives can be envisaged to dispose of non-overlapping scope elements located at higher abstraction level allowing to drive the software process:

- to define a new i* element/diagram at higher abstraction level allowing to represent a service the agents should provide in a non redundant way. This proposal was made into [ERPM06, PEM07] and supplemented in [WAK08];
- to refine existing i* goals and tasks definitions in such a way that software modelers use goals in a non-overlapping way and decompose them as tasks. Consequently goal elements are used only to represent the highest level business processes of the project; lower level processes included in the goals are modeled as tasks. This avoids redundancy/overlapping of goal elements; the set of the i* goals encompass the whole business processes and user requirements that should be developed into the software application.

The formal rules depicted into this section illustrate those concepts; the class diagram of Figure 5-1 is a refinement of the meta-model of the SD and SR models of [SPGM05] including new concepts.

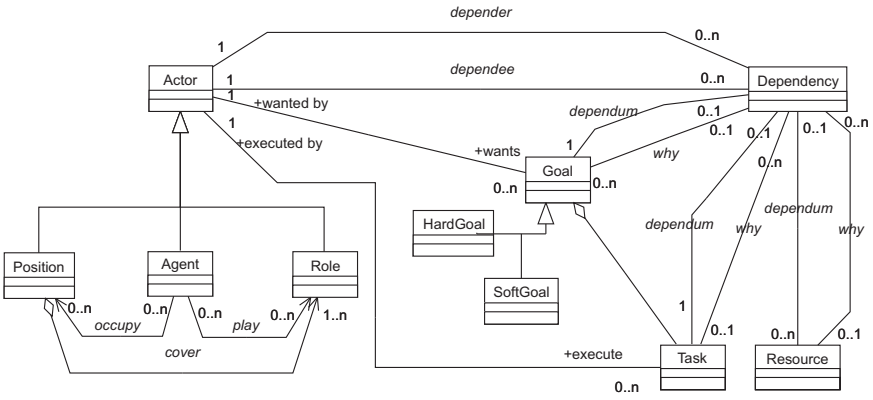


Figure 5-1. Strategic Dependency and Strategic Rationale Diagrams: a Meta-Model

Definition 1 *A task tk_i is $\langle tk_i^{pre}, \tau_i, tk_i^{post} \rangle$ where tk_i^{pre} represents the task preconditions (including the tasks that should have been performed before), τ_i is the functional specification of the task performed by an agent, tk_i^{post} describes the task postconditions. Tasks belong to the set TK representing all the project tasks.*

Definition 2 *$\langle a_j^{der}, a_j^{dee}, TK_j \rangle$ is a goal gl_j, where a_j^{der} represents the depender agent and, a_j^{dee} the dependee. TK_j is the subset of TK, it represents the tasks required to fulfil goal gl_j. Goals belong to the set GL representing all the project goals such that: $gl_i \in GL$.*

An additional constraint is required to illustrate the fact that goals are not overlapping business processes.

Constraint 1 *$\forall\, gl_i \in GL$ where gl_i is a specific goal of the project and GL is the set of goals of a particular project : $gl_i \cap gl_j = \emptyset$.*

This last constraint expresses the fact that goals can be decomposed into a succession of tasks.

Constraint 2 *$\forall\, gl_i \in GL$ where gl_i is a specific goal of the project and GL is the set of goals of a particular project : $gl_i = tk_i \cup tk_{i+1} \cup \ldots \cup tk_{i+n}$.*

5.2 Process Characteristics

In this section we firstly overview the *Software Process Engineering Metamodel* (SPEM) used to model the I-Tropos software process. In the next section, the process engineering concepts and the process phases are introduced. Finally, a brief discussion on pattern-oriented development takes place.

5.2.1 The Software Process Engineering Metamodel (SPEM)

SPEM is an OMG specification of an UML metamodel (and UML profile). It is used to represent a family of software development processes and their components [OMG05]. It constitutes a sort of "ontology" of software development processes. SPEM provides the minimum set of process modeling elements to describe any software development process without adding specific models or constraints for any specific area or discipline, such as requirements engineering or project management.

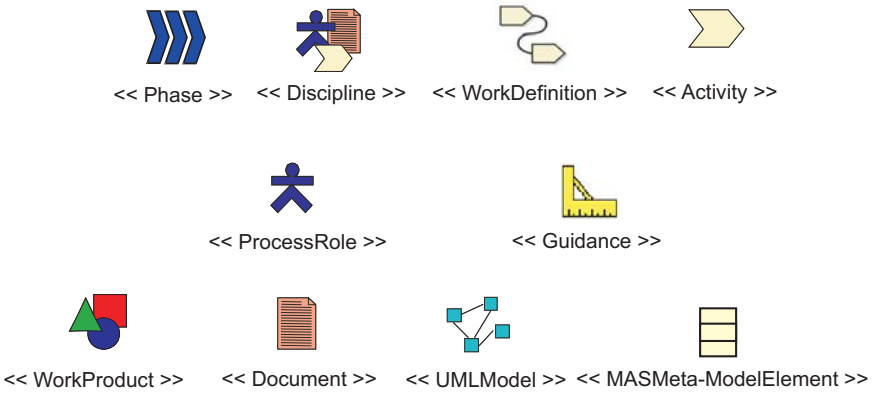


Figure 5-2. Some SPEM Icons

A full description of SPEM including each concept described below and their hierarchy can be found in [OMG05]. Relevant SPEM concepts used in this dissertation are (Figure 5-2 depicts, when existing, their related icons):

- *WorkDefinition*: it constitutes a kind of operation that describes the work performed in the process;
- *Phase*: it constitutes a specialization of the WorkDefinition such that its precondition defines the phase entry criteria and its goal (often called a "milestone") defines the phase exit criteria;
- *Iteration*: it constitutes a composite WorkDefinition with a minor milestone;
- *WorkProduct*: it constitutes an artifact (any tangible piece of information that is produced, consumed or modified by a process) of a process;
- *ProcessRole*: it defines responsibilities over specific WorkProducts;
- *Activity*: it constitutes a piece of work performed by a single ProcessRole;
- *Discipline*: it partitions the Activites within a process according to a common "theme";

- *ModelElement*: it constitutes an element describing one aspect of a software engineering process;
- *Guidance*: it constitutes an element aimed to provide more detailed information about the associated ModelElement;
- *Document*: it constitutes a stereotype ("a special kind") of WorkProduct;
- *UMLModel*[1]: it constitutes a stereotype ("a special kind") of WorkProduct;
- *MASModelElement*: it constitutes a stereotype ("a special kind") of WorkProduct (this element is not refereed to in [OMG05] but has been introduced and defined in [CSS03].

5.2.2 The Process Engineering Concepts

An "I-Tropos development" is made of disciplines[2] iteratively repeated while the relative effort spend on each one is variable from one iteration to the other. The Organizational Modeling and Requirements Engineering disciplines are respectively largely inspired by Tropos' Early and Late Requirements disciplines. The Architectural and Detailed Design disciplines correspond to the same stages of traditional Tropos. I-Tropos includes core activities i.e. *Organizational Modeling*, *Requirements Engineering*, *Architectural Design*, *Detailed Design*, *Implementation*, *Test* and *Deployment* but also support disciplines to handle the project development called *Risk Management*, *Time Management*, *Quality Management* and *Software Process Management*. All of them are overviewed in this Chapter. There is little need for support activities in a process using a waterfall SDLC since the core disciplines are sequentially achieved one for all. When dealing with a process using an iterative SDLC, the need for support disciplines for managing the whole software project is from primary importance. Figure 5-3 presents I-Tropos process' disciplines as a SPEM package

[1] Note that this stereotype will be used in this paper to represent i* [Yu95], AUML [AUML], etc. models which are not strictly speaking models referring to the Unified Modeling Language [UML].

[2] The phase and discipline notions are often presented as synonyms in software engineering literature. In [CKM02], Tropos is described as composed of five phases (Early Requirements, Late Requirements, Architectural Design, Detailed Design and Implementation). However [OMG05] defines disciplines as "*a particular specialization of Package that partitions the Activities within a process according to a common "theme".*", while the phase is defined as "*a specialization of WorkDefinition such that its precondition defines the phase entry criteria and its goal (often called a "milestone") defines the phase exit criteria*". In order to be compliant with the most generic terminology (the one described in the SPEM), traditional Tropos phases will be called disciplines in our software process description since they partition Activities under a common theme. In the same way, phases will be considered as groups of iterations which are workflows with a minor milestone. The same distinction is made into the *Unified Process*.

diagram, a detailed specification of those disciplines can be found later in this chapter as well as in [WKA06].

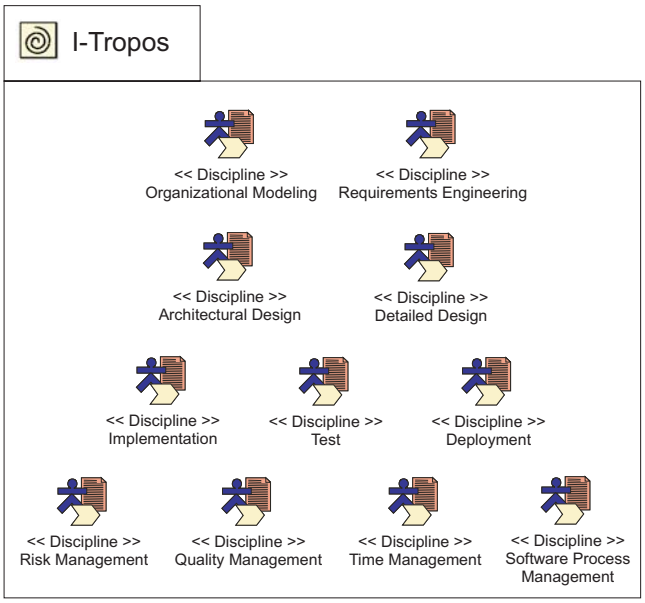


Figure 5-3. The I-Tropos Process Disciplines Package Diagram

As mentioned previously, using an iterative SDLC implies repeating process' disciplines many times during the software project. Each iteration belongs to one of the four phases inspired by the Unified Process (UP); a complementary documentation can be found in [Kruchten03] while a summary of each phase objective is depicted into the next section. These phases are achieved sequentially and have different goals evaluated at milestones through knowledge and achievement oriented metrics, those are informally described into the next section. Figure 5-4 offers a two dimensional view of the I-Tropos process: it shows the disciplines and the four different phases they belong to.

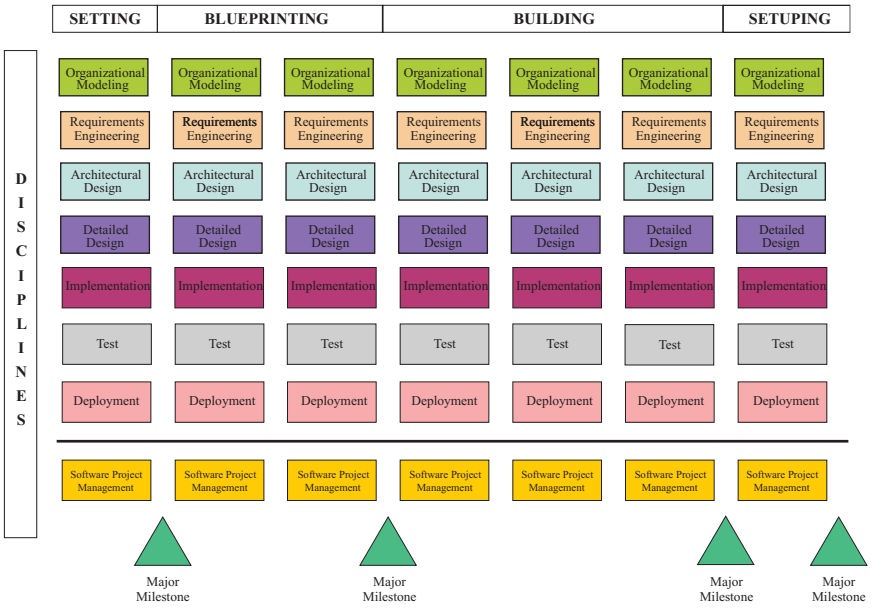


Figure 5-4. I-Tropos: Iterative Perspective

5.2.3 The Process Phases

I-Tropos phases are inspired by the UP phases[3]; each one is made of one or more iterations. Discipline are gone through sequentially; as stressed before the phases are separated by major milestones. Each of them has its own goal:

- The *setting* phase is designed to:
  - identify and specify most stakeholders requirements: the purpose of this phase is to identify up to 95 % of stakeholders requirements;
  - have a first approach of the environment scope: what the system has to cover;
  - identify and evaluate project's threats: risky features are assessed, prioritized and planned for practical resolution;
  - identify and evaluate quality factors: quality dependent features are assessed, prioritized and planned for practical resolution.

[3] The phases milestones expressed hereafter are based on the metrics expressed in the Rational Unified Process (see [Kruchten03]).

- The *blueprinting* phase is designed to:
    - Produce a consistent architecture for the system on the basis of the identified requirements: the purpose of this phase is to produce a design at 95 % stable;
    - Eliminate most risky features in priority: features risk is assessed during each iteration and the practical resolution planning is evaluated continuously;
    - Evaluate blueprints/prototypes to stakeholders; feedback will feed next iterations.
- The *building* phase is designed to:
    - Build a working application;
    - Validate developments.
- The *setuping* phase is designed to:
    - Finalize production: environmental constraints will cause system evolutions;
    - Train users;
    - Document the system.

5.2.4 Pattern-Oriented Development

Consequent effort has been spent to integrate pattern-oriented development into I-Tropos developments. A framework using the catalogue of architectural styles concerned with architectural design has been introduced in [KGM06] while one using social patterns concerned with detailed design has been proposed in [KFW08].

The I-Tropos software process envisages the adoption of patterns principally at two levels:

- The adoption of architectural styles into the architectural design discipline: architectural styles are intellectually manageable abstractions of system structure that describe how system components interact and work together [Shaw95]. In the context of MAS, we are interested in social structures that result from a design process. Guidance can be found in organizational theories, namely *Organization Theory* and *Strategic Alliances*. Organization Theory (e.g., [Mintzberg92, Scott98, YS95]) describes the structure and design of an organization; *Strategic Alliances* (e.g., [DG99, MSB99, Segil96])

models the strategic collaborations of independent organizational stakeholders who pursue a set of agreed upon business goals. A catalogue of organizational architectural styles as well as a framework to select the most appropriate for a particular project has been developed in [KGM06], they take place in the I-Tropos architectural design discipline;

- The adoption of social patterns into the detailed design discipline: social patterns are a set of predefined software micro-architectures including five different complementary dimensions (Social, Intentional, Structural, Communicational and Dynamic) destined to be used/reused on multiple software projects. A catalogue of Social Patterns has been developed in [DKP03, KFWA07] they take place in the I-Tropos detailed design discipline.

The I-Tropos process in its generic description envisages the use of patterns at both architectural and detailed design stages but does not point to the systematic use of it. [SAMC07] and [Silva07] refine the Detailed Design process described in [WKA06] - which is the previous version of the one depicted in this chapter - to systematically include Social Pattern at this development stage. This I-Tropos refinement is outside the scope of this dissertation and adoption into the generic process is left for future work.

5.3 The Process Core Disciplines

The I-Tropos process has been fully described using the Software Engineering Process Metamodel in [WKA06]. That technical report describes each process' *Discipline*, *Activity*, *Role*, *WorkDefinition* and *WorkProduct*, so that it can be used as reference or guide to the methodology[4]. A lightened overview of the process is given in this section.

5.3.1 Organizational Modeling

The Organizational Modeling discipline, strongly inspired from the Tropos *Early Requirements* stage, aims to understand the problem by studying the existing organizational setting. The emphasis is put on understanding the motivation and rationale that underlie system requirements.

As shown in Figure 5-5, two roles are involved, *Organization Modeler* and *Stakeholder*, and there are four WorkProducts produced by this discipline: *Strategic Dependency Model*, *Strategic Rationale Model*, *Stakeholder List* and *Stakeholder Intention List*.

[4] Note that the methodology has subsequently evolved since the edition of the paper so that [WKA06] can be considered has a reference report only for the core disciplines (even if some have changed names). The reference document for the support disciplines is this thesis.

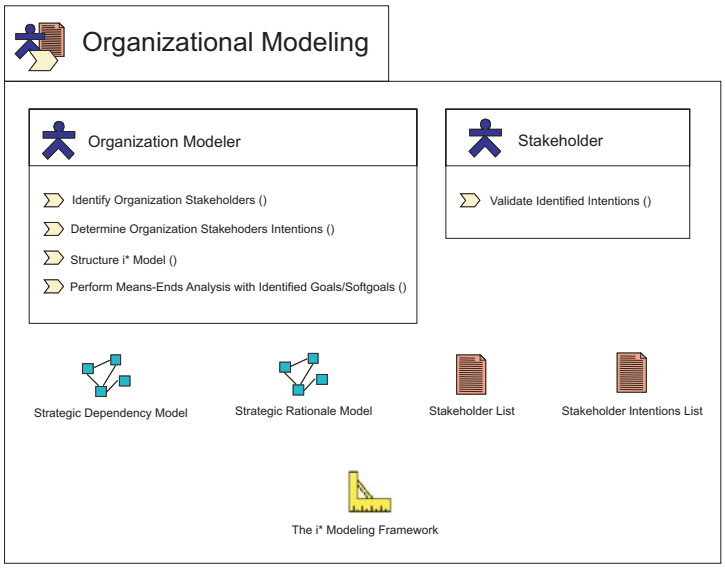


Figure 5-5. Organizational Modeling ProcessPackage

During this discipline, organization modelers model the target domain in terms of social actors and their intentions. Each goal intention is analyzed from the actor's point of view, resulting in social dependencies between them.

As represented in Figure 5-6, the major steps identified to deal with Organizational Modeling issues are:

- **Organization Stakeholders Analysis**: This WorkDefinition is mostly the responsibility of the Organization Modeler. It regroups all the activities performed to identify stakeholders and to analyze and validate their intentions;

- **Strategic Analysis**: On the basis of the stakeholders identified and their intentions, the Requirements Engineer creates the Strategic Dependency and Strategic Rationale Models.

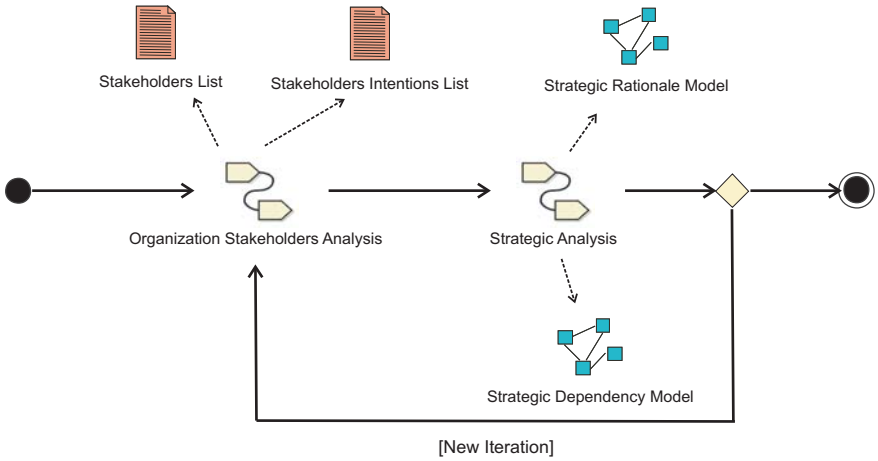


Figure 5-6. Organizational Modeling Viewed as a SPEM WorkDefinitions Flow

On a practical basis, the organization modeler **identifies the organization stakeholders** so that he can dispose of a Stakeholders List. He then **determines the organization stakeholders intentions** and **validates the identified intentions** with them. He finally **structures the i* models** to create the Strategic Dependency Model and **performs means-end analysis with identified goals and softgoals** to create the Strategic Rationale Model. This process is presented in Figure 5-7.

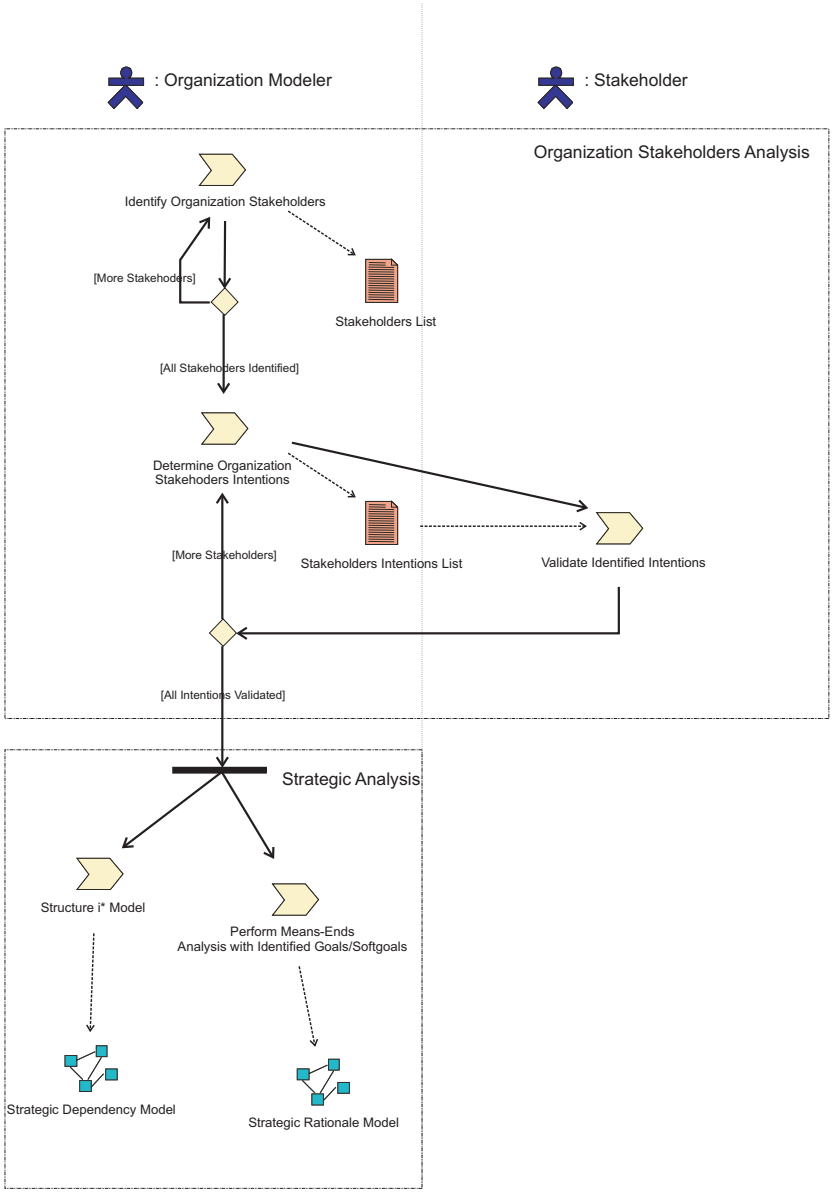


Figure 5-7. Organizational Modeling Workflow[5]

[5] Note that at activates level, workflows are given for information only since, at that level activates are atomic and, within a WorkDefinition, can be fulfilled in an undefined sequence or in parallel.

5.3.2 Requirements Engineering

The Requirements Engineering discipline, inspired from the Tropos *Late Requirements* stage, extends models created previously by including the system to-be, modelized as one or more actors. The system dependencies contribute in meeting goals of the stakeholders identified in Organizational Modeling discipline.

As shown in Figure 5-8, two roles are involved, *Requirements Engineer* and *Stakeholder*, and four WorkProducts are produced by this discipline: *Strategic Dependency Model*, *Strategic Rationale Model*, *Stakeholder List* and *Stakeholder Intention List*.

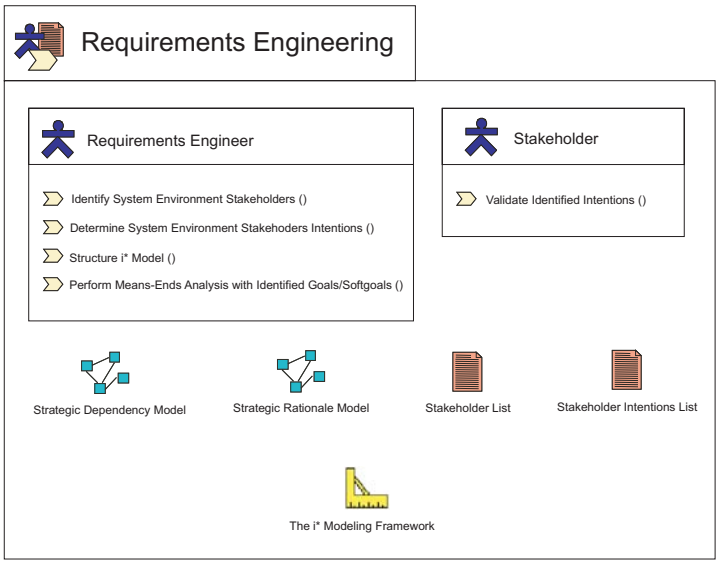


Figure 5-8. Requirements Engineering ProcessPackage

During this discipline, the new dependencies created by adding the system to the model are defining the system functional (goals) and non-functional (softgoals) requirements.

As represented in Figure 5-9, the major steps we identify to deal with Requirements Engineering issues are:

- **System Environment Stakeholders Analysis**: This WorkDefinition is mostly the responsibility of the Requirements Engineer. It regroups all the activities performed to identify stakeholders and to analyze and validate their intentions;

- **Strategic Analysis**: The Strategic Dependency and Strategic Rationale Models are created on the basis of the identified stakeholders and their intentions.

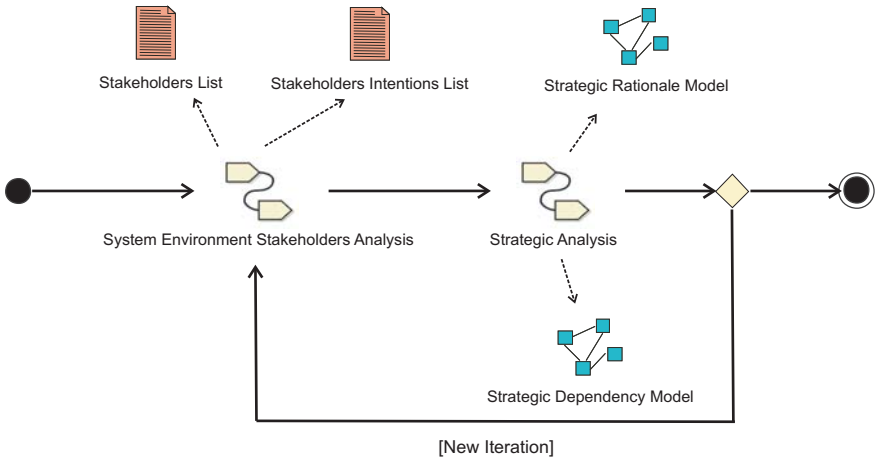


Figure 5-9. Requirements Engineering Viewed as a SPEM WorkDefinitions Flow

On a practical basis, the requirements engineer **identifies the system environment stakeholders** so that he can dispose of a Stakeholders List. He then **determines the system environment stakeholders intentions** and **validates the identified intentions** with them. He finally **structures the i* models** to create the Strategic Dependency Model and **performs means-end analysis with identified goals and softgoals** to create the Strategic Rationale Model. This process is presented in Figure 5-10.

Figure 5-10. Requirements Engineering Workflow

5.3.3 Architectural Design

The Architectural Design discipline, inspired by the Tropos *Architectural Design* stage, aims to build the system's architecture specification, by organizing the dependencies between the various sub-actors identified so far, in order to meet functional and non-functional requirements of the system.

As shown in Figure 5-11, three roles are involved, *Software Architect*, *System Specifier* and *Analyst*, and there are five WorkProducts produced by this discipline: *Strategic Dependency Model*, *Strategic Rationale Model*, *Actors List*, *Actors Intention List* and *System Architecture Specification*.

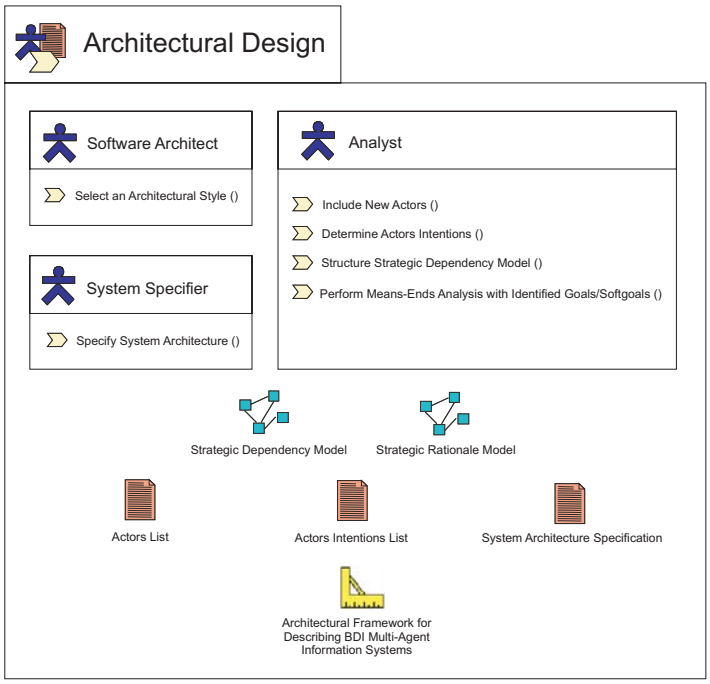


Figure 5-11. Architectural Design ProcessPackage

As represented in Figure 5-12, the major steps we identify to deal with Architectural Design issues are:

- **System Architecture Analysis**: This WorkDefinition is the responsibility of the Software Architect and is concerned with the process of selecting an architecture from the architectural styles catalogue;

- **Actors Analysis**: This WorkDefinition is concerned with the identification of new actors and their intentions inherited from the use of a specific architectural pattern;

- **Strategic Analysis**: This WorkDefinition is concerned with the inclusion of new actors and their intentions revealed by the use of a specific pattern into Strategic Dependency and Strategic Rationale Diagrams;

- **System Architecture Design**: This WorkDefinition is the responsibility of the System Specifier and is concerned with the process of specifying formally the architecture using an Architectural Description Language.

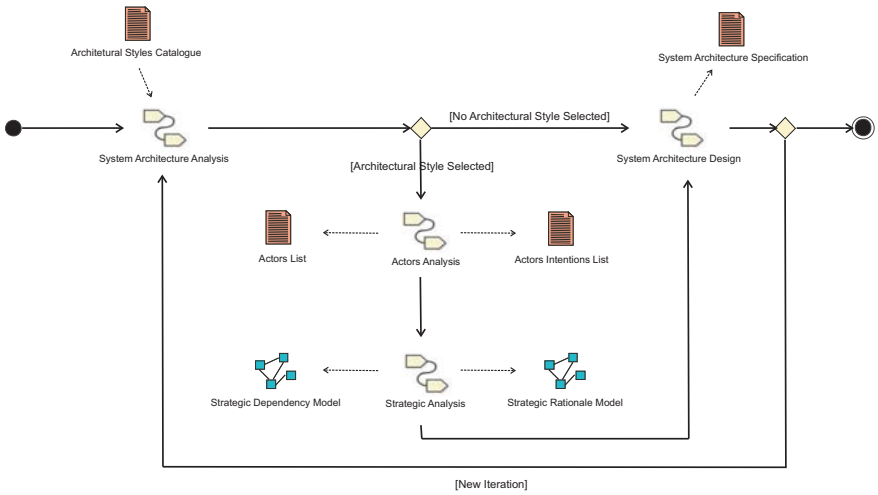


Figure 5-12. Architectural Design Viewed as a SPEM WorkDefinitions Flow

Figure 5-13 describes the Architectural Design discipline workflow. The Software Architect uses a non-functional requirements analysis to **select the most appropriate Architectural Style** for the module to-be from the Architectural Styles Catalogue (see [KGM06]). If such a style has been selected, **new actors and their identified intentions are added** to the Strategic Dependency and Strategic Rationale Models according to the semantics of the selected style. Finally, the System Architecture is **formally specified** with the ADL presented in [FKWA07].

For a long time, the exploitation of software architectures and architectural styles was informal and ad hoc. Architectural configurations were typically described using informal box-line diagrams in design documentation, providing little information about the computations represented by boxes, the interface of the components, or the nature of their interactions. This lack of information severely limited the usefulness of these diagrams for reasoning and analyzing system architecture.

In recent years, there have been several proposals for providing a sounder basis for describing and reasoning about software architecture. In particular, modeling nota-

tions known as architecture description languages have emerged. Architectural description languages are formal languages that are used to specify the architecture of a system [SG96].

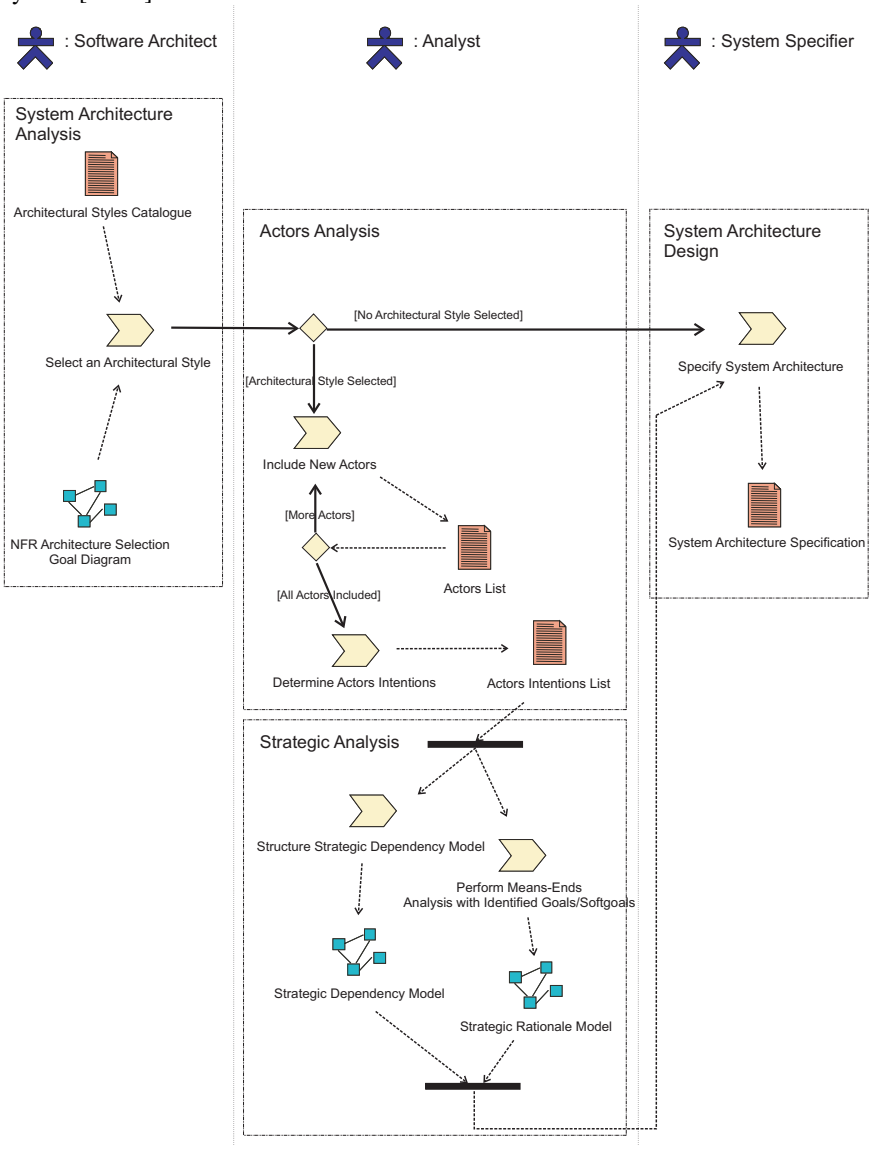


Figure 5-13. Architectural Design Workflow

5.3.4 Detailed design

The Detailed Design discipline, inspired by Tropos *Detailed Design*, aims at defining the behavior of each architectural component in further detail. This discipline is concerned with the specification of the agent micro level taking into account the implementation platforms. The objective is to perform a design that will map directly to the code.

Diagrams used here are AUML diagrams, including Communication Diagram (UML Sequence-like Diagram), Dynamic Diagram (UML Activity-like Diagram) and Structural Diagrams (UML Class-like Diagram). For further details, see [DKP03].

As shown in Figure 5-14, three roles are involved, *Software Architect*, *Functional Analyst* and *Agent Designer*, six WorkProducts are produced by this discipline: *Strategic Dependency Model*, *UML-like Class Diagram with Agent Concepts*, *UML-like Activity Diagrams*, *Extended AUML Sequence Diagram*, *Social Patterns Catalogue* and *Services List*.

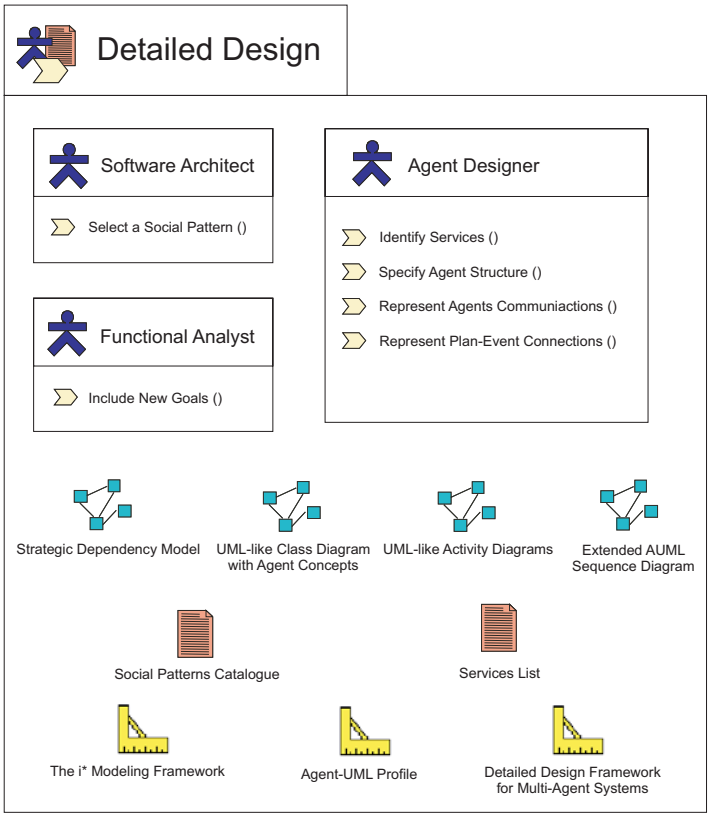


Figure 5-14. Detailed Design ProcessPackage

As represented in Figure 5-15, the major steps identified to deal with Detailed Design issues are:

- **Social Analysis**: This WorkDefinition is the responsibility of the Software Architect and is concerned with the process of selecting a social pattern from the social patterns catalogue;

- **Strategic Analysis**: This WorkDefinition is concerned with the inclusion of new goals inherited from the use of a specific social pattern;

- **Agent Behavior Description**: This WorkDefinition is the responsibility of the Agent Designer and is concerned with the full description of the agent behavior through multiple views.

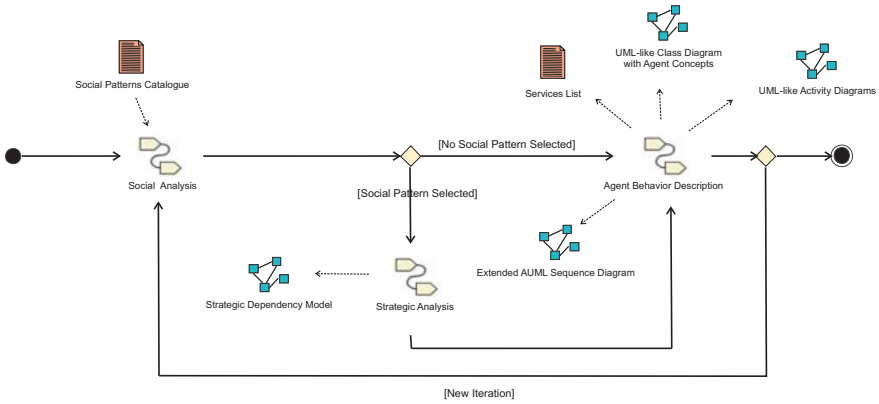


Figure 5-15. Detailed Design Viewed as a SPEM WorkDefinitions Flow

Figure 5-16 describes the workflow of the Detailed Design discipline. The Software Architect **selects the most appropriate Social Patterns** for the components under development from the catalogue overviewed in [DKP03]. **New goals are included** to the Strategic Dependency Model (Social Dimension) according to the semantics of the pattern. The Agent Designer **identifies services** provided by each agent to achieve goal dependencies. Each service belongs to an agent and is represented with an NFR goal analysis to refine the Strategic Rationale Diagram (Intentional Dimension). The **structure of each agent and its components** such as Plans, Events and Beliefs are then specified with an agent UML class diagram (Structural Dimension). **Agents communicate through events exchanged in the system and modeled** in a temporal manner with extended Agent UML sequence diagrams (Communicational Dimension). **The synchronization and the relationships between plans and events are designed** through agent oriented activity diagrams (Dynamic Dimension).

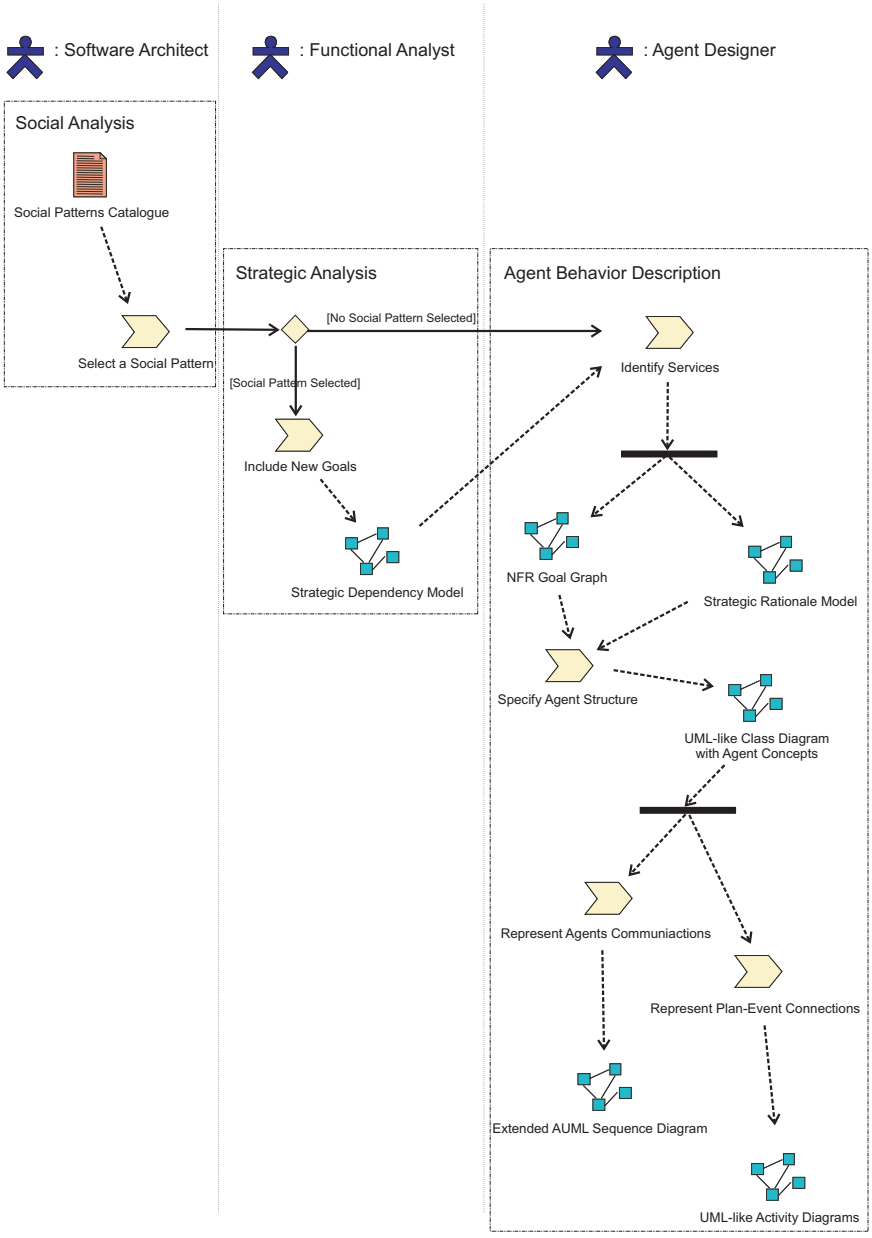


Figure 5-16. Detailed Design Workflow

5.3.5 Implementation

The implementation discipline aims to produce an executable release of the application on the basis of the detailed design specification. To achieve it, in the context of our design process framework, an Agent-Oriented programming platform is required.

As shown in Figure 5-17, three roles are involved, *Developer*, *User Interface Designer* and *Software Architect*, and four WorkProducts are produced by this discipline: *Agent Model*, *Implementation Model*, *Executable Release* and *User Interface Prototypes*.

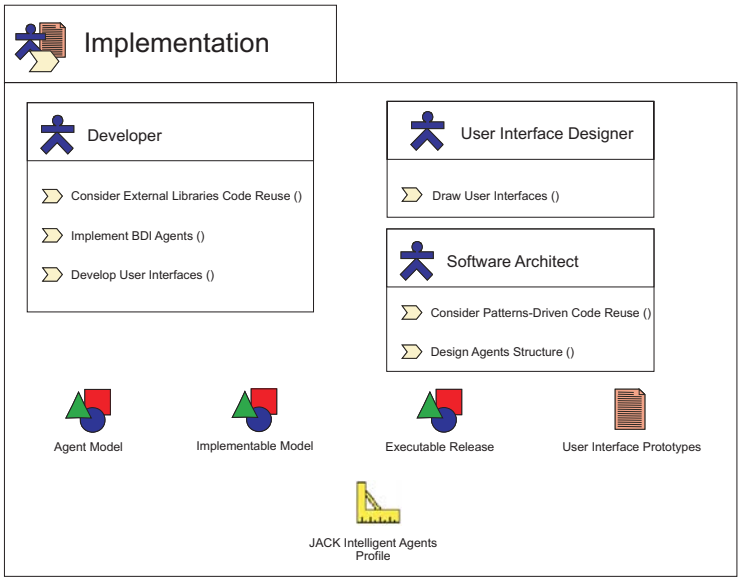


Figure 5-17. Implementation ProcessPackage

As represented in Figure 5-18, the major steps we identify to deal with Implementation issues are:

- **Code Reuse Analysis**: This WorkDefinition is the responsibility of the Software Architect and is concerned with the study of code reuse issued from the used patterns or other libraries;

- **Model Structuration**: This WorkDefinition is concerned with the automatic code generation and other user interfaces;

- **Model Implementation**: This WorkDefinition is the responsibility of the Developer and is concerned with the practical implementation of the application routines.

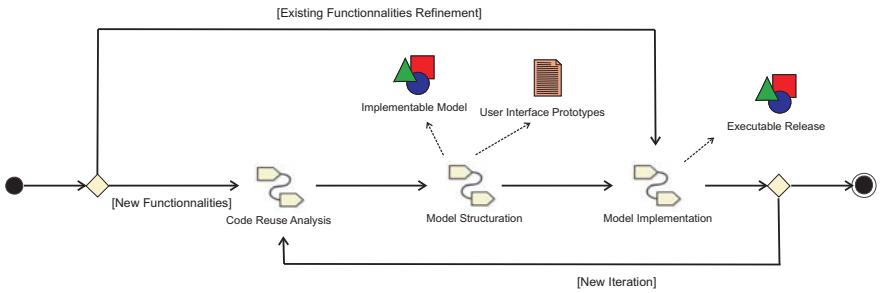


Figure 5-18. Implementation Viewed as a SPEM WorkDefinitions Flow

Figure 5-19 describes the Implementation discipline workflow. Firstly, code recycling is envisaged. The **integration of previously developed components/modules** is tried first, driven by styles and patterns selected earlier. Secondly the **incorporation of code through the use of external libraries** such as Open Source Software or *Commercial Off-the-Shelf* (COTS) components is considered. The next step in the process is the **creation of an agent skeleton** based on the specification developed during Detailed Design. The **user interfaces** of the application are also sketched, previously developed interfaces are refined on the basis of the user's feedback. The **code associated to the Beliefs, Desires and Intentions of each agent** is written and the user interfaces are developed.

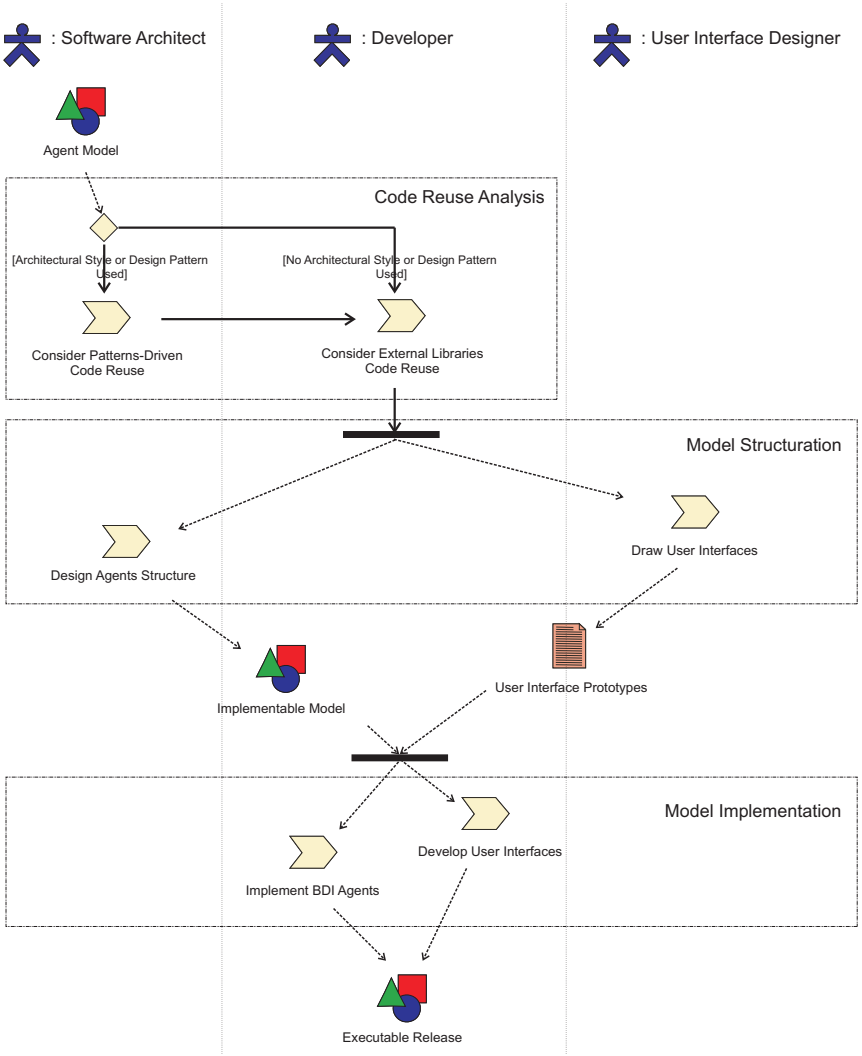


Figure 5-19. Implementation Workflow

5.3.6 Test

This discipline aims on evaluating the quality of the executable release. The Test Policy is defined with the End User, which will provide feedback on the product.

As shown in Figure 5-20, three roles are involved, *End User*, *Test Manager* and *Tester*. Four WorkProducts are produced by this discipline: *Executable Release*, *Test Policy*, *Validation Evaluation* and *Test Report*.

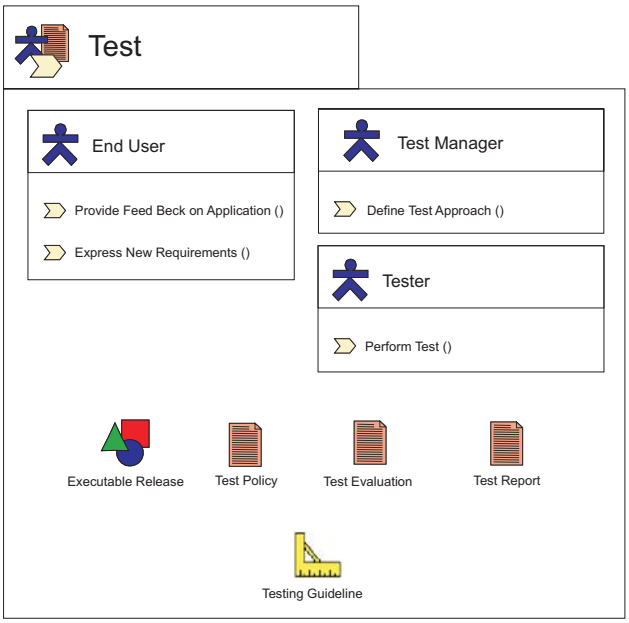


Figure 5-20. Test ProcessPackage

As represented in Figure 5-21, the major steps identified to deal with Test issues are :

- **Test Management**: This WorkDefinition is concerned with the activities destined to manage the process of testing;
- **Testing**: This WorkDefinition is concerned with practical activities of testing the prototypes and application to the end-users.

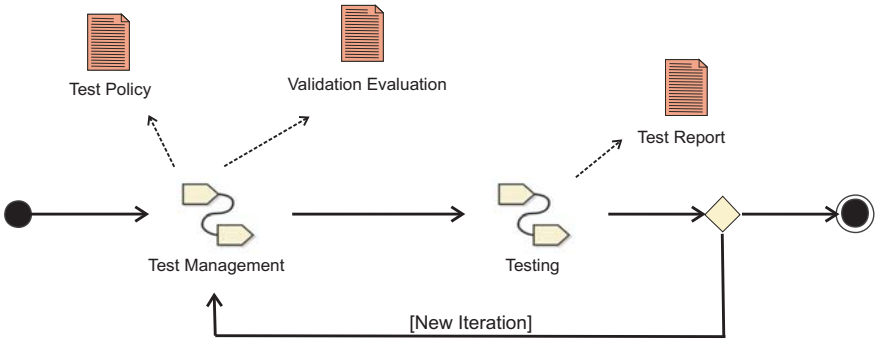


Figure 5-21. Test Viewed as a SPEM WorkDefinitions Flow

Figure 5-22 describes the Test discipline workflow. Firstly, the test manager **defines the test approach**. **The tests are then performed** to the end users who **provide feed back onto the application** and **express new requirements**. The test manager finally **reviews and evaluates the tests** to provide feed back for the next application.

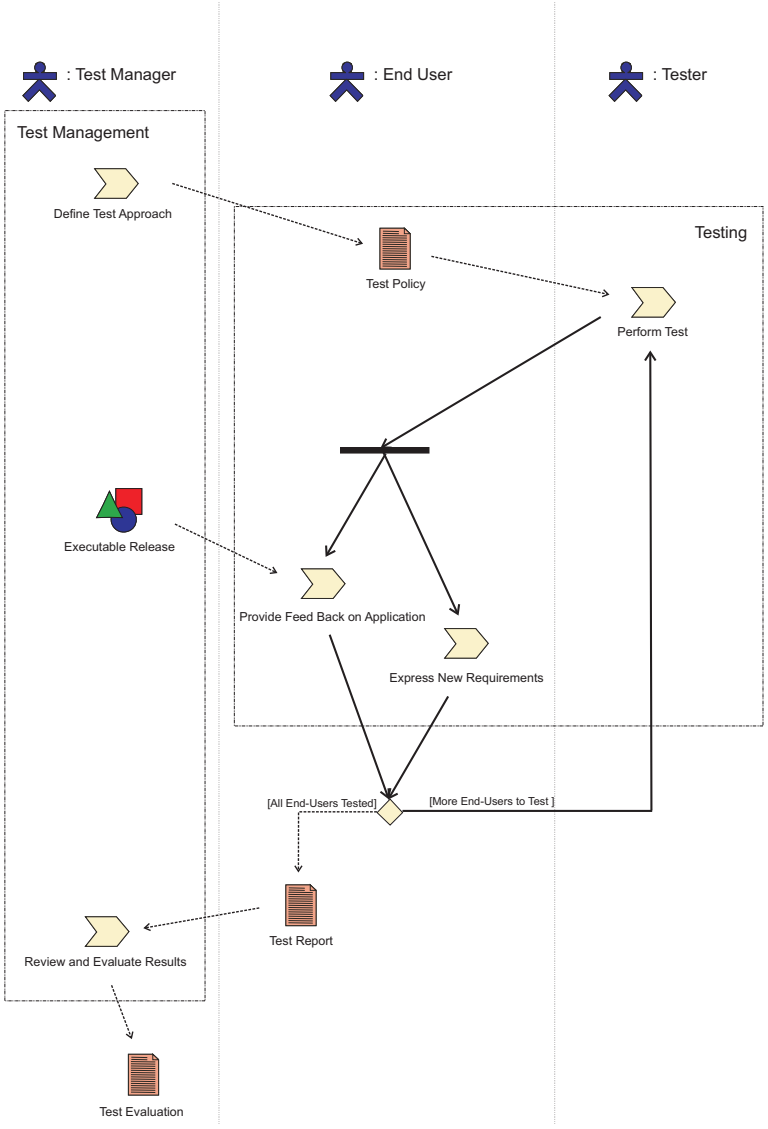


Figure 5-22. Test Workflow

5.3.7 Deployment

The aim of the deployment discipline is to test the software in its final operational environment. This discipline includes installing the release, writing User Manual, testing the deployment, reviewing and evaluating results.

As shown in Figure 5-23, five roles are involved, Deployment Manager, Installer, Developer, User Manual Developer and Tester, six WorkProducts produced by this discipline: Operational Release, Installable Release, Deployment policy, User Manuals, Deployment Evaluation and Test Report.

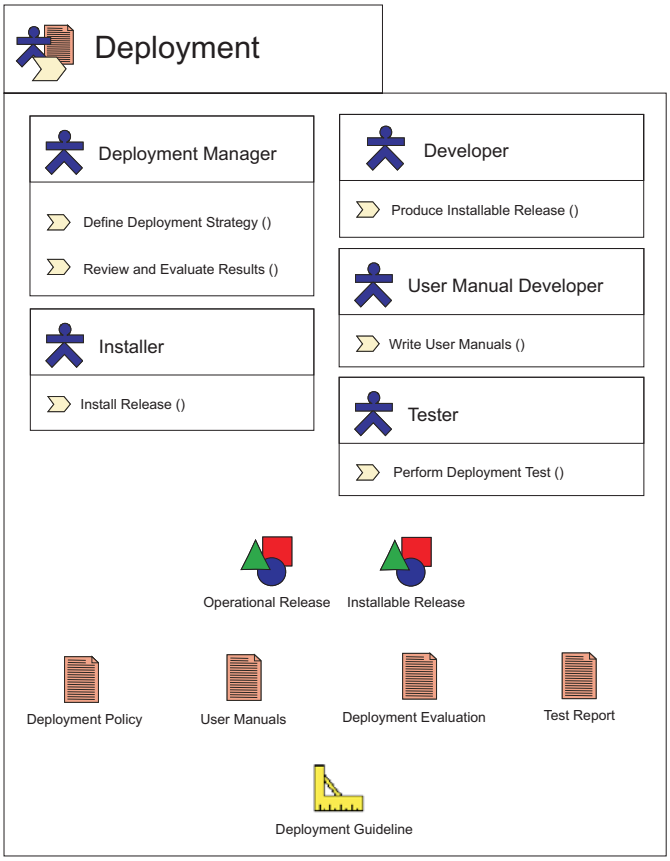


Figure 5-23. Deployment ProcessPackage

As represented in Figure 5-24, the major steps we identify to deal with Deployment issues are:

- **Deployment Management**: This WorkDefinition is concerned with the activities destined to manage the process of deployment;
- **Testing**: This WorkDefinition is concerned with practical activities of deployment the application into the organization.

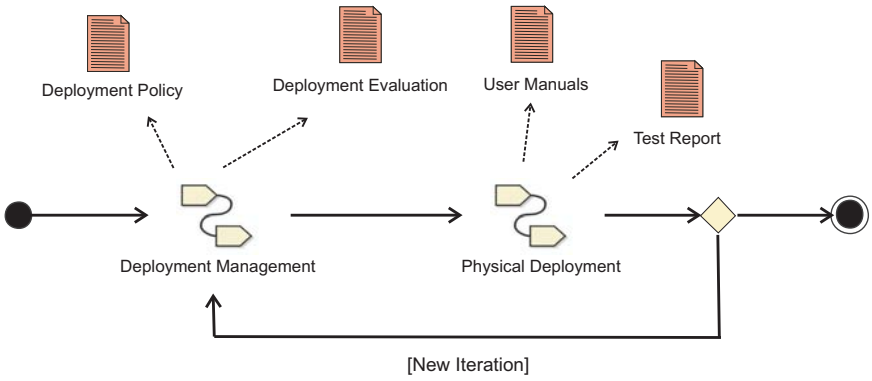


Figure 5-24. Deployment Viewed as a SPEM WorkDefinitions Flow

Figure 5-25 describes the Test discipline workflow. Firstly, the deployment manager **defines the deployment strategy**. **Users Manuals are written** for the end users and **an installable release is produced** and then **installed**. **Deployment tests are then performed**. The deployment manager finally **reviews and evaluates the results**.

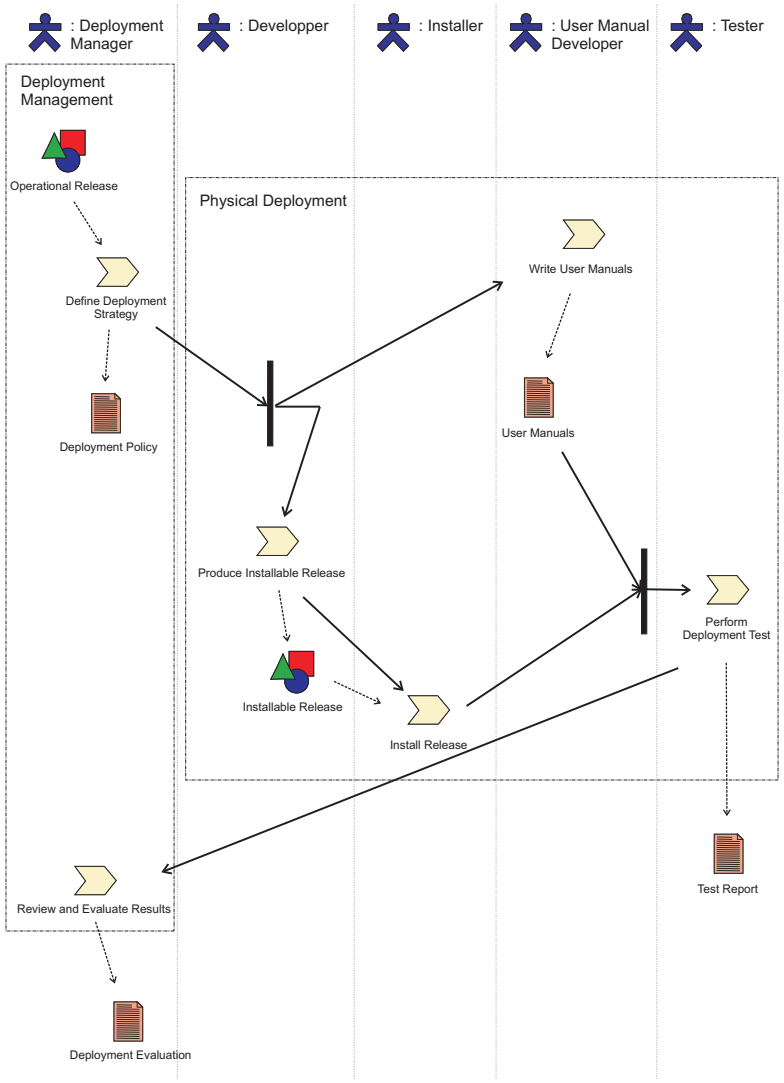


Figure 5-25. Deployment Workflow

5.4 The Process Support Discipline

In this section, the I-Tropos support disciplines i.e. Risk Management, Quality Management, Time Management and Software Process Management are overviewed. These disciplines purpose is to provide features to support software development i.e. tools to manage risks, quality levels, time, resources allocation but also the software

process itself. All those features can be regrouped onto the term *software project management*.

5.4.1 Risk Management

Risk Management is the process of identifying, analyzing, assessing risk as well as developing strategies to manage it. Strategies include transferring risk to another party, avoiding risk, reducing its negative effects or accepting some or all of the consequences of a particular one. Technical answers are available to manage risky issues. Choosing the right mean to deal with particular risk is a matter of compromise between level of security and cost. This compromise requires an accurate identification of the threats as well as their adequate evaluation.

Figure 5-26. Risk Management ProcessPackage

The analysis of risks in terms of the links existing between the business assets of an organization and the technical aspects associated with its information system seems to be best suited for the application on SE. To achieve such a result, a specific framework has been included into our methodology materialized through the Risk Management discipline.

The major steps we identify to deal with risk issues are:

- **Goal-based Threats Identification**: It consists of identifying the potential threats and evaluating their impact onto the (functional) features that will be developed within the project. Practically, this WorkProduct implies finding

out the threats that are going to act upon the project and quantifying them. To achieve such a result, the *probability of unsatisfactory outcome* and the *impact of unsatisfactory outcome* are multiplied for each identified risks to compute the *Risk Exposure*;

- **Operational Measures**: this includes the measures taken to deal with the identified threats. Iterative development planning will be driven by impact on functional developments of the identified threats;

- **Risk Monitoring**: this WorkProduct includes the evaluation of the measures taken to deal with the identified threats.

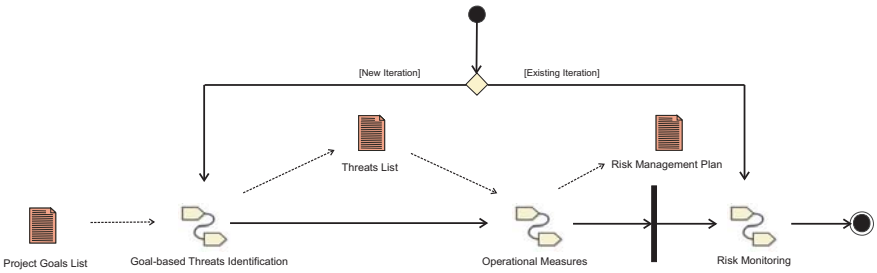


Figure 5-27. Risk Management Viewed as a SPEM WorkDefinitions Flow

Figure 5-28 describes the Risk Management discipline workflow. The risk manager identifies and defines the potential threats the project in general and more particularly its goals are facing. Risk analysis should be validated by stakeholders as they know the actual system as well as the specific aspects of their business. Each threat will be attributed a weight (as a numerical value), the bigger, the potentially most damageable feature in terms of finance, image, etc.

The exposure of each particular goal to each particular threat is then evaluated on the basis of a low/medium/high/none scale. The risk hierarchy is established through a "Risk Table" assigning to each goal a threat exposure probability. We basically define four values for threat exposure probability:

- Low: "L" in a yellow filled cell, has a weight of 1;
- Medium: "M" in a orange filled cell, has a weight of 2;
- High: "H" in a red filled cell, has a weight of 4;
- Non-existing: an empty cell, this goal is not concerned by this threat, has a weight of 0.

Using the risk table, every threat weight is multiplied by every occurrence of threat probability of occurrence ("L", "M" or "H"). An example of a Risk Table is shown on Table 5-1.

	Threat weight		Goal 1	Goal 2	Goal 3
Threat 1	1			H	L
Threat 2	2		L		M
Goal risk exposure			**2**	**4**	**5**

Table 5-1. Risk Evaluation Table

For instance, "Threat 1" has a weight of 1 and "Threat 2" has a weight of 2:

- Goal 1 has a total weight of: (1*0) + (2*1) = 2;
- Goal 2 has a total weight of: (1*4) + (2*0) = 4;
- Goal 3 has a total weight of: (1*1) + (2*2) = 5.

Risk hierarchy is computed on the basis of each goal total weight: the heavier, the most risky issue.

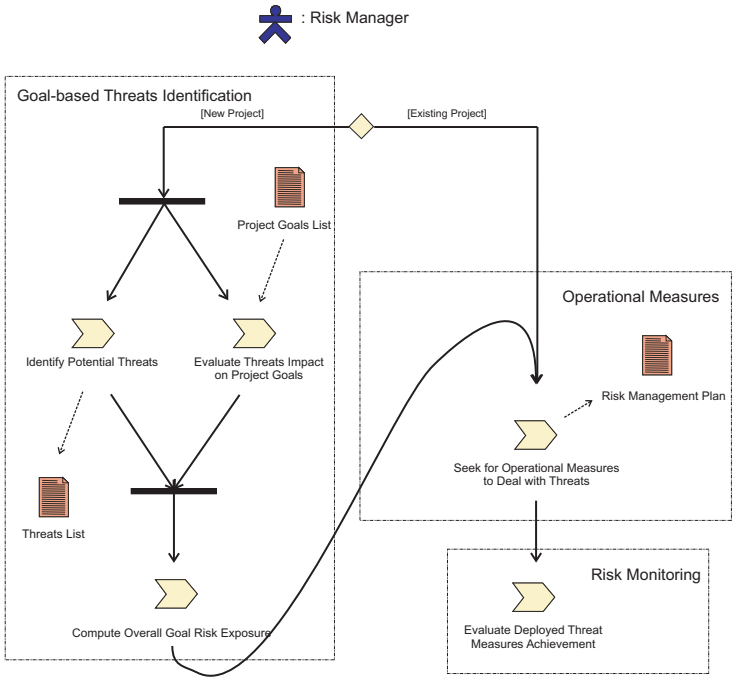


Figure 5-28. Risk Management Workflow

5.4.2 Quality Management

Quality Management is the process of ensuring that quality expected and contracted with clients is achieved throughout the project. Strategies include defining quality issues and the minimum quality level for those issues. Technical answers are available to reach quality benchmarks. Choosing the right mean to deal with quality issues is a matter of compromise between level of quality and cost. This compromise requires an accurate identification of the quality benchmarks as well as their adequate evaluation.

Figure 5-29. Quality Management ProcessPackage

The analysis of quality in terms of the links between the quality benchmarks and the features to be developed seems to be best suited for the application on SE. To achieve such a result, a specific framework has been included into our methodology: the Quality Management discipline.

The major steps we identify to deal with quality issues are:

- **Goal-based Quality Identification**: It consists in identifying the quality issues and evaluating their minimal level onto the (functional) features that will be developed within project. Practically, this WorkProduct implies finding out the quality features of interest onto the project, weighting their relative importance and quantifying an acceptable level;

- **Operational Measures**: it includes the measures taken to deal with the identified quality features. Iterative development planning will be driven by impact on functional developments of the identified quality measures;

- **Quality Monitoring**: this WorkProduct includes the evaluation of the measures taken to deal with the identified quality features.

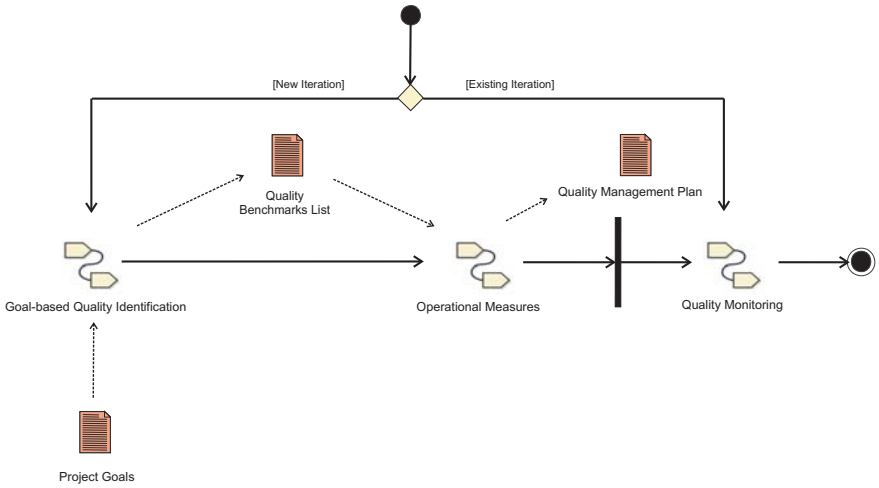


Figure 5-30. Quality Management Viewed as a SPEM WorkDefinitions Flow

Figure 5-31 describes the Risk Management discipline workflow. The quality manager identifies and defines the potential quality factors of the project goals. Quality analysis should be validated by stakeholders as they know the actual system as well as the specific aspects of their business. Each quality factor will be attributed a weight (as a numerical value), the biggest, the most quality concerned feature.

The exposure of each particular goal to each particular quality factor is then evaluated on the basis of a low/medium/high/none scale. The quality hierarchy is established through a "Quality Table" assigning to each goal a quality factor concern probability. We basically define four values for quality factor concern probability:

- Low: "L" in a yellow filled cell, has a weight of 1;
- Medium: "M" in a orange filled cell, has a weight of 2;
- High: "H" in a red filled cell, has a weight of 4;
- Non-existing: an empty cell, this goal is not concerned by this quality factor, has a weight of 0.

Using the quality table, every quality weight is multiplied by every occurrence of quality probability ("L", "M" or "H"). An example of a Quality Table is shown on Table 5-2.

	Quality factor weight		Goal 1	Goal 2	Goal 3
Quality factor 1	1			H	L
Quality factor 2	2		L		M
Goal quality weight			**2**	**4**	**5**

Table 5-2. Quality Evaluation Table

For instance, "Quality factor 1" has a weight of 1 and "Quality factor 2" has a weight of 2:

- Goal 1 has a total weight of: (1*0) + (2*1) = 2;
- Goal 2 has a total weight of: (1*4) + (2*0) = 4;
- Goal 3 has a total weight of: (1*1) + (2*2) = 5.

Quality hierarchy is computed on the basis of each goal total weight, the heavier, the more quality concerned issue.

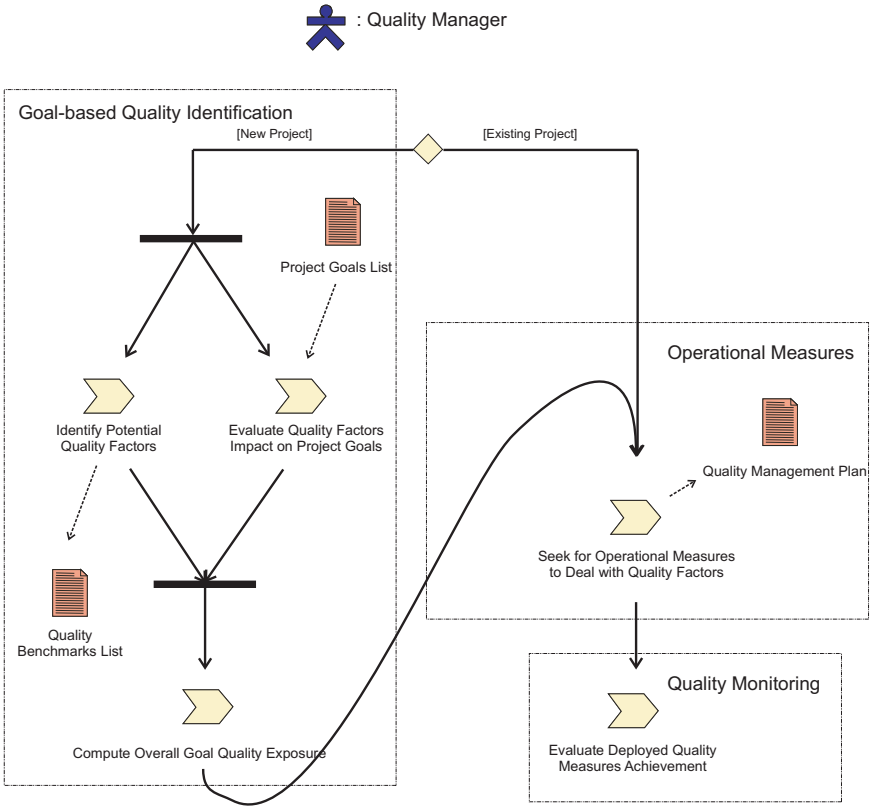


Figure 5-31. Quality Management Workflow

5.4.3 Time Management

Time Management is the process of monitoring and controlling the resources (time, human and material) spent on the activities and tasks of a project. This discipline is of primary importance since, on the basis of the risk and quality analyses, the global iterations time and human resources allocation are computed; they are revised during each iteration.

Figure 5-32. Time Management ProcessPackage

Planning in terms of number of iterations, iteration duration, iteration focus, number of human resources playing the specific I-Tropos roles, etc. is, in our approach, based on i* goals prioritized using risk and quality issues. To achieve such a result, a specific framework has been included into our methodology: the Time Management discipline.

The major steps identified to deal with time issues are:

- **Goal Prioritization**: On the basis of the risk and quality management plans, the (functional) features that will be developed within the project are prioritized. Practically, this WorkDefinition implies classifying the most risky and quality dependent goal to the less one;

- **Iteration Planning**: On the basis of the classification made into the goal prioritization the features developed during each iteration are planned. This also requires human resources allocation to fulfill the activities required for iteration objective achievement;

- **Time Monitoring**: this WorkDefinition includes the evaluation of the iteration planning.

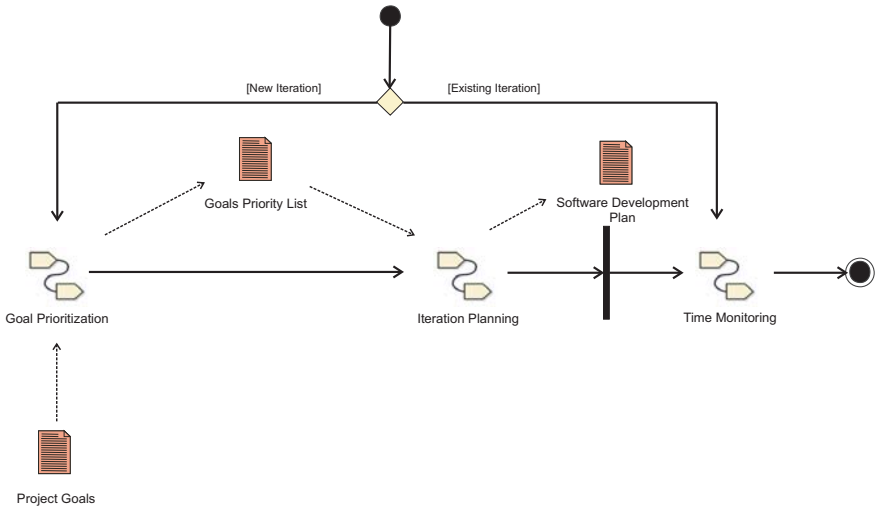


Figure 5-33. Time Management Viewed as a SPEM WorkDefinitions Flow

Figure 5-34 describes the Time Management discipline workflow. The time manager computes the relative risk exposure as well as the relative quality factors concerned by all project goals. Computation is realized by dividing each goal risk exposure and quality factors concerned by the overall goals risk exposure and overall quality factors concern. The objective is to get a relative measure of risk exposure and quality factors concern that can be used for comparison. At this level and on the basis of those relative measures a goal priority level can be computed. This computation still requires calibration. Indeed, the most risky and quality concerned issue should receive highest priority, the balance between those two factors should be determined for each particular project on the basis of the project characteristics. Accurate calibration remains an open issue, it should be evaluated on the basis of a large number of software projects. The goal with highest priority level will be envisaged at first in the elaboration phase, goals' priority rank is determined that way. This rank is then reviewed on the basis of the precedence constraints. Indeed, in some cases a goal requires to be envisaged before another due to practical aspects. This dimension can easily be incorporated in goal priority ranking. When the final ranking has been determined, goals realization can be assigned to iterations for software project iterative planning. Such work also requires an adequate evaluation of the goal realization complexity; next paragraph envisages effort estimation on the basis of goals.

Determining the effort required to realize a goal is another fundamental issue of time management. As pointed out in the state of the art, at early stages software sizing was essentially based on a number of lines of source code (SLOC), the later estimation techniques found in SE literature mostly focus on attributing points to scope elements (for example Function Points [LJ90], Use Case Points [SW01], Internet Points [CXG01], etc.). Such a technique can be mapped to higher level i* SDD with respect to the rule defined in Section 5.1.3, this technique described hereafter will be

referenced as *Goal Points*. Assigning points to a goal requires estimating their complexity. This issue is very similar to the one pointed out into the Use Case Points technique. Basically, the Use Case Points technique defines three levels of complexity both for agents and use cases, the later are, in that technique, evaluated as:

- Simple when a simple UI touches only a single database entity, its success scenario has 3 steps or less and its implementation involves less than 5 classes;
- Medium when more interface design is required and touches 2 or more database entities, its success scenario has 4 to 7 steps it is considered as medium and its implementation involves 5 to 10 classes;
- Complex when a complex user interface is required or processing and touches 3 or more database entities, it success scenario has over 7 steps and its implementation involves more than 10 classes.

Following the definition of Section 5.1.3, a goal is a non-overlapping element that can be decomposed into a series of atomic tasks. Goal complexity estimation can thus be evaluated on the basis of the number of tasks to be performed to resolve its success scenario. On the basis of Use-Case Points literature we propose the to evaluate goals complexity on the following basis:

- if the goal success scenario involves 3 or less tasks it is considered as simple and will be assigned a weight of 5;
- if the goal success scenario involves 4 to 7 tasks it is considered as medium and will be assigned a weight of 10;
- if the goal success scenario involves 7 or more tasks it is considered as complex and will be assigned a weight of 15.

Agents can be of different nature, they can be physical devices, other systems or human beings. This distinction is made into the Use Case Points method and will be used as a starting point here:

- a simple agent represents another system with a defined *Application Programming Interface* (*API*), it will be assigned a weight of 1;
- a medium agent represents another system interacting through a protocol, like TCP/IP, it will be assigned a weight of 2;
- a complex agent is a person interacting via an interface, it will be assigned a weight of 3.

This technique as well as the accuracy of the use of use case points environmental and technical factors needs to be refined and balanced on the basis of a large number of software projects.

In this software project management framework, iteration planning is the process of attributing a (partial) goal realization into a particular iteration making part of a particular phase. The first step toward planning is to determine the number of iterations. Experience in RUP software projects (see [Kruchten03, WLK04a, WLK04b]) gives a number of iterations varying from 6 plus or minus 3 depending on the project size:

- the setting phase is made of 0 to 1 iteration;
- the blueprinting phase is made of 1 to 3 iteration(s);
- the building phase is made of 1 to 3 iteration(s);
- the setuping phase is made of 1 to 2 iteration(s).

Moreover classical schedule for effort repartition between the phases is:

- 10 per cent for the setting phase;
- 30 per cent for the blueprinting phase;
- 50 per cent for the building phase;
- 10 per cent for the setuping phase.

This means that the total amount for a goal computed with the Goal Points technique must be divided with respect to the previous repartition. Goals realization can be planned onto the different iterations of the different phases, with respect to the facts that:

- the most prior goal must be planned first;
- the iterations should also be balanced more or less uniformly in terms of effort.

Defining the process activities performed during the different iterations is the responsibility of the process manager. This discipline is depicted into the next section.

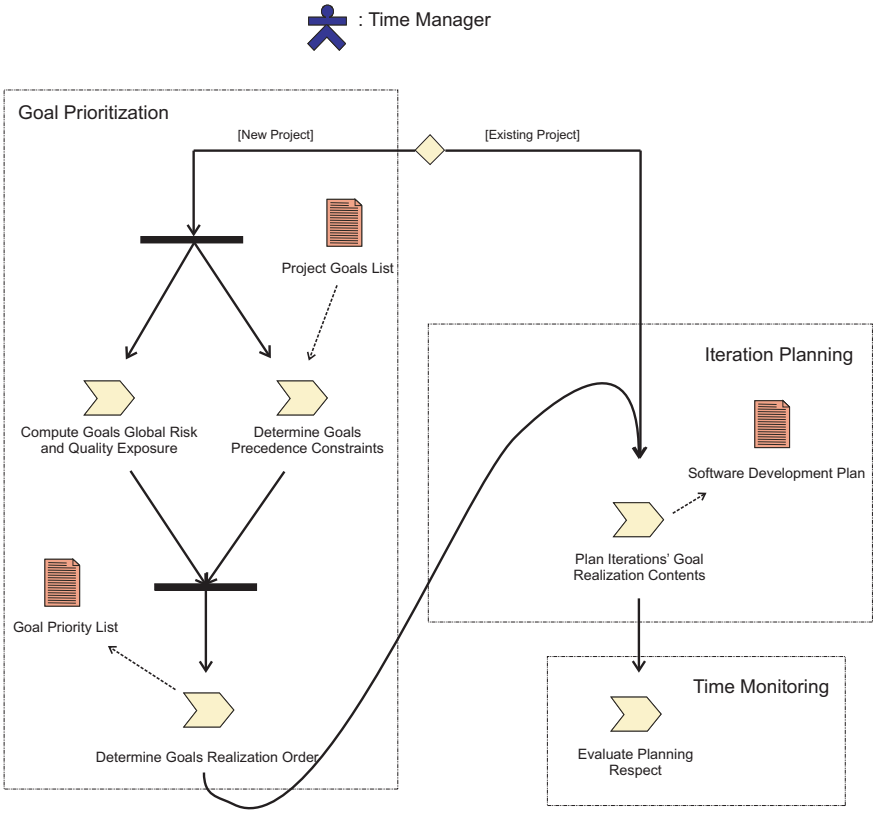


Figure 5-34. Time Management Workflow.

5.4.4 Software Process Management

Software Process Management is "*the use of process engineering concepts, techniques, and practices to explicitly monitor, control, and improve the systems engineering process. The objective of systems engineering process management is to enable an organization to produce system/segment products according to plan while simultaneously improving its ability to produce better products*." [CAWG96] In the context of this dissertation, Software Process Management regroups the activities aimed to tailor the generic process onto a specific project as well as improving the software process.

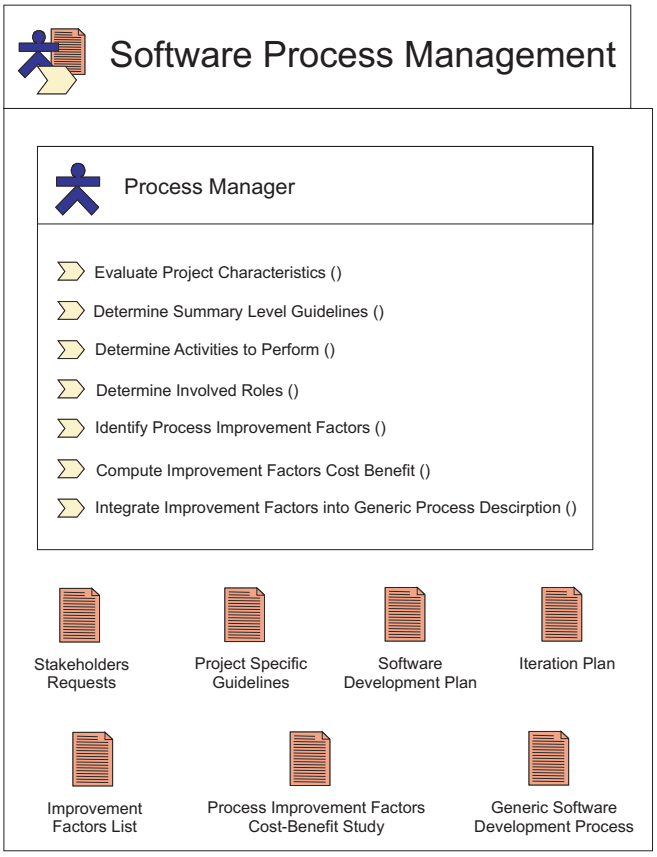


Figure 5-35. Software Process Management ProcessPackage

5.4.4.1 Process Tailoring

According to [Jalote01], "*Tailoring is the process of adjusting a previously defined process of an organization to obtain a process that is suitable for the particular business or technical needs of a project*." In other words, tailoring the process can be defined as adding, deleting, or modifying the activities of a generic process so that the resulting process is customized for a defined project objectives achievement.

Process tailoring is the responsibility of the Process Engineer. On the basis of the characteristics of the particular project (the target organization specificities, the characteristics of the team achieving the project, etc.) and of tailoring guidelines, he produces a project plan with general tailoring rules during the setting phase. At the beginning of the iteration the Project Manager performs a detailed tailoring where each activity defined in the generic process and performed during the present iteration is justified.

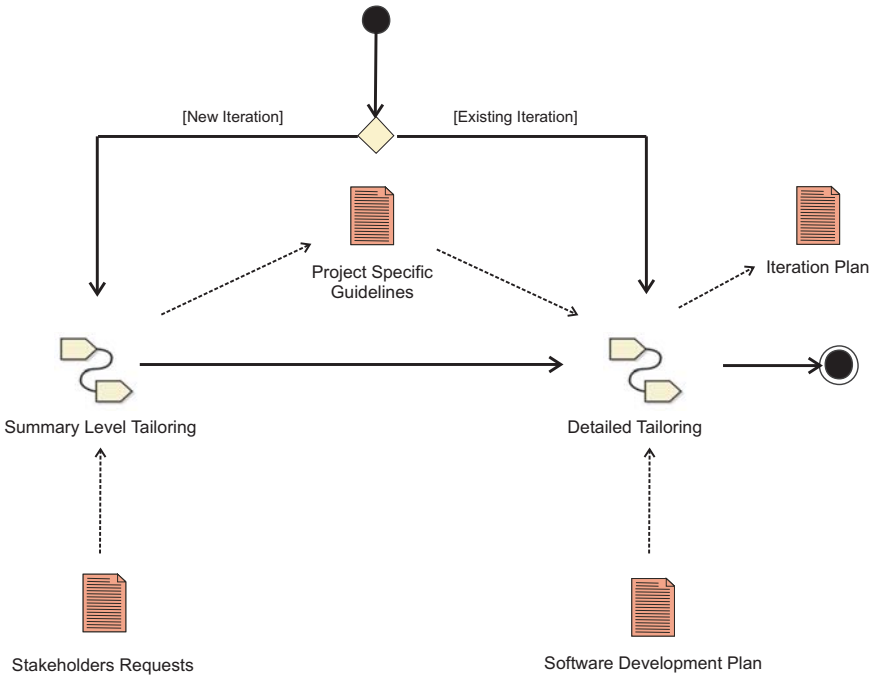


Figure 5-36. Software Process Tailoring Viewed as a SPEM WorkDefinitions Flow

At the early stages of the project, the process manager identifies the required WorkProducts to be performed onto the generic process description. This activity is mostly conditioned by the fact that:

- setting phase is mostly focused on requirements engineering and project planning;
- blueprinting phase is mostly focused on software design and prototyping;
- building phase is mostly focused on fully implementing the software product;
- setuping phase is mostly focused on deploying the software into the organization.

For each iteration, the process manager identifies and lists the activities that must be performed into the Iteration Plan. This process is reviewed at the beginning of each new iteration.

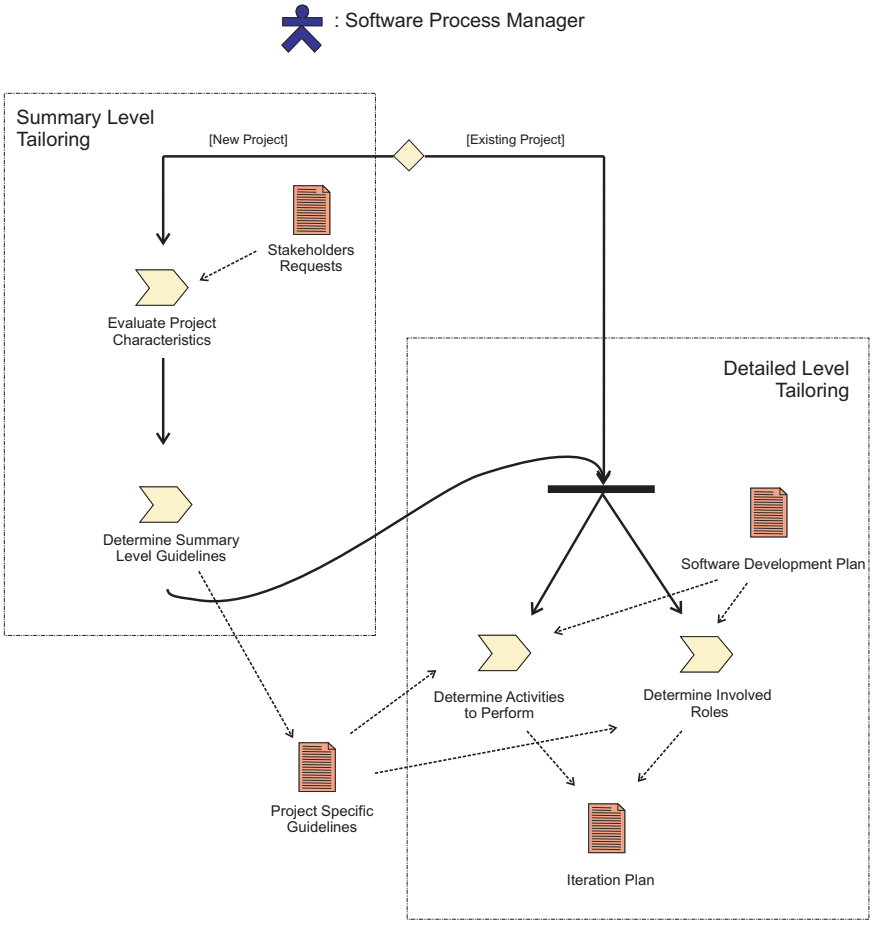


Figure 5-37. Software Process Tailoring Workflow

5.4.4.2 Software Process Improvement

The generic process description of I-Tropos has been presented into this chapter. However, processes in general and software processes in particular are continuously evolving to reach a higher maturity. Certifications such as *ISO*, *CMMI* or *Six Sigma* are precisely intended to evaluate organizations' level of process maturity. Such features are particularly important into software engineering; *CMMI-SW* is for example designed to measure software development processes maturity. Software process improvement allows to continuously reviewing the techniques to get more efficient procedures, to avoid resources waste, to better take risks into consideration or to produce higher quality levels products. I-Tropos should never be envisaged as a static process but as a continuously evolving process adapting to the organization and adopting projects specificities. To achieve such a continuous improvement process, a

specific framework has been included into the Software Process Management discipline.

Software Process Improvement includes:

- **Improvement Factors Identification**: this WorkDefinition includes an in depth review of each project by senior management to find out the optimization potentialities. A cost/benefit study can then be performed to estimate the improvement factor implementation profitability;

- **Improvement Factors Integration**: this WorkDefinition includes the practical activities to put the improvement factors into application.

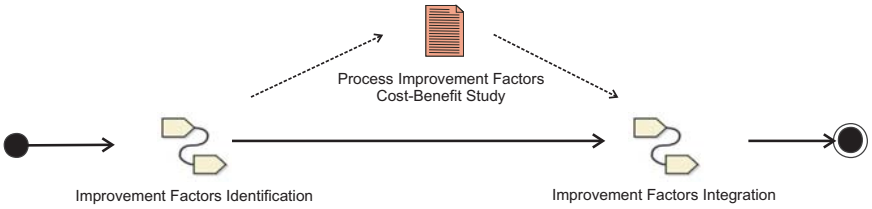


Figure 5-38. Software Process Improvement Viewed as a SPEM WorkDefinitions Flow

On a practical basis, the process manager continuously **identifies the process improvement factors** during the project. He then **computes these improvement factors cost benefit** to determine whether they should be included into the generic process description. Process improvement analysis should be validated by senior management, as their experience in software process is considerable. Profitable improvement factors are then **integrated into the generic software process description** which will be used as the guideline for forthcoming software projects.

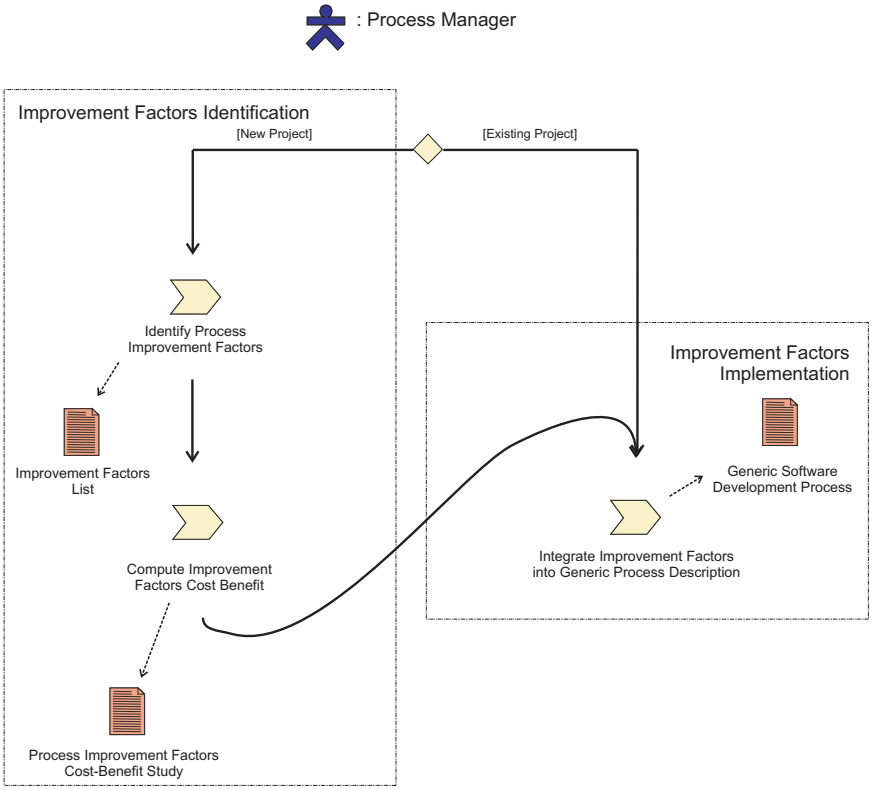


Figure 5-39. Software Process Improvement Workflow.

5.5 Future Work

The inclusion of the framework developed in [WAK08] constitutes an interesting evolution of the I-Tropos software process. It should nevertheless be extended to ensure a benefit compared to the actual process based on the extension of traditional i* goals. Indeed, it proposes a Strategic Services Diagram including higher level elements – the services – representing the non-overlapping processes to be developed into the software application. This diagram includes the representation of threats which are not represented as dependencies but rather as elements potentially related to the services. In the same way the diagram could be extended to include quality factors also potentially related to the services so that the entire software management framework depicted in this chapter could be driven by that single diagram. An algorithm can easily be executed onto the diagram to compute a services hierarchy on the basis of threats and quality factors. This work is currently under development.

5.5 Chapter Summary

In this chapter, we have presented the I-Tropos framework that offers a broader perspective of the traditional Tropos methodology for designing MAS. The framework is composed of seven complementary core disciplines: *Organizational Modeling*, *Requirements Engineering*, *Architectural Design*, *Detailed Design*, *Implementation*, *Test* and *Deployment* but also support disciplines to handle the project development called *Risk Management*, *Time Management*, *Quality Management* and *Software Process Management*. These disciplines repeated iteratively provide continuously evolving dimensions to completely fulfill the development of MAS applications involving users in an intensive manner. For each discipline, a generic description has been given, the fundamental concepts and notions were introduced. The workflow of the performed WorkDefinitions and Activities has been also given for both the core and the support disciplines.

Our contribution of this chapter is the framework itself. It integrates the agent concepts extracted and generalized from agent-oriented modeling language i* and uses goal elements as scope elements to drive the development process. The Software Process Engineering Meta-model concepts are reused with some extensions and adaptations for describing the I-Tropos MAS development process. The description covers the core disciplines as well as the support disciplines essential for controlled iterative development. The chapter finally points to future work designed to dispose of a diagram allowing project management capabilities directly integrated and deduced from high level modeling diagrams.

In the following, the framework will be applied to a case study (Chapter 6), a CASE-Tool for I-Tropos developments will also be presented (Chapter 7).

References

[AUML] FIPA, "Agent UML", http://www.auml.org/.

[ERPM06] H. Estrada, A.M. Rebollar, O. Pastor and J. Mylopoulos. "An Empirical Evaluation of the i* Framework in a Model-Based Software Generation Environment". *CAiSE 2006*, 513-527, 2006.

[CAWG96] Capability Assessment Working Group on Systems Engineering, "Systems Engineering Capability Assessment Model", INCOSE-TP-1996-002-01, Version 1.50a, June 1996.

[CKM02] J. Castro, M. Kolp and J. Mylopoulos. "Towards A Requirements-Driven Development Methodology : The Tropos Project". In *Information Systems*, Elsevier, 2002.

[CSS03] M. Cossentino, L. Sabatucci and V. Seidita. "Spem description of the passi process", Technical Report 20-03, ICAR-CNR, 2003.

[CXG01] Cost Xpert Group Inc. "Estimating Internet Development", 2001. Available at http://www.costxpert.com/Reviews_Articles/SoftDev/.

[DKP03] T. T. Do, M. Kolp and A. Pirotte. "Social Patterns for Designing Multi-Agent Systems". In *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, San Francisco, USA, July 2003.

[DG99] P. Dussauge, and B. Garrette. "Cooperative Strategy: Competing Successfully Through Strategic Alliances". *Wiley and Sons*, 1999.

[FKWA07] S. Faulkner, M. Kolp, Y. Wautelet and Y. Achbany. "A Formal Description Language for Multi-Agent Architectures". In *Proceedings of the 17th International Workshop on Agent-Oriented Information Systems*, Hakodate, Japan, 2006.

[LJ90] G. C. Low, D. R. Jeffery. "Function Points in the estimation and evaluation of the software process". IEEE Transactions on Software Engineering, 16(1), 64–71, 1990.

[IBM03] IBM. "The Rational Unified Process. Version 2003.06.00.65". *Rational Software Corporation*, 2003.

[Jade] Java Agent DEvelopment framework (JADE). http://jade.cselt.it/

[Jalote01] P. Jalote. "Software Project Management in Practice". *Pearson Education*, 2002.

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh. "The Unified Software Development Process". *Addision-Wesley*, 1999.

[JB00] I. Jacobson and S. Bylund. "The Road to the Unified Software Development Process". *Cambridge University Press*, 2000.

[KGM06] M. Kolp, P. Giorgini and J. Mylopoulos. “Multi-Agent Architectures as Organizational Structures”. In *International Journal of Autonomous Agents and Multi-Agent Systems* (*JAAMAS*), Kluwer, 2006.

[KFW08] M. Kolp, S. Faulkner and Y. Wautelet. “Social-Based Design Patterns for Agent-Oriented Software Engineering”. In *International Journal of Intelligent Information Technologies* (*IJIIT*), 4(3):23-44, Idea Group, 2008.

[Kruchten03] P. Kruchten. “The Rational Unified Process: An Introduction”. *Addison-Wesley*, 2003.

[Mintzberg92] H. Mintzberg. “Structure in fives: designing effective organizations”. *Prentice-Hall*, 1992.

[MSB99] J. Morabito, I. Sack and A. Bhate. “Organization modeling: innovative architectures for the 21st century”. *Prentice Hall*, 1999.

[OMG05] OMG, “The Software Process Engineering Metamodel Specification. Version 1.1”, January 2005.

[PEM08] O. Pastor, H. Estrada and A. Martiınez, “The strengths and weaknesses of the i* Framework: an experimental evaluation”. In *P. Giorgini, N. Maiden, J. Mylopoulos, E. Yu editors*, *Social Modeling for Requirements Engineering*, in *Cooperative Information Systems series*, MIT Press, 2008.

[Shaw95] M. Shaw. “Patterns for Software Architectures”. In *J. Coplien, D. Schmidt (eds)*, *Pattern Languages of Program Design*, Addison-Wesley, pp. 453-462, 1995.

[SG96] M. Shaw and D. Garlan. “Software Architecture: Perspectives on an Emerging Discipline”. *Prentice Hall*, 1996.

[Scott98] W.R. Scott. “Organizations: rational, natural, and open systems”, *Prentice Hall*, 1998.

[Segil96] L. Segil. “Intelligent business alliances: how to profit using today's most important strategic tool”. *Times Business*, 1996.

[SW01] G. Schneider and J. Winters. “Applying Use Cases - A Practical Guide” – Second Edition. *Addison Wesley*, Boston, 2001.

[SPGM05] A. Susi, A. Perini, P. Giorgini and J. Mylopoulos. “The Tropos Metamodel and its Use. Informatica”. 29(4):401–408, 2005.

[SAMC07] C. Silva, J. Araújo, A. Moreira, J. Castro. “Designing Social Patterns using Advanced Separation of Concerns”. In *19th Conference on Advanced Information Systems Engineering* (*CAiSE 2007*), Trondheim, Norway, June 2007.

[Silva07] C. Silva, “Separating Crosscutting Concerns in Agent Oriented Detailed Design: The Social Patterns Case”. *PhD Thesis*, *Universidade Federal de Pernambuco*, February 2007.

[UML] OMG. “OMG Unified Modeling Language Specification. Version 2.1.2”. November 2007.

[WAK08] Y. Wautelet, Y. Achbany and M. Kolp. "A Service-Oriented Framework for MAS Modeling". In *Proceedings of the the 10th International Conference on Enterprise Information Systems* (*ICEIS 2008*), Barcelona, 2008.

[WLK04a] Y. Wautelet, L. Louvigny, M. Kolp, "Le Unified Process comme méthodologie de gestion de projet informatique. Eléments d'application en milieu sidérurgique", *Working Paper IAG 109/04*, *Université catholique de Louvain*, 2004.

[WLK04b] Y. Wautelet, L. Louvigny, M. Kolp, "Modélisation Orienté-Objet d'aspects opérationnels de base de données sidérurgique", *Working Paper IAG 111/04*, *Université catholique de Louvain*, 2004.

[WKA06] Y. Wautelet, M. Kolp and Y. Achbany. "S-Tropos: An Iterative SPEM-Centric Project Management Process". *Working Paper IAG 06/01*, *Université catholique de Louvain*, 2006.

[YS95] M.Y. Yoshino, and U. Srinivasa Rangan. "Strategic alliances: an entrepreneurial approach to globalization". *Harvard Business School Press*, 1995.

[Yu95] E. Yu. "Modeling Strategic Relationships for Process Reengineering". *PhD thesis*, *University of Toronto*, *Department of Computer Science*, Canada, 1995.

6 Case Study

In this chapter the I-Tropos methodology is validated onto the development of an enterprise information system. Indeed, a production management software system of a coking plant is being built for a steel production company located in the Walloon region to provide its users a powerful tool for information management, process automation, resource and production planning, decision making, etc. Production of coke is one of the steps of steel making; the reasons for the choice of such a case study are exposed in section 6.1. Section 6.2 briefly explains the steps in steel production. Main project characteristics are depicted in section 6.3. Developments made during the four project phases are depicted in sections 6.4 to 6.7. Finally, section 6.8 concludes the chapter.

6.1. Agents in the Steel Industry

First of all one question must firstly be answered: *How can an industrial domain such as the steel industry that seems to belong to the past be interested in agent technologies?* In other words why agent-oriented modeling (and development) would be more indicated than traditional – possibly object – technologies.

The steel industry is by essence an agent-oriented world. Indeed, factories as a coking plant or a blast furnace are made of hundreds of different types of agents: software agents, machines, automates, humans, sensors, releases, effectors, controllers, pyrometers, mobile devices, conveying belts, etc. These are agents in the sense that:

- They are autonomous and dedicated to specific tasks;
- They are situated in a physical environment;
- They can act upon their environment if this is necessary by warning users, proposing solutions or taking autonomous action.

That is why agent-oriented modeling of such plants leading to the development of adequate software applications can provide advantages to the company especially in domains like information management, process automation, resource and production planning, decision making, etc.

The interested reader can find a complete RUP/UML development of a production management system for the coking plant in [WLK04/109]. Further developments of this case study have been realized in [WLK04/111, HKW06/03].

6.2. The Steel Production Process

The aim of a steel manufacturing company like the one being worked on is to extract iron from the ore and to turn it into semi-finished steel products. Several steps compose the transformation process, each step is generally assumed by a specific metallurgic plant:

- *Sintering Plant.* Sintering is the preparation of the iron ore for the blast furnace. The minerals are crushed and calibrated to form a sinter charge;
- *Coking Plant.* Coal is distilled (i.e., heated in an air-impoverished environment in order to prevent combustion) to produce coke;
- *Blast Furnace.* Coke is used as a combustion agent and as a reducing agent to remove the oxygen from the sinter charge. The coke and sinter charge are loaded together into the blast furnace to produce cast iron;
- *Steel Making Plant.* Different steps (such as desulphuration, oxidation, steel adjustment and cooling) are necessary to turn cast iron into steel slabs and billets. First, elements other than iron are removed to give molten steel. Then supplementary elements (such as titanium, niobium and vanadium) are added to make a more robust alloy. Finally, the result - finished steel - is solidified to produce slabs and billets;
- *Rolling Mill.* The manufacture of semi-finished products involves a process known as hot rolling. Hot-rolled products are of two categories: flat (plates, coiled sheets, sheeting, strips,…) produced from steel slabs and long (wire, bars, rails, beams, girders,…) produced from steel billets.

6.3. The Project

As pointed out earlier, this project objective is to develop a production management software system of the coking plant of a steel production company located in the Walloon region to provide its users a powerful tool for information management, process automation, resource and production planning, decision making, etc.

The whole I-Tropos project profile is illustrated on Figure 6-1. The phases will be described onto the rest of this paper with focus on the most relevant I-Tropos *Disciplines*, *WorkDefinitions* and *Activities* performed. The process is iterative which implies that within an iteration each discipline is approached with variable effort. For readability purposes, we will present the project phase by phase and, for each of those, we will focus on the disciplines on which most effort has been spent. So that in the Setting phase the Organisational Modeling and the Requirements Engineering core disciplines are highlighted as well as all of the support disciplines. Similarly when presenting the Blueprinting phase, we will focus on Detailed Design. Imple-

mentation will be the discipline focused on into the Building phase, and finally the Deployment phase will be the focus of the Setuping phase.

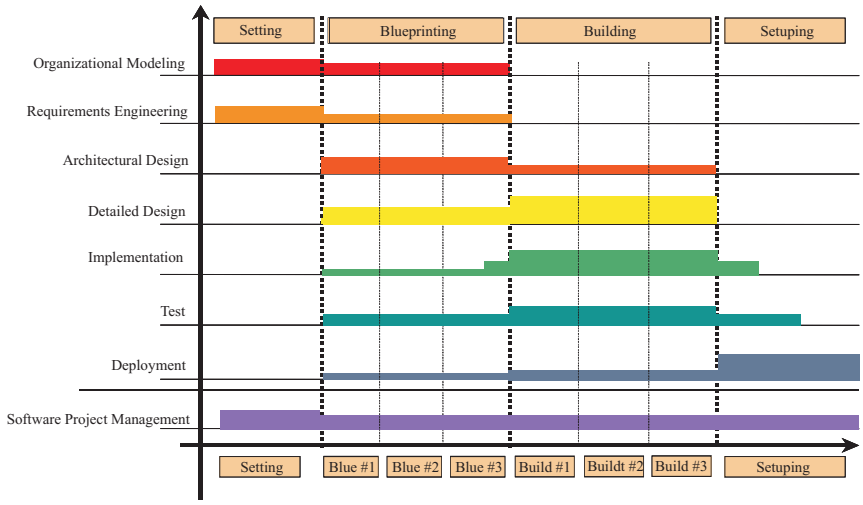


Figure 6-1. I-Tropos profile for the Carsid Coking Plant project.

6.4. The Setting Phase

During the Setting phase, considerable effort has been spent onto the core disciplines *Organizational Modeling* and *Requirements Engineering* as well as onto the support disciplines.

6.4.1. I-Tropos Core Disciplines

i* Strategic Dependency (SDD) and Strategic Rationale (SRD) Diagrams are used as WorkProducts by the I-Tropos software development process for organizational modeling and requirements engineering. Figure 6-2 represents the SDD while Figure 6-3 represents the SRD. Those models encompass both the actors, goals, tasks or resources identified when representing the organization as well as those identified when modeling user requirements. They are consequently WorkProducts produced into these two first process' disciplines. These diagrams have evolved later in the project, especially during the Blueprinting phase since at Setting only 95 per cent of user requirements are identified. The reader should keep in mind that we are facing an iterative process and that the diagrams are continuously subject to evolution on the basis of the feedback gathered during an iteration.

6.4.1.1. Organizational Modeling

The job of preparing coke from a suitable coal involves a long, hard process in which the coal is submitted to a series of successive transformations. The aim is to produce high quality coke while dealing with all the by-products generated during the different stages of the process. The coking plant is divided into three sections.

Preparation section: The admittance and primary treatment of the coal to be used to make the coke is the beginning of the chain in the coking plant.

Until the 80s the coal came from Belgian mines. However, the closure of these mines drove Cockerill to import the coal from other sources. It is shipped from the United States, Poland, South Africa and Australia to the ports of Antwerp and Rotterdam. The properties of each type of coal differ according to where it comes from. Three means of transport are used to transfer the coal from the ports to the plants: by road, by barges and by train, this last method being the most commonly used at the modeled coking plant.

The coals that arrive are stored either in a pit or in a bunker that serves as a security stock to provide protection against the difficulties that may arise from problems in supplies or deliveries. Use of this stock guarantees production for a maximum of about 15 days. Prolonged storage could in fact have a detrimental effect on the quality of the coal used, which would inevitably also have repercussions on the quality of the coke produced.

A series of transport instruments (buckets, conveyor belts, etc.) are used to carry the coal from the bunker to the storage bins. This transport is supervised by a machine that scans and detects the metals that might be present so as to prevent tears from occurring in the belt. The bins are divided into two categories according to whether they are storage or mixing bins. The latter can hold 250 tones and there are 10 of them. These allow the coals from different origins to be blended in order to obtain the required chemical characteristics. This blending is carried out under the bins: the desired amounts of coal are poured from each bin into buckets and from there onto a single conveyor belt which carries the mixture for milling. The storage bins have a capacity ten times larger than those used for mixing and can be used to fill the latter if needed.

The first treatment acts on the physical nature of the coal since it involves the crushing, or milling, of the load that has been carried from the mixing bins. After travelling along a series of conveyor belts, a mill crushes the different coals present in the blend in order to produce a homogeneous granulometry, which differs according to the origin of the coals. A metal detector identical to the one used when the raw coal was being transported is used to monitor the operations in order to protect the machinery. After milling, the product that is obtained is known as the “coal charge” and is carried on a series of belts from the milling tower to the coal tower. This tower is located above the ovens and has a capacity of 3 600 tones, that is, the amount needed for a day’s production.

Oven Section: Before beginning the process of actually distilling the coal charge, the charge must be transferred to the ovens. The team in charge of the ovens puts a certain amount of charge into a machine called a charging machine. This mechanism, which is situated above the oven batteries, then fills the ovens through openings in the top—the charge holes. These openings are in the shape of bottlenecks and each oven has four of them. During loading a "leveller bar", which is part of the pusher car, flattens the coal charge evenly throughout the oven to ensure a better distribution of the charge and a more homogeneous heating process. The loadings, or coking, follow a very precise schedule that attempts to maximize the number of charges performed per day.

Once charging is finished, the openings at the top and in the sides are closed and the process of firing the coal charge begins. The operation lasts between 16 and 19 hours, depending on the battery the oven is located in. It is, however, difficult to estimate the exact number of hours required for firing because of the numerous uncertainties linked with the ovens. Charges are performed in steps of five, which means that the procedure involves charging oven (n+5) after having charged oven (n). This method prevents large variations in temperature from taking place within the ovens.

The ovens are assembled in series called batteries. There are four of these and are made up of 20 to 50 ovens, which give a total assembly of 122 ovens at the CARSID coking plant. The properties of the ovens as regards theoretical firing time and the amounts that can be charged differ according to the battery the ovens belong to. The four batteries are lined up in a row with the coal tower situated in the middle. This layout allows machinery to circulate more easily from one oven to the next.

The reason why the ovens are arranged in batteries is that it helps to reduce heat losses in each of them. The space between the ovens, called a side wall, is made up of a series of vertical stacks which the gases pass through, allowing firing to take place inside the ovens. Thus, these stacks, called flues, help firing in two adjacent ovens. Heat exchange takes place by conduction without direct contact with the coal charge. The gases used come either from the coking plant or from the blast furnace, which enables the coking plant to remain independent on the energetic level. The combustion fumes are recovered by passing them through piles of refractory bricks during a first cycle. In a second cycle, the combustion gases are blown through them in order to recover their heat. Switching from one cycle to the other is called "inversion" and takes place every half an hour.

The oven reaches a temperature above 1200°C during firing, which allows the coal charge to be transformed into red-hot coke, which already possesses the required final characteristics. The gases released from the distillation process are recovered so that they can be evaluated.

Cooling Section: When firing finishes, the pushing operation, that is to say the emptying of the oven, begins. To perform this process, the side doors of the oven are opened and the pusher car pushes the oven load out. Then the coke guide, a mobile

toughed passageway, receives the charge and transfers it to a bogie, the quenching car, and the cooling operation begins. The quenching car is led towards the quenching tower and 25m³ of water is poured onto it. This must cool and quench the incandescent coke. Most of this water evaporates directly on contact with the coke. The waste waters are treated in a settling tank for later reuse. The coke is sprayed by hand should it be only partially quenched.

In the next stage the coke is "put on standby", which involves discharging it onto an inclined "coke wharf" and leaving it to lose some of its moisture. After a waiting period of about thirty minutes, the coke car transfers the coke obtained to the coarse coke tower. There, the screening process starts. This involves separating out the fragments considered to be "metallurgical coke", i.e. those that can be used in the manufacture of steel, from others that will be used for other purposes. The selection is performed according the size of the pieces, which must be big enough to provide the charge with good permeability. The fragments that are considered to be suitable for metallurgical use are sent to the blast furnace, while the "small coke" is carried off to the sinter.

Certain reparation tasks must be carried out on the ovens. Teams of builders are responsible for filling cracks and mending other problems that may arise concerning the ovens. These workers use them and know the problems that can come about in the bricks that line them. Repairing an oven is done at a temperature of 700°C. It is impossible to go below this temperature as the bricks would change their state and become as brittle as glass. Repair work is therefore a serious, difficult job, bearing in mind the high temperatures that must be dealt with.

Finally, there is a measurement and adjustment team that is in charge of supervising the proper functioning of the coke production machinery. The temperature of the ovens is controlled by measuring the heat present in the flues of the ovens, and can therefore be adjusted. This team is also in charge of inverting the heating cycle of the battery, which takes place about every half an hour.

This production process has been represented in the SDD of Figure 6-2 and the SRD of Figure 6-3[1] with the different elements defined by those models.

- Human actors represented as circles, at organizational modeling we distinguish:
    - *FADS Team*: the team in charge of handling coal reception;
    - *Oven Team*: the team in charge of baking coal charge;
    - *Setting Team*: the team in charge of the oven setting and inversion;
    - *Set-Up Team*: the team in charge of loading the Larry Car.

[1] for readability purpose figure 6-2 and 6-3 have also been produced in Appendix B.

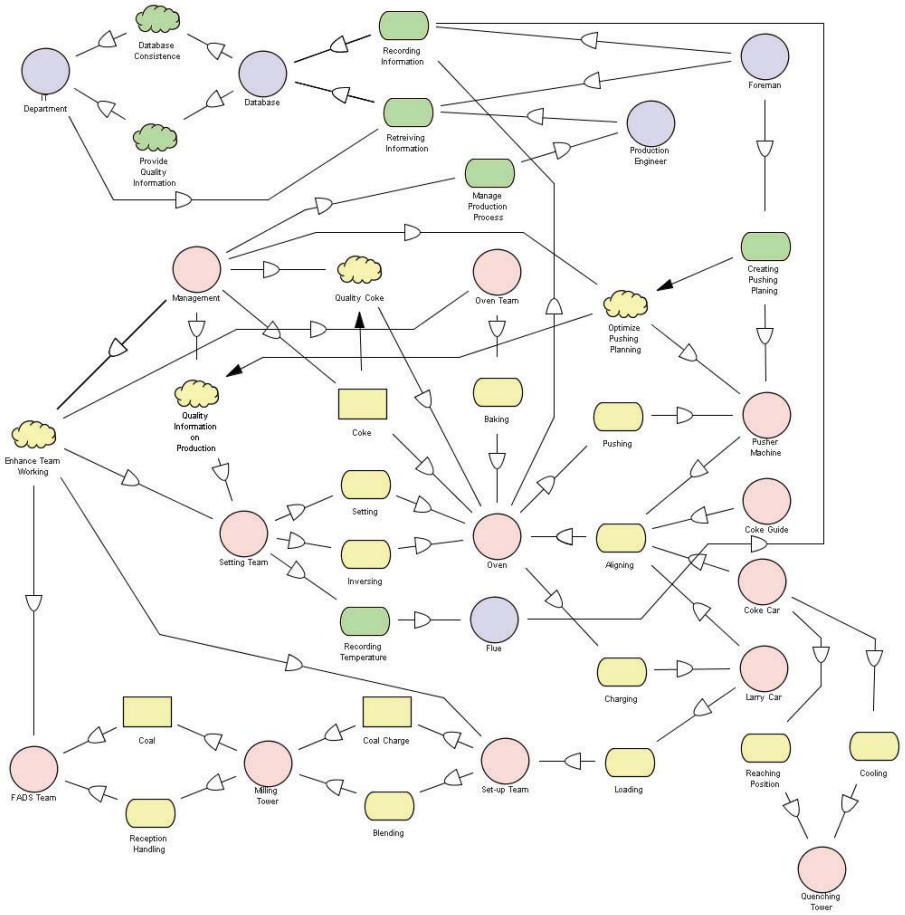


Figure 6-2. The Strategic Dependency Diagram developed during the Setting Phase

- Equipment actors also represented as circles, at organizational modeling we distinguish:
    - *Coke Car*: the wagon that moves the red-hot coke to the *Quenching Tower*;
    - *Coke Guide*: the mechanical device that guides the red-hot coke into the *Coke Car* when it is pushed out of the *Oven*;

- *Flue*: The flues are the stacks in the side wall, which is the space between two *Oven*s;
- *Larry Car*: the devices that fills the *Oven* with *Coal Charge*;
- *Milling Tower*: the tower that contains the *Coal* and that blends it to fill the *Larry Car*;
- *Oven*: the oven in which the *Coal Charge* is baked;
- *Pusher Machine*: the devices that pushes the red-hot coke out of the *Oven* into the *Coke Car*;
- *Quenching Tower*: the tower that contains the water used to cool down the red-hot coke contained in the *Coke Car*.

- Resources represented as rectangles, at organizational modeling we distinguish:
  - *Coal:* the raw coal received at the coking plant;
  - *Coal Charge:* the result from mixing and blending the different coals, that is to be baked in the *Oven*;
  - *Coke:* the final product that is to be used in the blast furnace;
  - *Position* (*Oven*): the *Oven* position, needed by *Larry Car*, *Pusher Machine* and *Coke Car*;
  - *Position* (*Quenching Tower*): the *Quenching Tower* position, needed by the *Coke Car*.

- Goals represented as rounded rectangles, at organizational modeling we distinguish:
  - *Aligning*: *Pusher Machine*, *Coke Car*, *Coke Guide* and *Larry Car* have to be aligned on the *Oven* position in order to do their tasks properly;
  - *Baking*: *Oven Team* supervises the baking the *Coal Charge* in the *Oven*;
  - *Blending*: *Milling Tower* blends *Coal* into *Coal Charge*;
  - *Charging*: *Larry Car* loads the *Oven* with *Coal Charge*;
  - *Cooling*: *Quenching Tower* cool down red-hot *Coke*;

  - *Inversing*: *Setting Team* activates inversion phase in an *Oven*;
  - *Loading*: *Set-up Team* loads the *Larry Car* with *Coal Charge*;
  - *Manage Production Process*: the *Management* manages the production process using *Production Engineers*;
  - *Pushing*: *Pusher Machine* pushes the red-hot *Coke* out of the *Oven* trough the *Coke Guide* into the *Coke Car*;
  - *Reaching Position*: the *Coke Car* must reach the *Quenching Tower* in order to cool down the red-hot *Coke*;
  - *Reception Handling*: *FADS Team* receives *Coal* form different suppliers and stocks it into the *Milling Tower*;
  - *Setting*: *Setting Team* arranges the setting of an *Oven*.

- Softgoals represented as clouds, at organizational modeling we distinguish:
  - *Quality Coke*: *Management* wants to insure that the *Coke* produced in the coking plant is good as the steel quality depends on the *Coke* quality;
  - *Enhance Team Working*: with all these teams on the coking plant, productivity directly depends on how teams are communicating and working together.

- Tasks represented as an hexagon, at organizational modeling we distinguish:
  - *Bake*: baking the *Coal Charge* into *Coke*;
  - *Blend*: blending the *Coal* to make *Coal Charge*. It is divided into three sub-tasks:
    - *Detect Metals*: detect metal from *Coal*, once more;
    - *Crush Coals*: crushing *Coal* into powder;
    - *Mix Coals*: mixing the crushed *Coal* to form *Coal Charge*.
  - *Charge*: the *Larry Car* fills the *Oven* with *Coal Charge*. It is divided in three sub tasks:
    - *Open Oven*: open the *Oven* upper door;

- *Load Oven*: fill the *Oven* with *Coal Charge*;
- *Close Oven*: close the *Oven* upper door.

- *Cool*: cool down the red-hot *Coke* that just finished *Baking*. It is divided into two sub-tasks:
  - *Drop Water*: drop a certain amount of water on the red-hot *Coke*;
  - *Recuperate Steam*: recuperate as much steam as it is possible, in order to use it again as water;
- *Inverse*: every half-hour, during the baking, the gas direction is inverted into the side-walls;
- *Push*: the *Pusher Machine* pushes the red-hot *Coke* out of the *Oven*. It is divided in three sub tasks:
  - *Open Oven*: open the *Oven* side doors;
  - *Push Load Out*: the Pusher machine's arm enters the *Oven*, pushing the load out;
  - *Close Oven*: close the *Oven* side doors.
- *Set*: the *Setting Team* sets the *Oven* for baking. These are the setting of the *Oven*;
- *Supply Coal*: *FADS Team* receives *Coal* from suppliers and delivers it to the *Milling Tower*. It is divided in three sub tasks:
  - *Coal Reception*: reception of *Coal* from suppliers;
  - *Detect Metals*: detecting metal into the received *Coal* and removing it;
  - *Transport Coal*: transporting the Coal to the *Milling Tower*.

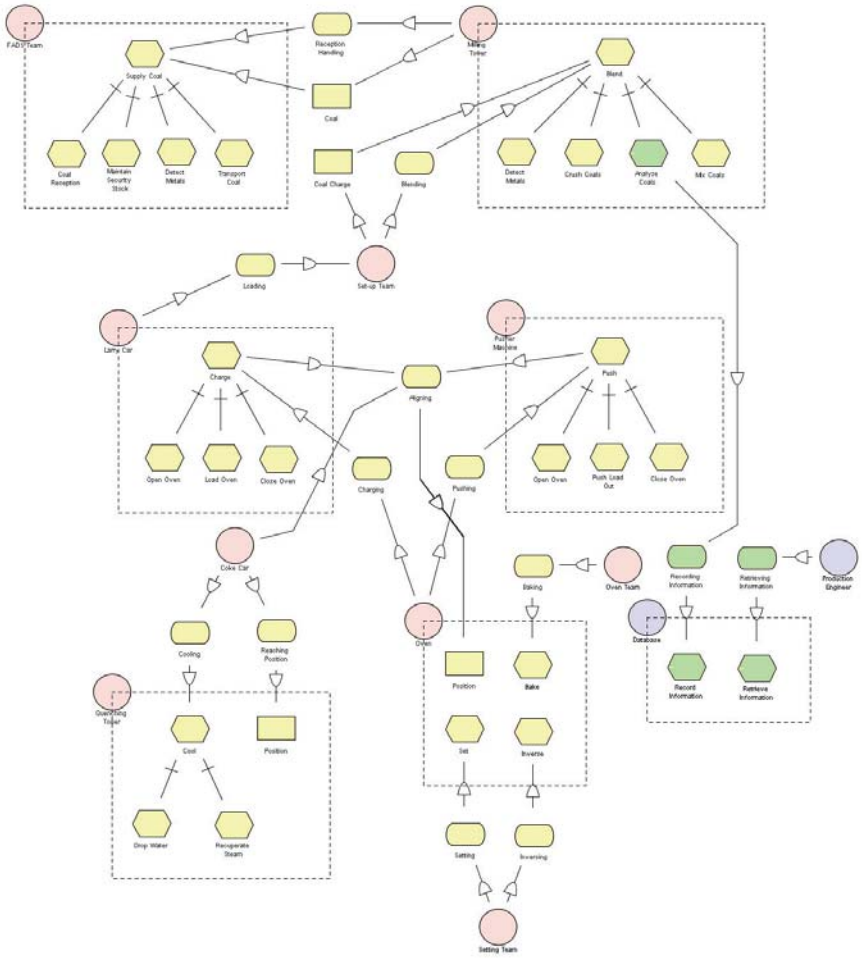


Figure 6-3. The Strategic Dependency Diagram developed during the Setting Phase

The organizational modeling discipline allowed us to find out the most important actors of the coking plant working on the production of coke through the achievement of multiple goals. The identified goals will be taken as fundamentals for software project management as will be described into the next sections.

6.4.1.2 Requirements Engineering

User requirements emerging during the multiple conducted interviews are described into this section. Those are mainly the production information and management tools. The example of *Creating Pushing Planning* is given hereafter.

At pause time, the *Foreman* launches a branch and bound model (called Patricio Villa model) to compute an optimal pushing sequence. If this model does not find any solution or if the *Foreman* is unsatisfied with the one created, the system launches a lightened model to produce a default planning. Once the sequence is set, the *Foreman* can modify it manually; the planning is finally recorded into the *Database*. More information can be found in [WLK04/111].

This production process has been represented in SDD and SRD with the different elements defined by those models.

- Human actors represented as circles, at requirements engineering we distinguish:
    - *Foreman*: the man responsible for planning the pushing sequence;
    - *IT Department*: the team that manages the database and production information;
    - *Management*: the human resources in charge of managing the coking plant;
    - *Production Engineer*: the man responsible of managing the production process.
- Equipment actors also represented as circles, at requirements engineering we distinguish:
    - *Database*: used to store and retrieve all the information on production.
- Goals represented as rounded rectangles, at requirements engineering we distinguish:
    - *Creating Pushing Planning*: during each pause, the *Foreman* sets-up a planning of the pushing order for the next pause;
    - *Manage Production Process*: the *Management* manages the production process using *Production Engineers*;
    - *Pushing*: *Pusher Machine* pushes the red-hot *Coke* out of the *Oven* trough the *Coke Guide* into the *Coke Car*;
    - *Recording Information*: recording information into the *Database*;
    - *Recording Temperature*: during the baking period, *Setting Team* may ask the *Flues* to record their temperature;

- *Retrieving Information*: retrieving information from the *Database*.

- Softgoals represented as clouds, at requirements engineering we distinguish the:

  - *Database Consistence*: in order to accurately plan and manage, the information in the *Database* has to be accurate and consistent;

  - *Provide Quality Information*: information on production has to be accurate, which is the responsibility of the *IT Department*;

  - *Quality Information on Production*: in order to effectively manage the coking plant, *Management* wants to be provided with quality information on production;

  - *Optimize Pushing Planning*: one on the very important aspect of the coking plant productivity is how pushing is planned. This directly depends on the quality of the information available.

- Tasks represented as an hexagon, at requirements engineering we distinguish the:

  - *Maintain Security Stock*: a security stock is kept in order to provide protection against the difficulties that may arise from problems in supplies or deliveries. It is one of the sub-tasks of *Supply Coal*.

6.4.2. Risk Management

The Risk Management discipline uses the goals identified into the organizational modeling and requirements analysis disciplines as fundamental scope elements. Consequently the identified threats are envisaged on the basis of their impact on them. Risk analysis was done in collaboration with stakeholders estimating and validating the possible threats impact. Risk quantification is done on a double basis, at first the general threat weight is estimated on the basis of the impact it can have on the project under development, system produced or organization concerned. At second stage the involvement of each goal with respect to the risk evoked is evaluated following a low/medium/high scale. That work led to the identification of six categories of threats:

- **Requirements poorly understood**: the system does not what the users expect it to do. This threat is particularly faced by user intensive software applications. This kind of risk has a weight of 3 because huge resources can be devoted to produce an inadequate system;

- **Facility damage**: damage caused to production facility. A good example is when the pusher machine pushes while there is hot coke stuck on the oven's

walls, resulting in damaging the pusher machine and/or the oven's walls. This kind of risk has a weight of 2 because it is costly to repair facilities;

- **Mechanical error**: error due to machinery dysfunction. A failure in the coke car engine, making impossible to cool down coke, is an example. This risk has a weight of 1 because it will delay production;

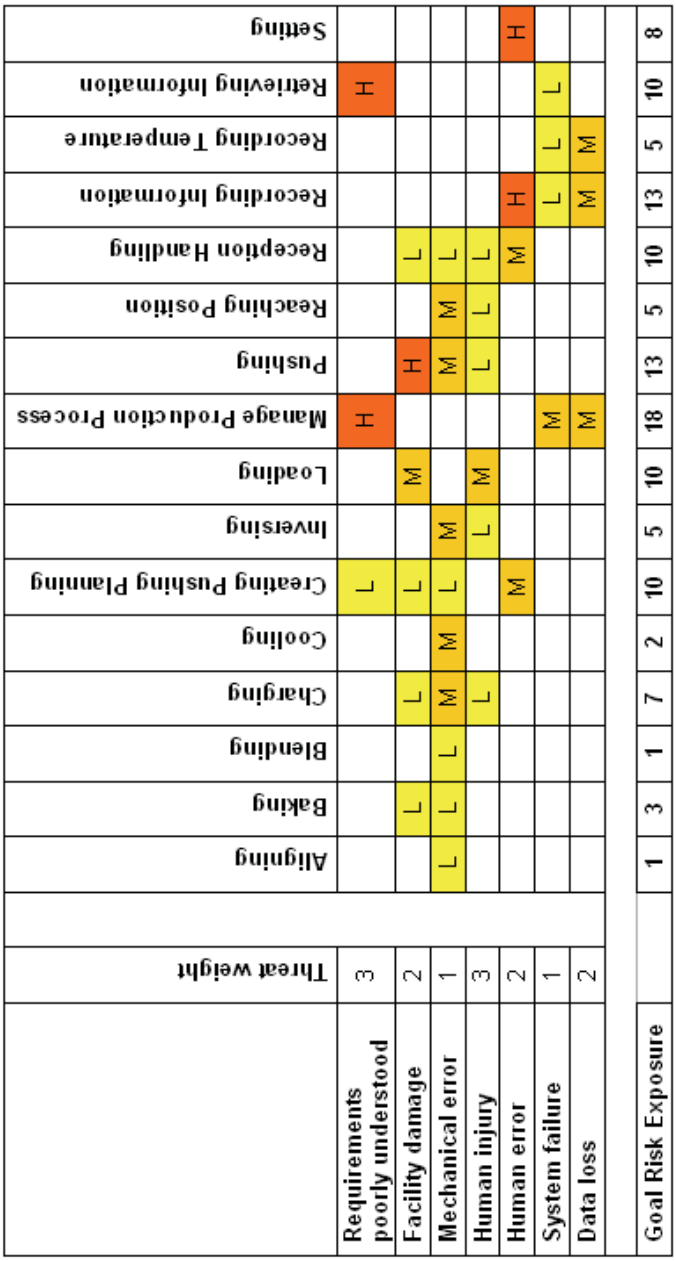
- **Human injuries**: injury (or even death) of the personal. A coking plant is a hazardous place; a replacement worker was recently killed in a coke plant in Ohio, and other numerous accident of this type took place in the past. This risk has a weight of 3 because it is very costly in terms of money and reputation;

- **Human error**: error due to human intervention. This risk has a weight of 2 because it can potentially lead to other risks.

- **System failure**: the information system unusable. The system may be down and cannot be used for a certain amount of time. This risk has a weight of 1 because it will delay production.

- **Data loss**: some needed data is lost or never existed. It may happen if one of the worker forgets to encode some data. This risk has a weight of 2 because it will delay production and can potentially lead to other risks.

Using requirements identified in the SSD and the risk list depicted above, Table 6-1 summarizes the impact of the identified threats onto the modeled goals. When a particular goal is facing a particular threat it is marked and a measure is given, for example the goal *aligning* faces the threat *mechanical error* even if the probability of attendance of the threat on the goal remains low. The overall risk exposure of each goal is finally computed on the basis of the threats they faced, the intensity and the threat's weight.



	Threat weight	Aligning	Baking	Blending	Charging	Cooling	Creating Pushing Planning	Inversing	Loading	Manage Production Process	Pushing	Reaching Position	Reception Handling	Recording Information	Recording Temperature	Retrieving Information	Setting
		1	3	1	7	2	10	5	10	18	13	5	10	13	5	10	8
Requirements poorly understood	3						L			H						H	
Facility damage	2		L		L		L		M		H		L				
Mechanical error	1	L	L	L	M	M	L	M			M	M	L				
Human injury	3				L			L	M		L	L	L				
Human error	2						M						M	H			H
System failure	1									M				L	L	L	
Data loss	2									M				M	M		
Goal Risk Exposure																	

Table 6-1. Project Goals Risk Exposure

6.4.3. Quality Management

The Quality Management discipline uses the goals identified into the organizational modeling and requirements analysis disciplines as fundamental scope elements. Consequently the identified quality factors are envisaged on the basis of their impact on them. Quality analysis was done in collaboration with stakeholders estimating and validating the possible quality factors impact. That work led to the identification of six categories of quality factors:

- **Reliability**: Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. This quality factor deals with the question: *What level of confidence can be placed in what the system does?*

- **Efficiency**: Software Efficiency deals with execution time, the later can be influenced by source code optimization, the use of performing algorithmic techniques, faster sequential execution or code parallelization. This quality factor deals with the question: *How well are resources utilized?*

- **Usability**: Software Usability is the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. This quality factor deals with the question: *How easy is it to use?*

- **Integrity**: Software Integrity is the ability of software to resist, tolerate and recover from events that threaten its dependability. This quality factor deals with the question: *How secure is it?*

- **Testability**: Software testability is a software characteristic that refers to the ease with which some formal or informal testing criteria can be satisfied. This quality factor deals with the question: *How easy is it to verify conformance to requirements?*

- **Flexibility**: Software flexibility is the ability of a software system to adapt change. This quality factor deals with the question: *How easy is it to modify?*

- **Interoperability**: Software interoperability is defined as the ability for multiple software components written in different programming languages and distributed across multiple platforms to communicate and interact with one another. This quality factor deals with the question: *How easy is it to interface with another system?*

	Quality Factor Weight	Aligning	Baking	Blending	Charging	Cooling	Creating Pushing Planning	Inversing	Loading	Manage Production Process	Pushing	Reaching Position	Reception Handling	Recording Information	Recording Temperature	Retrieving Information	Setting
		16	3	1	11	5	31	1	5	32	20	16	1	30	18	26	14
Reliability	3	M			L		M			M	H	M		H	M	H	L
Efficiency	2	L			L		H			H	M	L		L	L	M	M
Usability	1						M			M						M	
Integrity	3						L			M				H	M	M	
Testability	1		L	L		L	M	L	L	M		M	L			M	L
Flexibility	2	M			M	L	H		L	M	M	M					M
Interoperability	2	M	L		L	L	L		L	M		L		M	M		L
Goal Quality Exposure																	

Table 6-2. Project Goals Quality Exposure

Using requirements identified in the SDD and the quality factors list depicted above, Table 6-2 summarizes the impact of the identified quality factors onto the modeled goals. When a particular goal is facing a particular quality factor it is marked and a measure is given, for example the goal *aligning* faces the quality factor *efficiency* even if the concern remains low. The overall quality factor exposure of each goal is finally computed on the basis of the quality factors they faced, the intensity and the quality factor's weight.

6.4.4. Time Management

In the two previous sections we studied the threats and the quality factors impact on the goals of the SDD. In this section, we will use the global risk and quality exposures to determine a goal priority. Indeed, facing an iterative life cycle we have to plan the goals realization. Goals with highest priority will firstly be designed, prototyped and tested during the Blueprinting phase and firstly implemented during the Building phase. This allows scrutinizing the trickiest issues first so that risks are taken earlier onto the project when corrective actions are the easiest to put into practice.

Computations are summarized in Table 6-3, the method used is the following:

- On the basis of the Overall Risk Exposure, the Relative Risk Exposure is computed using the formula: Relative Risk Exposure = Overall Risk Exposure/Total Risk Exposure;

- On the basis of the Overall Quality Exposure, the Relative Quality Exposure is computed using the formula: Relative Quality Exposure = Overall Quality Exposure/Total Quality Exposure;

- The Priority Level is computed by balancing the Relative Risk Exposure and the Relative Quality Exposure. For this particular project we chose a repartition key of 75 per cent for the risk component and 25 for the quality component. This repartition key can vary from one project to another and should be calibrated on the basis of data collected from a large number of projects as well as by the working team experience;

- On the basis of the Priority Level, the goals priority is deduced.

Thanks to the priority list, the precedence constraints and the estimated goal complexity we determine the iteration plan of table 6-4. Goal complexity is an estimation of the amount of effort required to develop it. Effort evaluation of the development of a coking plant information system using use case points and COCOMO II is given in [WK06]. The I-Tropos methodology uses three categories of goal complexity, i.e., Low/Medium/High, owning respectively a weight of 5/10/15. Transforming overall weight to man-mount requires calibration on the basis of a large number of projects and remains an open issue. The proposed planning will be subject to modifications/reviews into the software project because of process' iterative nature.

	Goal	Goal Risk Exposure	Relative Risk Exposure	Goal Quality Factors Concern	Relative Quality Factors Concern	Goal Priority Level	Goal Priority Rank	Goal Complexity	Goal Complexity Weight	Precedence
1	Aligning	1	0,008264463	16	0,069565217	0,023589651	13	L	5	
2	Baking	3	0,024793388	3	0,013043478	0,021855911	14	L	5	
3	Blending	1	0,008264463	1	0,004347826	0,007285304	16	M	10	
4	Charging	7	0,05785124	11	0,047826087	0,055344951	9	M	10	
5	Cooling	2	0,016528926	5	0,02173913	0,017831477	15	M	10	
6	Creating Pushing Planning	10	0,082644628	31	0,134782609	0,095679123	4	H	15	G4, G10
7	Inversing	5	0,041322314	1	0,004347826	0,032078692	12	L	5	
8	Loading	10	0,082644628	5	0,02173913	0,067418254	6	M	10	
9	Manage Production Process	18	0,148760331	32	0,139130435	0,146352857	1	H	15	
10	Pushing	13	0,107438017	20	0,086956522	0,102317643	3	H	15	
11	Reaching Position	5	0,041322314	16	0,069565217	0,04838304	11	M	10	
12	Reception Handling	10	0,082644628	1	0,004347826	0,063070428	8	L	5	
13	Recording Information	13	0,107438017	30	0,130434783	0,113187208	2	L	5	
14	Recording Temperature	5	0,041322314	18	0,07826087	0,050556953	10	L	5	
15	Retrieving Information	10	0,082644628	26	0,113043478	0,090244341	5	M	10	G13
16	Setting	8	0,066115702	14	0,060869565	0,064804168	7	M	10	
	Sum	121	1	230	1	1			145	

Table 6-3. Goal Prioritization.

	Goal	Priority	Complexity	Blueprinting 1	Blueprinting 2	Blueprinting 3	Building 1	Building 2	Building 3
1	Manage Production Process	1	H	X			X		
2	Recording Information	2	L	X			X		
3	Pushing	3	H	X			X		
4	Charging	9	M	X			X		
5	Creating Pushing Planing	4	H		X			X	
6	Retrieving Information	5	M		X			X	
7	Loading	6	M		X			X	
8	Setting	7	M		X			X	
9	Reception Handling	8	L		X			X	
10	Recording Température	10	L			X			X
11	Reaching Position	11	M			X			X
12	Inversing	12	L			X			X
13	Aligning	13	L			X			X
14	Baking	14	L			X			X
15	Cooling	15	M			X			X
16	Blending	16	M			X			X
	Total Effort Weight		145	15	16,6666 7	16,6666 7	30	33,3333 3	33,3333 3

Table 6-4. Iteration Plan.

6.4.5. Software Process Management: Process Tailoring

In the previous section, goals were prioritized and planned for practical realization into the multiple iterations. The activities performed during each iteration depend however on the iteration's goals focus. Blueprinting iterations are mostly focused on application architecture and prototyping while Building iterations are aimed to build fully executable functionalities. For readability purposes, we will not detail each activity of the process but rather detail the effort spent on the disciplines' WorkDefinitions for the first goal of the first Blueprinting iteration, this can be found into table 6-5.

Roles relative involvement into the first goal of the first Blueprinting iterations is also detailed in this section (table 6-6).

Iter.	Goal	Discipline	WorkDefinitions	Effort
Blueprinting 1	**Manage Production Process**	**Organizational Modeling**	Stakeholder Analysis	M
			Strategic Analysis	M
		Requirements Engineering	System Environment Analysis	M
			Strategic Analysis	M
		Architectural Design	System Architecture Analysis	H
			System Architecture Design	H
		Detailed Design	Social Analysis	M
			Agent Behavior Description	M
		Implementation	Model Structuration	L
			Model Implementation	L
		Test	Test Management	H
			Testing	H
		Deployement	Deployment Management	L
			Physical Deployment	None

Table 6-5. Process Tailoring: WorkProducts Effort for Two Goals in Blueprinting 1.

Iter.	Goal	Discipline	WorkDefinitions	Effort
Blueprinting 1	Manage Production Process	**Organizational Modeling**	Organizational Analyst	M
			Stakeholder	M
		Requirements Engineering	System Analyst	M
		Architectural Design	Software Architect	H
			Analyst	M
			System Specifier	L
		Detailed Design	Software Architect	M
			Functional Analyst	M
			Agent Designer	L
		Implementation	Developer	L
			User Interface Designer	H
		Test	Test Manager	H
			Tester	H
			End User	H
		Deployement	Deployement Manager	L
			Developer	None
			User Manuel Developer	None
			Installer	None

Table 6-6. Process Tailoring: Roles Effort for Two Goals in Blueprinting 1.

6.5. The Blueprinting Phase

The Blueprinting phase has been planned into the previous section. It mostly focuses on the design, prototyping and testing of the goals identified into the Setting phase, consequently considerable effort has been spent onto the core discipline *Detailed Design*. This section briefly overviews the WorkProducts produced during this disciplines.

At detailed design stage, the system's design is fully documented with different types of diagrams. No design pattern was selected and the following diagrams were produced:

- Communication Diagrams are used to express the dynamic of events and agents, in a temporal manner;

- Dynamic Diagrams offer a view which is close from Communication Diagrams, but with more details on the internal of agents. It depicts plans taken by agents in response for externals and internals events received;

- Structural Diagrams depicts a static view of agents, with their related events and plans. It is build using previously made Dynamic Diagrams, adding beliefs to the agents. Beliefs are what the agent know, and may be shared with other agents of the system (noted as "global belief" when shared).

A brief technical description of the produced diagrams is presented into the rest of this section.

6.5.1. Communication Diagrams

The Communication Diagram of Figure 6-4 (for readability purpose the figure has also been produced in Appendix B) illustrates the whole coke production process. It starts with the arrival of coal delivered by a supplier, and stops with the cooling of the red-hot coke at the quenching tower. It highlights events that are sent from on agent to one or more other agent. Internal events are not represented on this first version of our diagram.

The events of the Communication Diagram are depicted hereafter:

- *CoalArrived*: when a coal supplier delivers so coal at the coking plant, the FADS handles the reception;

- *CoalDelivered*: when the coal has been transferred to the *Milling Tower*, it starts the blending process;

- *CoalChargeReady*: once the *Coal Charge* is ready, the *Larry Car* can be loaded with it;

- *LoadingCompleted*: when the *Larry Car* is loaded, it moves to the *Oven* to fill it;

- *ChargingCompleted*: once the *Oven* is filled with *Coal Charge*, the *Oven* waits for setting to be completed;

- *SettingCompleted*: when settings are completed, the *Oven* is ready;

- *Ready*: when the *Oven* is ready, *Oven Team* can order it to start baking;

- *StartBaking*: when the *Oven* is requested to start baking, it starts;

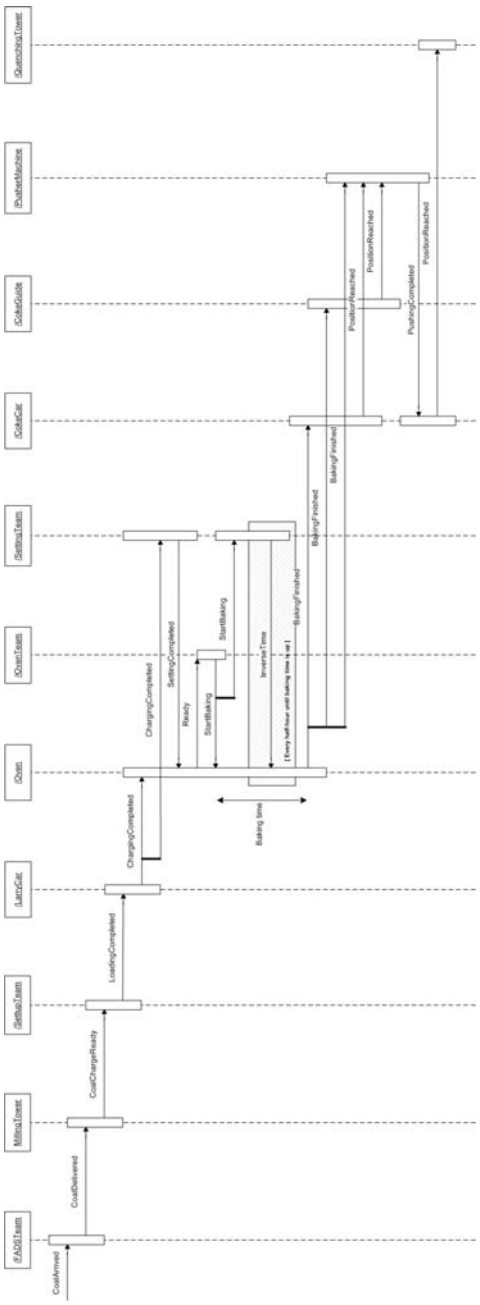


Figure 6-4. Communication Diagram, Blueprinting Phase.

- *InverseTime*: every half hour, the *Oven* is requested to do an “inversion”;
- *BakingFinished*: when the baking is finished, the *Oven* notices the *Coke Car*, the *Coke Guide* and the *Pusher Machine*;
- *PositionReached*: when a device reached the position it was moving towards;
- *PushingCompleted*: once the pusher machine pushed the entire *Oven* load out.

6.5.2. Dynamic Diagrams

During the first iteration, a total of six diagrams were made. Due to a lack of space, we will only present the first diagram, illustrated on Figure 6-5, it models the whole coke production process. It is a more detailed view than the one given by the Communication Diagram established earlier.

6.5.3. Plans

Tasks represented in the Strategic Rationale Model were represented using only one plan in the global Dynamic Diagram illustrated on Figure 6-5. However, as these tasks were composed of sub-tasks, we decided to illustrate those using detailed Dynamic Diagrams.

Here are the plans presented on Figure 6-5 (for readability purpose the figure has also been produced in Appendix B):

- *SupplyCoal*: when coal is received from suppliers, the *FADS Team* handles the coal reception and moves it to the *Milling Tower*. This plan is composed of several other plans, each plan corresponds to as task identified in the Strategic Rationale Model;
- *Blend*: when *Coal* arrives at the *Milling Tower*, it is blended to form *Coal Charge*. This plan is composed of several other plans;
- *Load*: once the *Coal Charge* is ready, it is loaded into the *Larry Car*;
- *Charge*: when the *Larry Car* is loaded an *Oven* is ready for baking, the *Oven* is filled with *Coal Charge*. This plan is composed of several other plans;
- *SettingOven*: once the *Oven* is filled, the *Setting Team* has to enter *Oven*’s setting;
- *ReadyForBaking*: when the settings are entered, the *Oven* is ready and waits for the signal to start baking;

- *OvenReady*: the *OvenTeam* is noticed that one of the *Ovens* is ready. It waits for human to order the *Oven* to start the baking process;

- *Bake*: the *Oven* is baking its content;

- *OventBaking*: the *SettingTeam* is noticed that one of the Ovens has started baking;

- *Inverse*: the *Oven* inverses the gas circulation in the side walls;

- *ReadyForPushing*: when the baking process is finished, the *Oven* waits for its content to be pushed;

- *MovingToPosition*: it concerns more than one agent. It is simply a mechanical device moving to reach a certain position, for example the *Pusher Machine* moving to reach the *Oven* position in order to push its content out;

- *Push*: Once the three needed devices (*CokeGuide*, *CokeCar* and *PusherMachine*) are in position, the *PusherMachine* starts pushing the content of the *Oven* out, trough the *CokeGuide* into the *CokeCar*. This plan is composed of several other plans;

- *MoveToQuenchingTower*: once the red-hot *Coke* is in the *CokeCar*, the *CokeCar* moves under the *QuenchingTwoer*, where it waits;

- *Cool*: once the *CokeCar* is in position, the *QuenchingWater* drops a certain amount of water on the red-hot Coke, and recuperates as much steam as possible. This plan is composed of several other plans.

6.5.4. Structural Diagrams

A sample of the Structural Diagram elaborated during the first Blueprinting iteration is illustrated on Figure 6-6. The agents depicted come from the various iterations of the phase, at this stage, 95 per cent of the architecture is fixed.

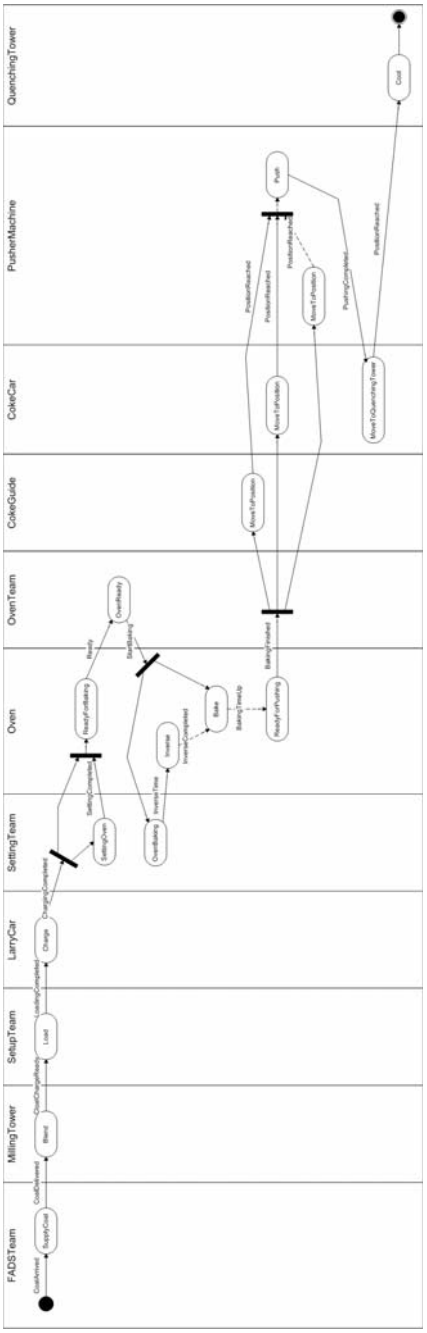


Figure 6-5. Dynamic Diagram, Blueprinting Phase.

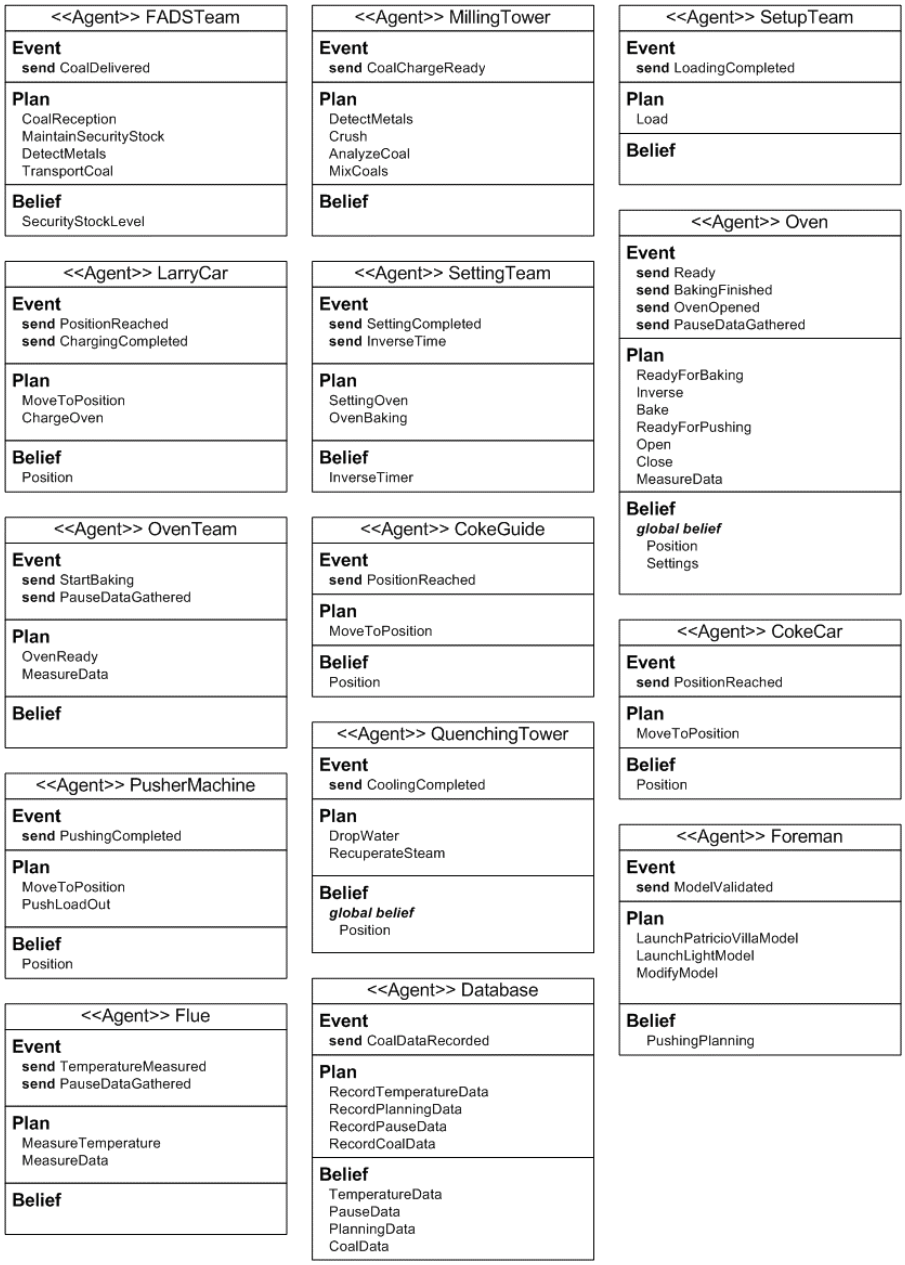


Figure 6-6. Sample of the Structural Diagram, Blueprinting Phase.

6.6. The Building Phase

In the Building phase, considerable effort has been spent onto the core discipline *Implementation* to implement the whole of the system's functionalities prototyped during the Blueprinting phase on which user feedback was received. This section briefly overviews the artifacts produced during those disciplines.

The *Oven Application* itself is described here with focus on the role of the *agents* and how they interact. Implementation is based onto the analysis and design disciplines exposed previously.

When a user gets connected to *Oven Application*, the User is instantiated and displays the interface depicted on figure 6-7. It allows the new coming user to register on the application (1). The information provided by the users is handled by the Database agent who checks the validity (2). Once this has been done, the production engineer can perform different types of actions related to the ovens. At any moment during the session the user can use the navigation-bar (3) to switch from one to another section.

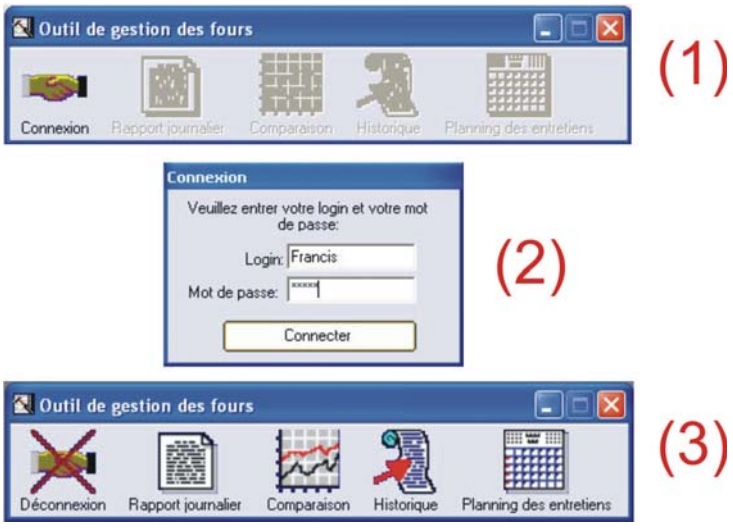


Figure 6-7. Oven Application Logon Window, Building Phase.

Access on the information on the ovens is managed by the Oven agent. Daily reporting (Figure 6-8) and oven maintenance management (Figure 6-9) are part of the strategic behaviour. Reporting is initiated by the Production Engineer on the basis of the strategic information provided by the Pusher Machine and Ovens.

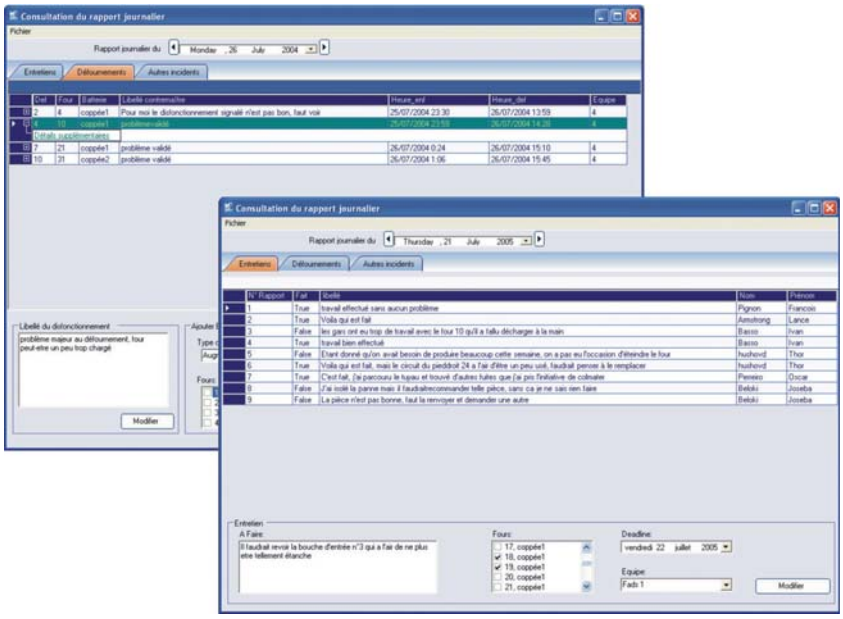


Figure 6-8. Daily Reporting Interface, Building Phase.

The maintenance policies are initiated by the Production Engineer from the strategic information provided by the Foreman.

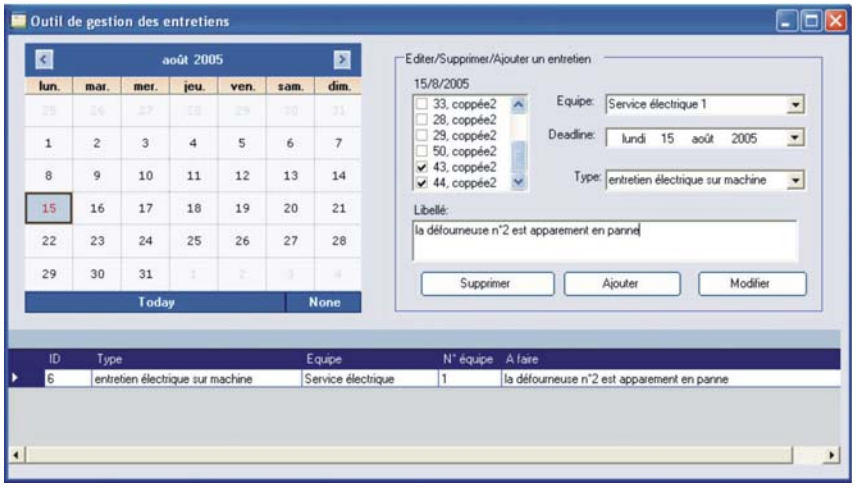


Figure 6-9. Daily Reporting Interface, Building Phase.

The Production Engineer selects the best pushing information and visualizes them into the comparison interface (Figure 6-10). The Production Engineer acts in the same way for oven comparison, the Pusher_Machine computes the pushing effort and the Production Engineer is in charge of updating the comparison interface.

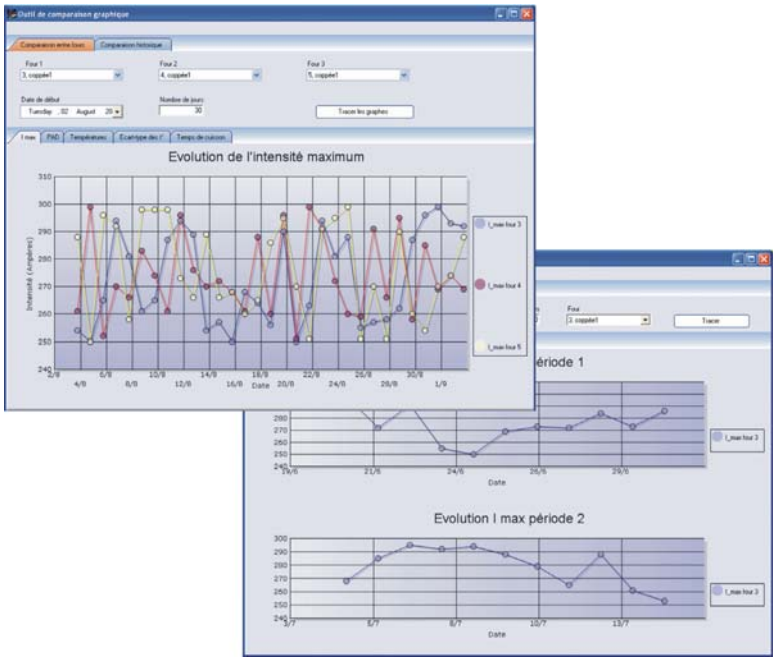


Figure 6-10. Daily Reporting Interface, Building Phase.

Figure 6-11 describes the oven maintenance report interface when the "oven maintenance" button of the navigation-bar is activated. Facing this interface, the users can fill one or several fields concerning maintenance (what should be done, which are the concerned ovens). The Production Engineer sends the query parameters to the Oven Team which provides the results to the Production Engineer. At any moment during the session, if the user selects a specific maintenance (id, date-hour, type,…) a request is sent to Oven Team to provide more information on this maintenance.

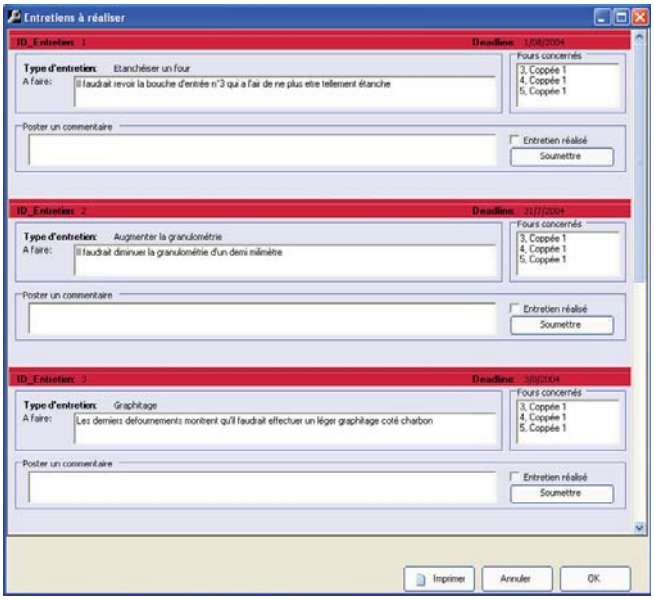


Figure 6-11. Oven Maintenance Reporting Interface, Building Phase.

6.7. The Setuping Phase

The Setuping phase is mostly concerned with the physical deployement of the application on site. User training is also an important aspect of this phase. The I-Tropos Deployment discipline has the most effort spent on during that phase, that is why we will present here the deployment diagram which is the WorkProduct being produced at that stage.

The physical deployment of the components described above is illustrated in Figure 6-12 done in the following manner:

- Application Server: the whole of the graphical components, the different control components (maintenance manager, detection and alert manager, graphical manager and the query manager) as well as the domain dependant storage components are installed onto this server;

- Identification Server: only the secured access manager is installed onto this machine;

- Database Server: the database component is located on that machine;

- Client Application Server: local computer of the Production Engineer, Foreman, etc.

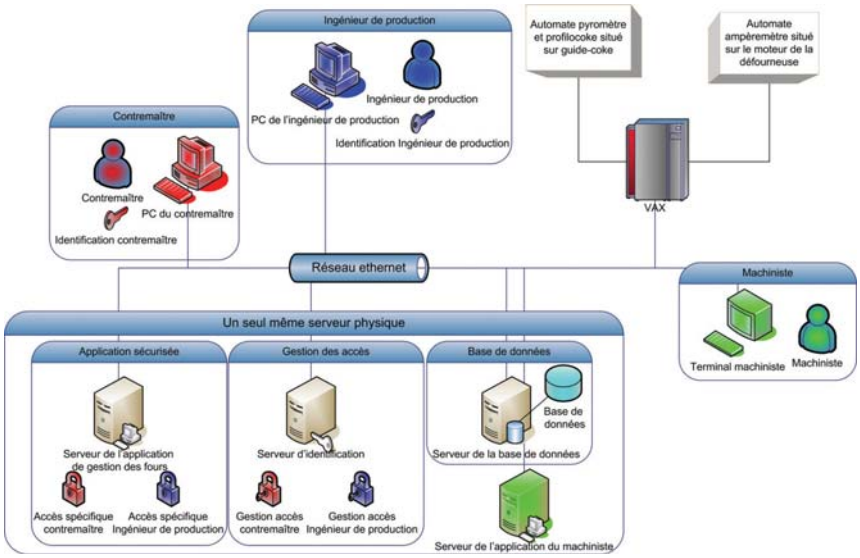


Figure 6-12. Deployment Diagram.

6.8. Chapter Summary

This chapter has presented a MAS management system of a coking plant developed using the framework proposed in Chapter 5. The application has applied several of the I-Tropos core disciplines presented here through the developed WorkProducts (i* models, communication diagrams, structural diagrams, GUI, etc.). A strong accent has been put on the support disciplines showing the application of the I-Tropos software project management framework.

References

[HKW06/03] C. Herssens, M. Kolp and Y. Wautelet. “Analyse et développement d'une application de gestion des fours à coke : application à Carsid”. Working Paper IAG 06/03, Université catholique de Louvain, 2006.

[WK06] Y. Wautelet, M. Kolp. “A Project Cost and Time Estimation Using Use Case Points and COCOMO II: The Coking Plant Case”. Technical Paper, Université catholique de Louvain, 2006.

[WLK04/109] Y. Wautelet, L. Louvigny and M. Kolp. “Le Unified Process comme méthodologie de gestion de projet informatique. Eléments d'application en milieu sidérurgique”. Working Paper IAG 109/04, Université catholique de Louvain, 2004.

[WLK04/111] Y. Wautelet, L. Louvigny and M. Kolp. “Modélisation Orienté-Objet d'aspects opérationnels de base de données sidérurgique”. Working Paper IAG 111/04, Université catholique de Louvain, 2004.

7 DesCARTES-Architect: a CASE-Tool for I-Tropos Software Development

This chapter presents a CASE-Tool (called DesCARTES[1]) aimed to ease Tropos/I-Tropos software developments. DesCARTES was developed with respect to the agent-oriented models and concepts as well as the other project management features presented in Chapter 5. This tool addresses to analysts, designers, software architects, developers and even project managers as will be shown in the next section. On a general basis, it allows resources working onto the project to:

- Represent the models in a graphical way;
- Reuse the catalogue of architectural styles and other design patterns to construct the MAS;
- Generate code on the basis of design models;
- Manage the software project.

The tool was developed in Java as a plug-in of the Eclipse platform. The analysis of the tool will be presented here using i* diagrams at analysis stage, UML class diagrams will be used to illustrate the design stages. The interface has been developed by bachelor trainees students and other master students (Loic Desguin [Desguin05], Gaëtan Martel [Martel05], Marco Montoisis [Montoisis06], Frédérique Jenquin [Jenquin06] and Martin Defourny [Defourny06]) under the supervision of Prof. Manuel Kolp and Yves Wautelet. It is continuously subject to further corrections and developments.

The rest of this chapter is organized as follows. Section 7.1 envisages the contributions and related work. Section 7.2 presents some of the principal objectives and functionalities of the tool. Section 7.3 discusses its architecture. Section 7.4 presents the analysis capabilities of the tool while Section 7.5 presents its design perspectives. Section 7.6 presents the code generation functionalities of the tool. Finally, Section 7.7 presents the project management module and Section 7.8 concludes this chapter.

[1] DesCARTES means ***Des****ign* ***C****ASE Tool for* ***A****gent-Oriented* ***R****epositories,* ***T****echniques,* ***E****nvironments and* ***S****ystems*.

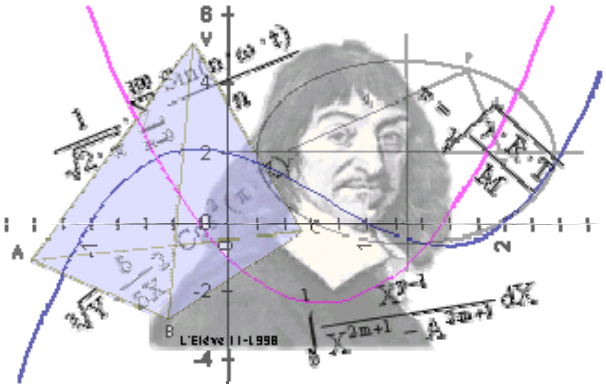


Figure 7-1. The DesCARTES-Architect Logo.

7.1. Related Work and Contributions

DesCARTES' objective is to offer a tool allowing Tropos and I-Tropos users to develop a software project easily. It is designed to support various models edition: i* models (Strategic Dependency and Strategic Rationale models), Non-Functional Requirements (NFR) models, UML models, AUML models as well as project management tools in the context of Tropos and I-Tropos developments.

Other CASE-Tools have been developed to support Tropos developments. Their functionalities vary from one another. Some just allow to edit i* diagrams, others include the edition of design diagrams, some include code generation. [i*WikiA] furnishes an exhaustive list of the main i* tools, it encompasses:

- OpenOME is designed to be a goal-oriented and/or agent-oriented modeling and analysis tool. It can be used as a standalone application and as a plug-in for other popular tools, such as Eclipse and Protégé;

- OME is graph editor to support goal-oriented and/or agent-oriented modeling;

- REDEPEND-REACT-BCN is a tool that supports i* modelling and the analysis of the resulting models. This version extends the REDEPEND i* modelling tool. The extension focus on the representation of the information system using the i* framework and provides specific functionalities for the generation and evaluation of alternative architectures for the modelled information system.

- TAOM4E supports a model-driven, agent oriented software development and, in particular, the Tropos methodology. It has been designed taking into account Model Driven Architecture (MDA) recommendations;

- GR-Tool is graphical tool in which it is possible to draw the goal models and run the algorithms and tools for forward and backward reasoning. The algorithms for the forward reasoning have been fully developed in java and are embedded in the GR-Tool.

- T-Tool provides a framework for the effective use of formal methods in the early requirements phase. The framework allows for the formal and mechanized analysis of early requirements specifications expressed in a formal modeling language;

- ST-Tool, the Secure Tropos tool, is a graphical tool where it is possible to draw Secure Tropos models and to perform the effective formal analysis of Secure Tropos specifications. The tool is written in Java with the swing components, and uses XML as its document format. Formal analysis is based on logic programming. ST-Tool allows to different systems based on Datalog to analyze Secure Tropos specification;

- J-PRiM is a tool in java that supports PRiM, a methodology that addresses i* modelling and analysis from a Process Reengineering point of view. J-PRiM allows to analyse an existing information system and to represent it as a hierarchy of i* elements. Once modelled, several alternatives for the system as-is can be explored, each of one modelled as a different i* model. All the generated alternatives can be evaluated by defining and applying metrics over the i* models in order to establish which is the most appropriate for the system to-be;

- jUCMNav is a graphical editor for ITU-T's User Requirements Notation (Z.150). URN is composed of two complementary notations: the Use Case Map (UCM) scenario notation and the Goal-oriented Requirement Language (GRL). GRL is based on the i* and NFR frameworks. jUCMNav is an Eclipse plug-in that provides editors for both notations, links between both views, analysis capabilities (including GRL model evaluations), and various import and export formats;

- SNet Tool: within this tool, the i* formalism is applied and extended in the context of a requirements engineering methodology to support inter-organizational networks. In particular, the tool provides an automatic transformation of graphical network representations (based on extended i*) into executable programs. Via this, network participants can simulate various network scenarios whose outcome may give valuable information regarding the risks and benefits of taking certain actions.

Main Purpose of the tool:	OpenOME	OME	REDEPEND-	REACT-BCN	TAOM4E	GR-Tool	T-Tool	ST-Tool	J-PRiM	jUCMNav	Snet Tool	DesCARTES	VISIO
i* modelling	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes			Yes	Yes
Analysis or Evaluation of models	Yes			Yes					Yes			Yes	
Support the TROPOS Methodology					Yes							Yes	
Forward and backward goal reasoning in Tropos						Yes						Yes	
Early requirements analysis using KAOS and Formal Tropos							Yes						
Formal analysis of functional and security requirements								Yes					
User Requirements Notation (URN) = UCM + GRL										Yes			
Simulation of networks scenarios											Yes		
Support the PRiM methodology				Yes					Yes				
NFR, UML and AUML modelling												Yes	
Support the I-Tropos methodology												Yes	
Provide forward engineering capabilities												Yes	
Provide an integrated software project module												Yes	
i* framework supported:													
Yu'95	Yes	Yes	Yes						Yes				
TROPOS					Yes	Yes	Yes	Yes				Yes	
Secure Tropos								Yes					
Iterative Tropos												Yes	
GRL										Yes			
Customizable													Yes
Availability of the tool:													
The tool is available for use	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes		Yes	Yes
The tool is available for development	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes		Yes	Yes
Programming Language													
Java	Yes	Yes			Yes			Yes	Yes	Yes	Yes	Yes	
VBA (Visual Basic for Applications)				Yes									
Datalog								Yes					
Microsoft proprietary													Yes
Platform requirements:													
Windows	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes			Yes - XP	Yes
Linux	Yes				Yes		Yes					Yes - kernel 2.6	
SunOS												Yes - 2.9	
MacOS												Available soon	
Other technology needed:													
Java Runtime Environment	Yes	Yes			Yes			Yes	Yes		Yes	Yes	
VISIO 2003				Yes									
Datalog Solver								Yes					
GraphViz for automatic layout										Yes			
OME											Yes		
Eclipse Prolog											Yes		
Indilog interpreter											Yes		
Eclipse with GEF and EMF										Yes		Yes	
MySQL server									Yes				
Current state of the tool													
There is an stable version	Yes	Yes	Yes	Yes	Yes				Yes	Yes		Yes	Yes
Ongoing work over the stable version	Yes	Yes		Yes	Yes				Yes	Yes		Yes	Yes

Table 7-1. i* tools comparison.

Table 7-1 summarizes the comparison of the available tools (from [i*WikiB]), DesCARTES is the tool offering the most exhaustive set of diagrams by including i* diagrams for requirements analysis, UML/AUML diagrams for software design as well as forward engineering capabilities, i.e., code generation. Furthermore, DesCARTES is, until now, the only i*/Tropos CASE-Tool including advanced software project management features; this probably come from the fact that no framework for this SE aspect had - until the present work - been offered for the Tropos methodology and its extensions.

7.2. CASE-Tool Objectives

DesCARTES is a CASE-Tool for Tropos/I-Tropos software development. As pointed out previously, the originality of the tool is that allows the development of the methodology analysis and design models as well as forward engineering capabilities and an integrated software project management module.

The tool requirements have been relatively easy to determine since the models used by the Tropos and I-Tropos methodologies are well known. Furthermore the experience provided by the study of the existing CASE-Tools gave a first advanced blueprint of the results to achieve. That is why the traditional waterfall Tropos methodology has been used to develop the DesCARTES CASE-Tool: i* diagrams are used for the analysis of the tool requirements while traditional UML diagrams are used for designing the tool. Indeed, to benefit of the use of an advanced graphical container and a powerful and mature API as well as to ease the trainees' work, the developments were made as a plug-in for the Eclipse platform IDE (Integrated Development Environment). The development language used is the object-oriented programming language Java.

The CASE-Tool is developed for different types of stakeholders involved in a role described in chapter 5 pursuing their own goal while using the application (similar to an activity they performed in the generic process description). Figure 7-2 represents the following dimensions:

- The *Requirements Engineer* and the *Organizational Modeler* depending on DesCARTES to *Draw an i* Diagram*;
- The *Agent Designer* depending on DesCARTES to *Draw an NFR Goal Graph* and to *Draw a UML-like Class Diagram*;
- The *Software Architect* depending on DesCARTES to *Assign an Architectural Style* and to *Assign a Design Pattern*;
- The *Developer* depending on DesCARTES to *Generate Code*;
- The *Risk Manager* depending on DesCARTES to *Define and Assign Threats*;

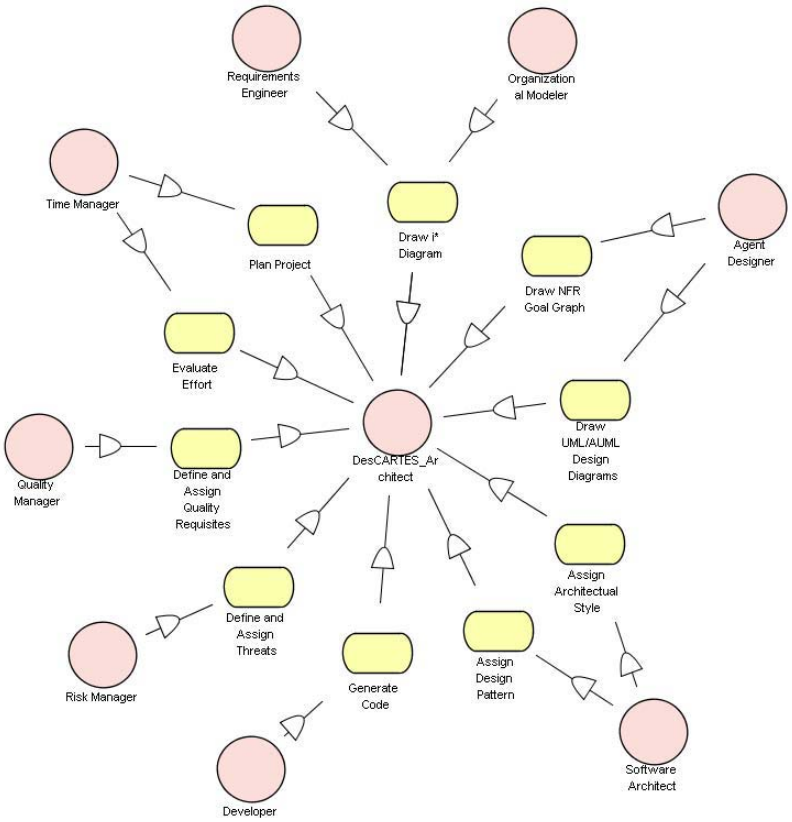


Figure 7-2. The Strategic Dependency Diagram of the DesCARTES CASE-Tool.

- The *Quality Manager* depending on DesCARTES to *Define and Assign Quality Requisites*;
- The *Time Manager* depending on DesCARTES to *Evaluate Effort* and to *Plan the Project*.

7.3. CASE-Tool Architecture

The DesCARTES architecture is developed using the *layered* [BMRSS96] and the *Model View Controller* (*MVC*) [BC87] architectural patterns. In this section, we firstly present the MVC architecture and then show how this architecture is integrated with the layered architecture to define DesCARTES.

7.3.1. The Model View Controller

The MVC architecture [BC87] proposes that external actions (from the user), data that models the external world and graphical representation are managed by three distinct object types, each of them specialized in a specific role.

Broadly speaking, the *View* manages the graphical representation of information that is modelled. The *Controller* interprets external action (e.g. user actions). The *Model* manages data, answers to data request (from the *View* in general) and perform any necessary modification (asked by the *Controller* for instance).

The MVC architecture aims to maximise the flexibility, and promotes information sharing. The *Model* centralizes information, several *Views* and several *Controllers*, can access to the *Model* and ask for modification.

DesCARTES requires several Controllers (DrawingPanel, TreeView, and PropertyPanel). These Controllers have a View and of course, they all share the same data (the Model). Every area of the screen (corresponding to a distinct Controller) has to be synchronized with the rest of the application (especially the other Controllers). Therefore sharing one centralized source of information seems the best solution.

7.3.2. Integrating MVC to the Layered Architecture

The major drawback of MVC is the communication between the three conceptual entities that need a more structured architecture to keep control of every interacting process.

In DesCARTES, a layer approach [BMRSS96] has been adopted and integrated with MVC concepts. DesCARTES proposes several layers, each of them corresponds to a MVC role. The communications are simplified since a layer communicates only with subjacent layers.

DesCARTES is organized following three major layers (Figure 7-3), eventually these layers will be composed of several packages: UI (graphical user interface, the Controller), DataView (the View) and Data (the Model). Beside the Data layer, stand the *DataHandler* and the *CodeGenerator* packages responsible for code generation and data accessing respectively.

At the top stands the UI layer which plays the Controller role in the MVC model. It also represents a part of the information present at the lower layers. Some components used by the UI layer have their own internal model that must be synchronized

with the View. This synchronization is required by the DataView layer, which is an *Observable* entity. The main part of the whole set of elements which act as Controllers are *Observers* from the DataView layer. The explanations of the *Observable* and the *Observer* entities could be found in [Java]. The UI layer is implemented using the Eclipse components. For instance, the TreeView (on the left of Figure 7-5) is an Eclipse component which holds its own an internal model providing a representation in the form of a hierarchy of the existing *DataView* elements (typically the parent child relationship in DesCARTES is root, discipline, model and element). The Eclipse Component in charge of the TreeView uses this internal model to render the Tree View area. The UI package keeps the TreeView component's internal model up to date by obtaining information from the *DataView*. This process is applied to every Eclipse Component in the UI layer.

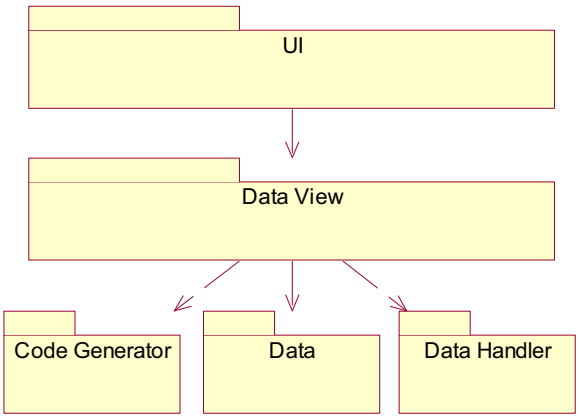


Figure 7-3. DesCARTES Layered Architecture

The DataView layer is responsible to hold information about graphical representation. It plays the View role in the MVC model but differentiates this role due to the layered architecture, every update (top-down or bottom-up) must pass through the Data View in order to propagate updates notifications (e.g., the only way to access or modify data in the Data layer is through the Data View). The DataView layer plays a more important role than the classical MVC View. It assures that update notifications are correctly propagated, from the model to the UI.

The Data layer plays the Model role in the MVC architecture. It provides the information to the upper layer (the DataView). The layered architecture simplifies communications between the three major actors of the MVC model.

7.3.3. Communications between Layers

As explained earlier, the Model updates (DesCARTES Data layer) imply some changes in the graphic representation (UI layer). Since the View is responsible to maintain this representation information, the DataView layer has to notify the UI layer about these eventual changes for updating the graphic representation. Figure 7-4 shows the communication between the three layers of DesCARTES.

When the DataView layer receives a change from the UI layer (e.g., addition of an agent, a method to an existing agent), it asks the Data layer to change its data accordingly, then the DataView sends a notification to all the registered *Observers* (situated in the UI layer). Each *Observer* has to update its data (e.g., the TreeView will update its list of agents when an agent is added to the MAS). In most cases, an *Observer* answers this notification by sending a request to the DataView asking for updated data, and then updates the (*Observer's*) representation based on the updated data returned by the DataView.

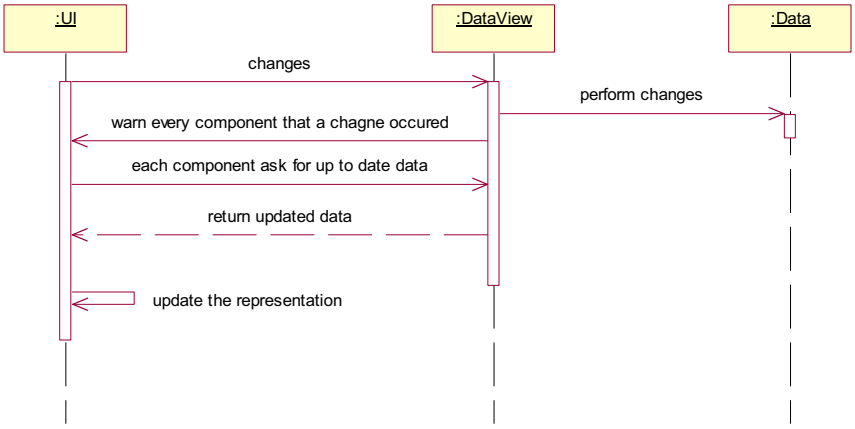


Figure 7-4. Communication between DesCARTES Layers

7.4. A Tool for Agent-Oriented Analysis

The analysis modeling capabilities of the tool help different roles involved onto the software project to:

- Edit i* diagrams;
- Edit NFR diagrams;
- Edit Use Case diagrams.

In this section, we present the layout of these editors as well as the design classes that have been required for their implementation.

7.4.1. The i* Editor

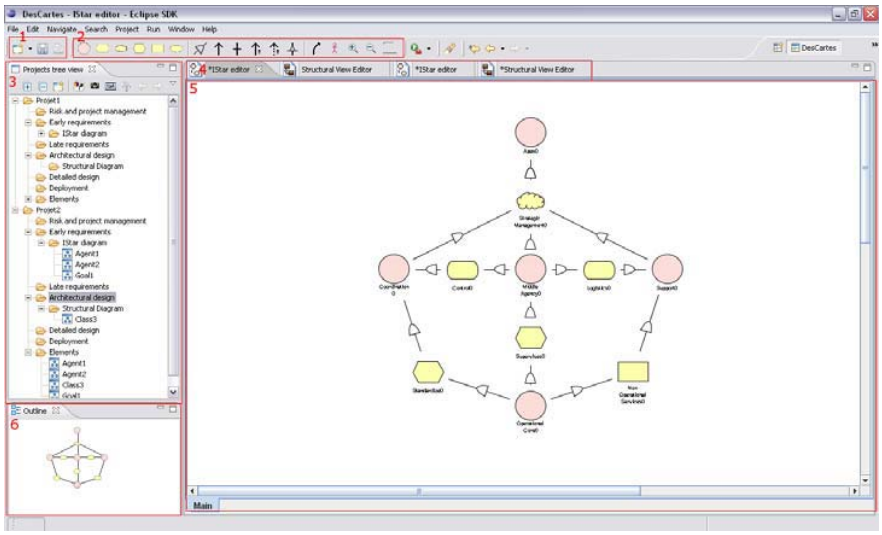


Figure 7-5. The i* Editor Layout.

Figure 7-5 represents the layout of the i* editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The i* elements icon bar (2), composed of:
    - the i* elements icons for their insertion into the drawing zone;
    - the relationship insertion icons to link the different elements;
    - icons to present the relationships as curve or straight line;
    - icons to change skins;
    - zoom icons;
    - a select all icon.

- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region. Actors rationale edition can be accessed by clicking onto the actor;
- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.4.2. The NFR Editor

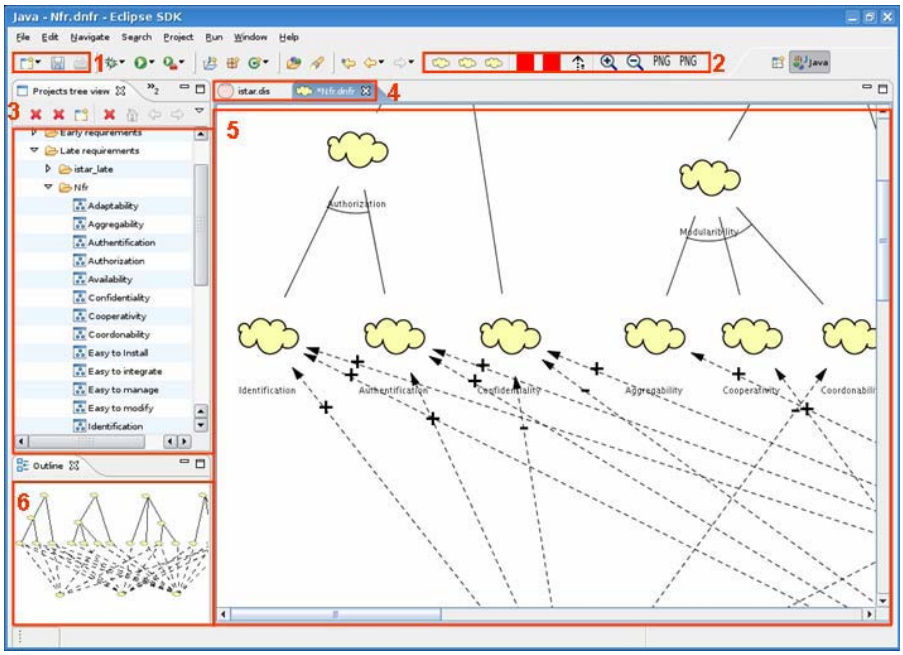


Figure 7-6. The NFR Editor Layout.

Figure 7-6 represents the layout of the NFR editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The NFR elements icon bar (2), composed of

- the NFR elements icons for their insertion into the drawing zone;
- the relationship insertion icons to link the different elements;
- icons to present the relationships as curve or straight line;
- zoom icons;
- a select all icon.

- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region;
- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.4.3. TheUse-Case Editor

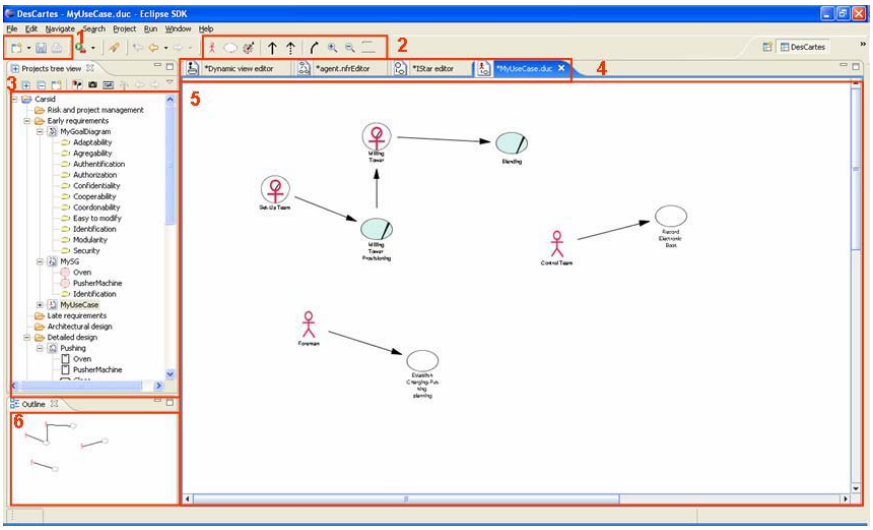


Figure 7-7. The Use-Case Editor Layout.

Figure 7-7 represents the layout of the i* editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The use-case diagram elements icon bar (2), composed of
    - the use-case diagram elements icons for their insertion into the drawing zone;
    - the relationship insertion icons to link the different elements;
    - icons to present the relationships as curve or straight line;
    - zoom icons;
    - a select all icon.
- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region;
- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.4.4. Editors Design

This section depicts the common design between the i*, NFR and Use Case editors. Those editors contain four principal parts: the drawing zone, the reduced view, the buttons area and the tree view.

Basically, an editor can contain its own sort of objects (elements) and relationships by adequately implementing their specificities. Figure 7-8 represents a simplified UML class diagram of a generic editor configuration.

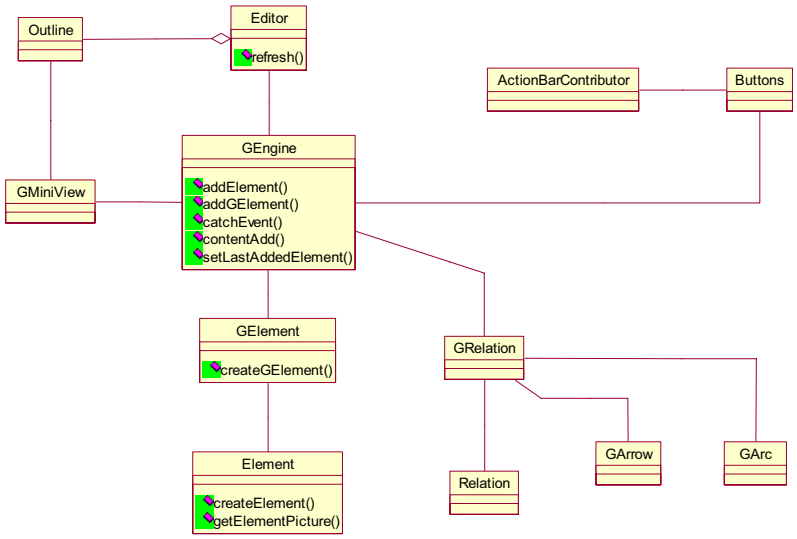


Figure 7-8. The Use-Case Editor Layout.

The generic behavior of adding an element to an editor is depicted in the UML sequence diagram of Figure 7-9. When the editor catches the event of creating a new element it calls the *catchEvent* method with the adequate elements. The graphical engine creates a new instance of the element class. The new element is returned and added to the graphical engine. The element and its pictorial representation of the element returned by the Element class are the parameters serving for the creation of the graphic element. The later is then added to the graphic engine and the editor is refreshed.

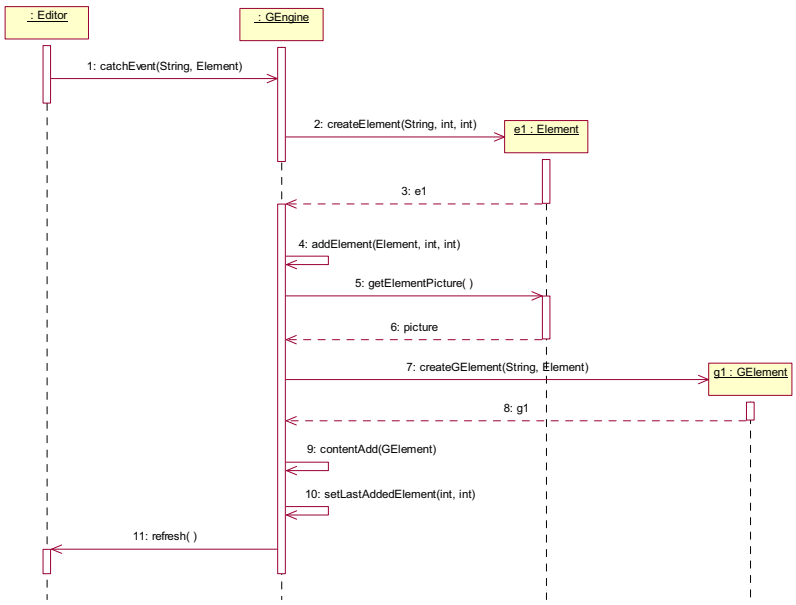


Figure 7-9. The Use-Case Editor Layout.

7.5. A Tool for Agent-Oriented Design

The design capabilities of the tool helps different roles involved onto the project to:

- Edit structural diagrams (enriched UML class diagrams);
- Edit communicational diagrams (enriched UML sequence diagrams);
- Edit dynamic diagrams (enriched UML activity diagrams).

In this section, we present the layout of these editors as well as the design classes that have been required for their implementation.

7.5.1. The Structural Diagrams Editor

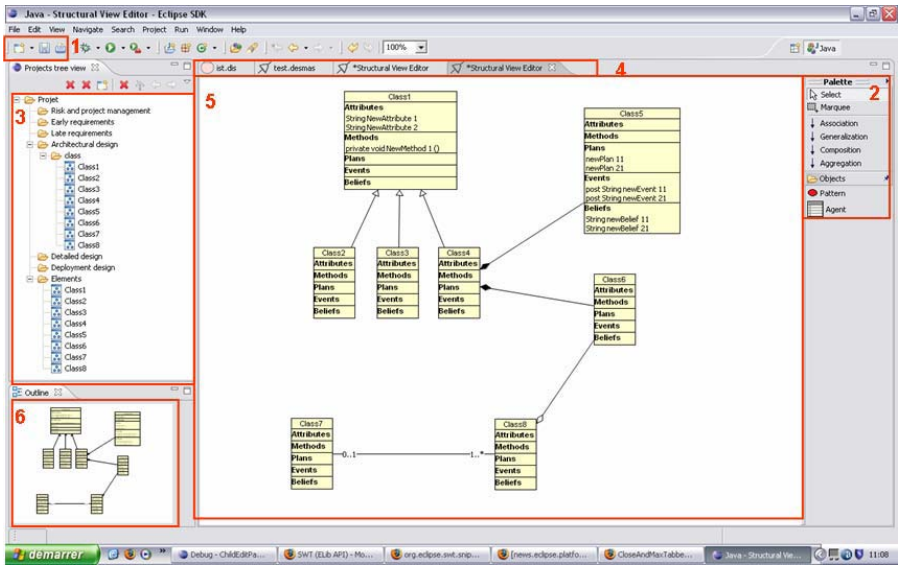


Figure 7-10. The Structural Diagram Editor Layout.

Figure 7-10 represents the layout of the structural diagrams editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The structural diagram elements icon bar (2), composed of
  - the agent element icons for insertion into the drawing zone;
  - the relationships insertion icons to link the different agents.
- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region;
- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.5.2. The Communicational Diagrams Editor

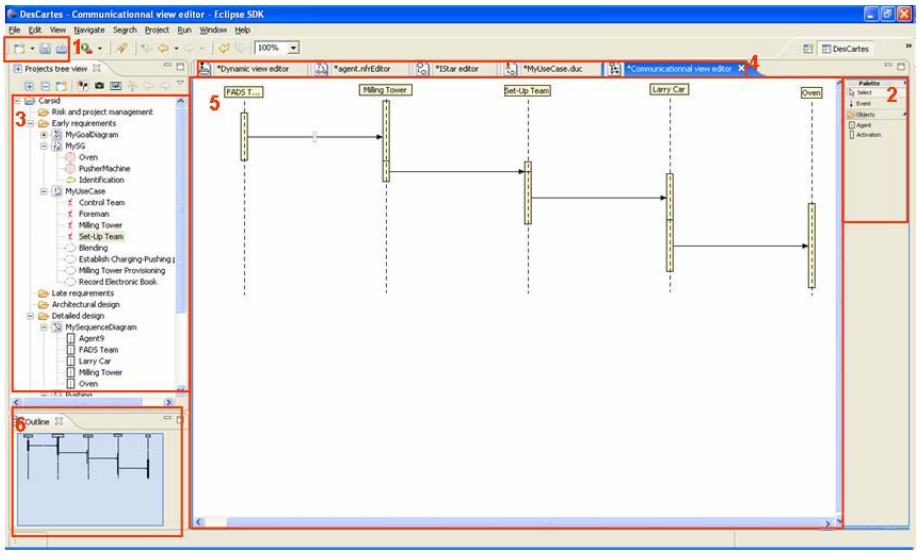


Figure 7-11. The Communicational Diagram Editor Layout.

Figure 7-11 represents the layout of the communicational diagrams editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The structural diagram elements icon bar (2), composed of
  - the agent element icons for insertion into the drawing zone;
  - the activation insertion icon to represent an algorithmic process;
  - the event icon to represent agents communication.
- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region;

- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.5.3. The Dynamic Diagrams Editor

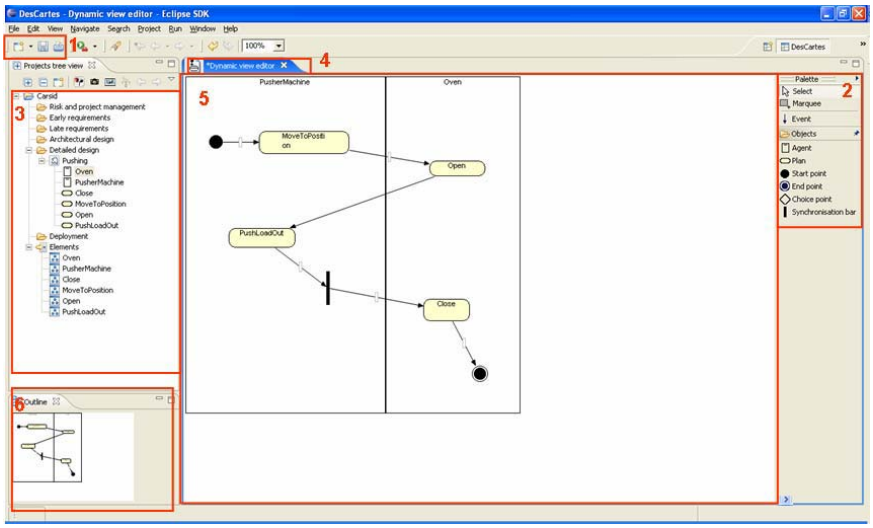


Figure 7-12. The Dynamic Diagram Editor Layout.

Figure 7-12 represents the layout of the dynamic diagrams editor included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The structural diagram elements icon bar (2), composed of
    - the agent element icon for insertion into the drawing zone;
    - the plan element icon for insertion into the drawing zone;
    - the start and end point elements icons for insertion into the drawing zone;
    - the event icon to represent agents communication;
    - the synchronization element icon for insertion into the drawing zone.

- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A drawing zone (5) with a dynamically allocable scroll region;
- A miniature view (6) enabling to get a complete vision of the edited diagram whatever the drawing zone zoom level.

7.6. A Tool for Code Generation

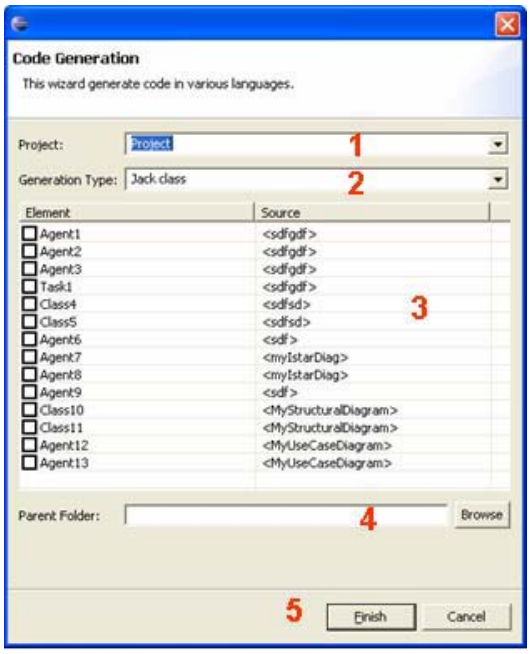


Figure 7-13. The Code Generation Layout.

Figure 7-13 represents the layout of the code generator included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The project selection combo-box (1);
- The generation type combo-box (2), generation possibilities include:
  - SQL (basis of Actors);

  - Jack Intelligent Agents (basis of Actors and Tasks);
  - Java (basis of Actors).

- The elements list (3) containing selection possibilities for the elements present in the selected project to be generated into the selected target code;
- A parent folder selection (4) allowing to select the folder in which the generated code will be put;
- The finish button (5) starting the code generation process.

7.7. A Tool for Software Project Management

The project management capabilities of the tool help different roles involved onto the project to:

- Manage Risks;
- Manage Quality;
- Manage Time.

In this section, we present the layout of these editors as well as the design classes that have been required for their implementation.

7.7.1. A Tool for Risk and Quality Management

Figure 7-14 represents the layout of the Requirements Editor included in the DesCARTES-Architect CASE-Tool.

Lines contain features that can be Use Cases in case of traditional RUP/UML development or goals in case of I-Tropos development; they are identified by a code, a number and a name. The tool enables to create a features tree, each one being able to contain several others. The columns contain the threats or quality factors. Inside the table, one can find a matrix enabling to edit dependencies existing between each feature and each threat/quality factor (dependencies are marked as green arrows). Each of the features is directly linked to a text document that can be modified. If this is the case, other documents depending on this one should be reviewed and the relationship into the matrix is marked as *suspect*. This is notified as a green arrow with a strikethrough. Moreover Use Cases, Goals, Threats and other Quality factors own attributes, Figure 7-15 represents the editor. Attributes can be added and removed directly into the tool using administration possibilities.

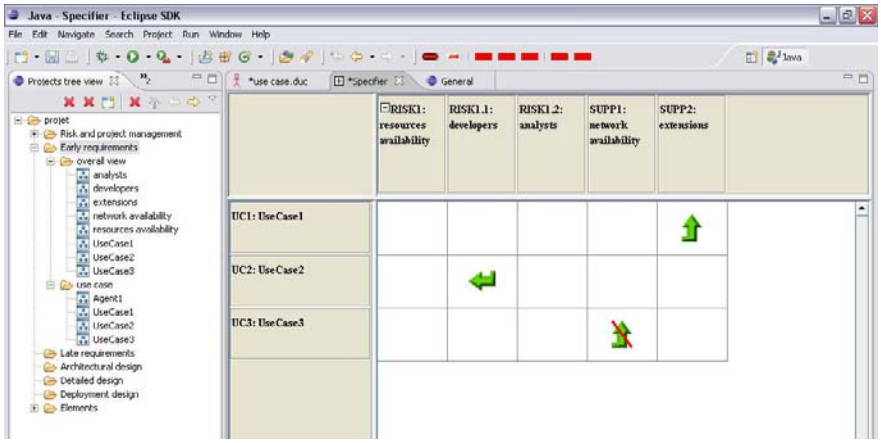


Figure 7-14. The Risk Management Tool Layout.

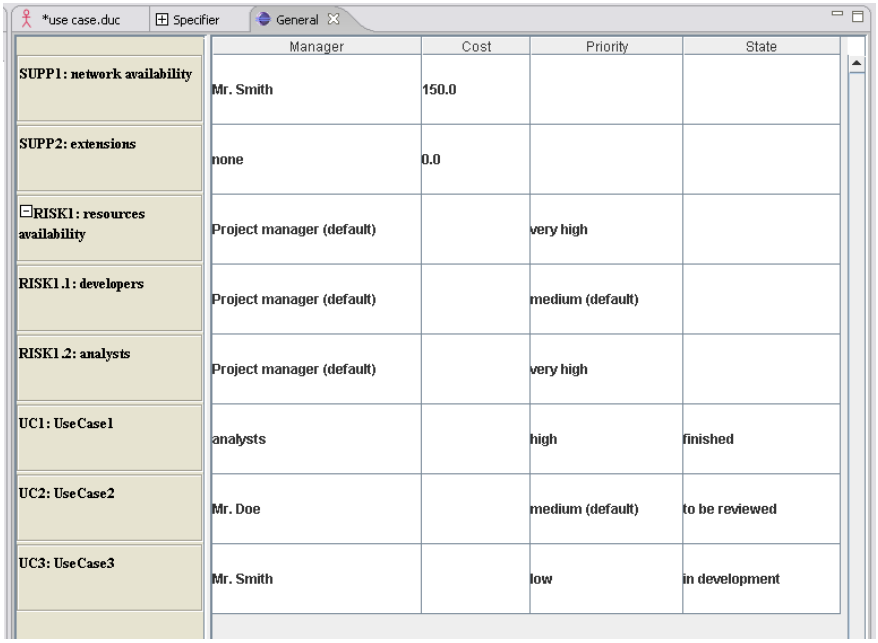


Figure 7-15. Features' attributes editor layout

7.7.2. A Tool for Time Management

Time management support provided by the DesCARTES-Architect tool encompass an effort estimation tool implementing Use Case Points, Goal Points and COCOMO II models as well as a Gantt charts Editor. The originality of the tool is that it can take an edited Use Case Model or i* Model as input and provide effort estimation on the basis of those models but also use the cost and scale drivers of the COCOMO II model. The time management module process is given in Figure 7-16. The tool output can directly be edited in the Gantt chart module to be used as a starting point for the time manager. This module is not fully functional yet and is currently under development by a master student.

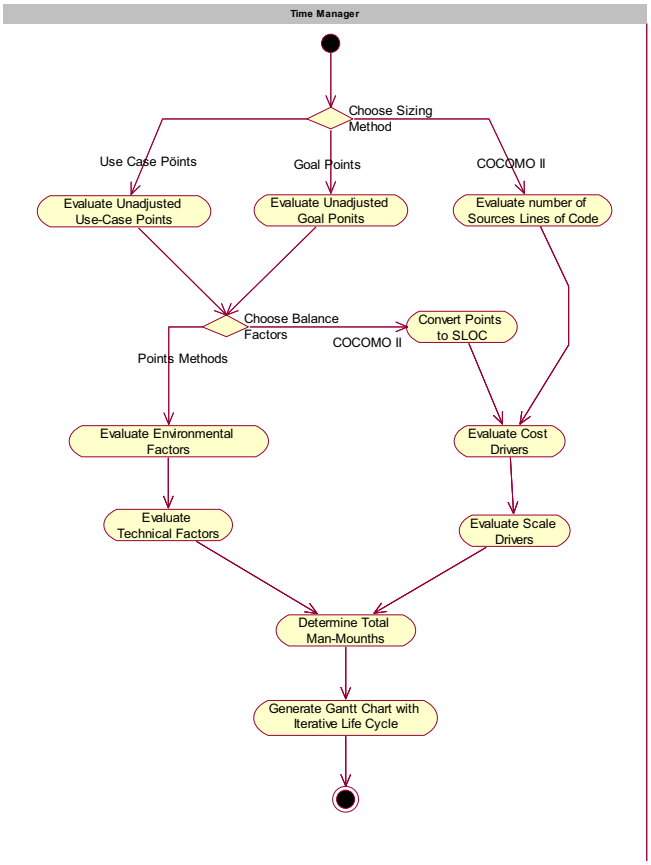


Figure 7-16. DesCARTES-Architect Time Management Workflow.

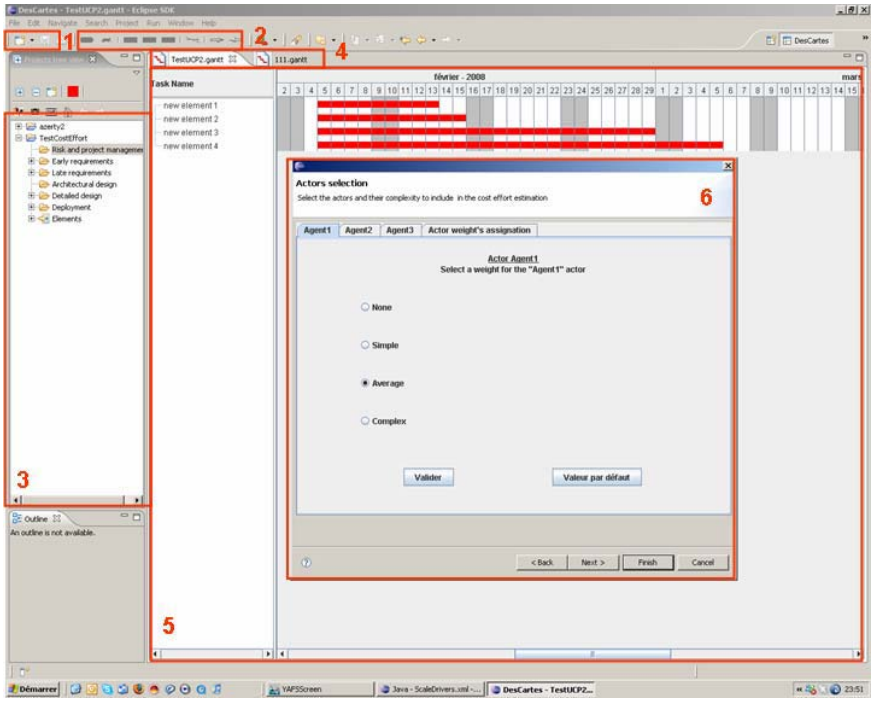


Figure 7-17. The Time Management Module Layout.

Figure 7-17 represents the layout of the time management module included in the DesCARTES-Architect CASE-Tool. Among the elements present, one can find:

- The open and save icon bar (1);
- The Gantt chart elements icon bar (2)
- An elements tree (3), i.e. each element present in the tree is logically represented in the tree. Tree elements can be drag and dropped to a drawing zone;
- A thumb-index (4) allowing to switch from an edited diagram to another;
- A editing zone (5) with a dynamically allocable scroll region;
- A selection window (6) used to select the sizing method and evaluate environmental and technical factors, cost and scale drivers.

7.8. Chapter Summary

This chapter has presented a CASE-Tool, called DesCARTES-Architect, to help the whole development process of multi-agent systems using the Tropos/I-Tropos methodologies. DesCARTES has been developed with respect to the agent-oriented concepts, it proposes tools for analysis and design diagrams edition, forward engineering capabilities as well as for software project management.

Finally, the interested reader can consult Appendix C which documents an online game for software project management learning.

Before concluding this dissertation, we point out some limitations of the tool:

- Actually, some bugs need to be corrected especially when using the tool with java 1.6;
- The project management module is still under development, the effort estimation tool will be available soon. The module needs however to be supplemented to better support the features defined in Chapter 5;
- JACK, Java and SQL code generation has been implemented. The possibility of other development languages should be added, especially Jade.

References

[BC87] K. Beck, W. Cunningham. "Using Pattern Languages for Object-Oriented Programs", workshop on the Specification and Design for Object-Oriented Programming (OOPSLA), 1987.

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. "Pattern-Oriented Software Architecture - A System of Patterns". John Wiley & Sons, 1996

[Defourny06] M. Defourny, "Développement d'un outil de gestion de projet pour la creation d'un plug-in Eclipse de conception de systèmes multi-agents". *Bachelor Thesis*, *Institut Paul Lambin*, 2006.

[Desguin05] L. Desguin. "Développement d'un outil graphique d'analyse système orienté-agent". *Bachelor Thesis*, *Institut Paul Lambin*, 2005.

[i*WikiA] i* Wiki, Available i* Tools, Available at http://istar.rwth-aachen.de/tiki-index.php?page_ref_id=21

[i*WikiB] i* Wiki, Comparing the i* Tools, Available at http://istar.rwth-aachen.de/tiki-index.php?page=Comparing+the+i%2A+Tools

[Java] http://java.sun.com/docs/

[Jenquin06] F. Jenquin, "Développement d'un outil CASE UML pour le design de systèmes multi-agents". *Bachelor Thesis*, *Ecole Supérieure d'Informatique*, 2006.

[Martel05] G. Martel. "Développement d'une application de conception orienté agent pour la méthodologie TROPOS". *Master Thesis*, *Université Catholoique de Louvain*, 2005.

[Montoisis06] M. Montoisis, "Développement d'un éditeur logiciel de modélisation organisationnelle de systèmes multi agent". *Bachelor Thesis*, *Ecole Supérieure d'Informatique*, 2006.