Software Process Modeling: Investigations Using the Rational Unified Process

Elena Roxana Tudoroiu

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montreal, Quebec, Canada

March 2005

Your file Votre référence
ISBN: 0-494-04454-3
Our file Notre référence
ISBN: 0-494-04454-3

Abstract

Software Process Modeling: Investigations using the Rational Unified Process

Elena Tudoroiu

The Rational Unified Process emphasizes the adoption of many best practices of modern software development, as a way to reduce the risk inherent in developing software. Three important characteristics such as use-case driven, architecture centric and iterative and incremental approaches make it a unique process that can guide software teams in the development of complex software systems in the fastest way.

The current approach used to model visually the Rational Unified Process is by using mostly activity diagrams, which prove to be deficient in expressing a unified view of roles, activities and artifacts in the same diagram.

We propose the use of Role Activity Diagrams (RAD), and our own extended version, XRAD, for Business Modeling, Requirements, Analysis and Design, Implementation, Test, and Deployment disciplines, as elements of originality of this thesis. We demonstrate through examples that XRAD process modeling result in superior expressive power when compared to the other diagrams that are available in the literature.

Acknowledgements

I would like to express my sincere gratitude to Dr. Paquet for his guidance, support and encouragement throughout the course of my research described in this thesis. I hope that the results obtained during this period to be a good recompense of all these conjugated efforts.

Table of Contents

List of Figures

Chapter 1 : Introduction

Process modeling is applied in a vast diversity of application areas such as business modeling, simulation, production engineering and, our focus in this dissertation, software process modeling. Software process modeling is about the unified representation of all process elements pertaining to a particular software development process, including the activities of the process (e.g. programming, unit testing, integration, risk analysis, etc.), the different roles responsible to undertake these activities, (e.g. designer, programmer, analyst, etc), and the various artifacts produced and consumed by these activities (e.g. program files, design diagrams, requirements, deployment plan, etc.).

1.1 Context and Scope

This thesis dissertation is the first one in a project at the Department of Computer Science and Software Engineering, which aims at the development of a formal context-driven software process development model, i.e. a context-driven meta-model enabling the representation of software development processes. It is important to note that this particular dissertation does not claim to express processes in a formal manner, but rather to explore the process modeling techniques used, particularly in the RUP, in order to establish their deficiencies and provide alternate solutions. This investigation will further the understanding of the team in the area of process modeling through concrete observation, enabling different critical viewpoints necessary for the development of an effective process modeling meta-model.

Our observation is based on the Rational Unified Process, and its use of Activity Diagrams to represent its process model. Note that in order to understand the process itself versus its model, we made a thorough review of the entire process, presented herein.

1.2 Thesis Statement

The process modeling techniques used to represent the RUP, mostly activity diagrams, are deficient in that they cannot show a unified view of process roles, activities, and artifacts in the same diagram. We investigate these deficiencies, and propose alternate solutions to enable a better process modeling diagramming technique enabling the unification of these three mandatory elements of all process models in the same diagram.

1.3 Contributions

Among the main contributions in this thesis, are the following:

- Development of several role activity diagrams (RAD) for the Business Modeling, Requirement, Analysis and Design, Implementation, Test and Deployment disciplines, as an element of originality. Each RAD diagram is developed in two different versions in order to improve their structures and readability.
- Development of the business modeling artifacts flow using RAD.
- Development of a new version for Business Modeling, Requirement, Analysis and Design, Implementation, Test and Deployment disciplines, called XRAD, as a combination of the second version and the artifacts, to become more attractive, expressive, and more compact than the original versions.
- Express why the new XRAD version is an element of originality.
- Development of a document legend to abbreviate the artifacts in each diagram to ensure readability.

- Explain the differences between the first and the second version for each RAD.

1.4 Structure of the Dissertation

In order to achieve our purpose, we first study existing processes used for developing software products. In Chapter 2, we will present an overview of the area of software process modeling. In Chapter 3, we will take a close look at RUP and to the modeling of roles, activities and artifacts developed in the core disciplines. In Chapter 4, we will be interested to give an alternative approach to modeling RUP, namely RAD diagrams, which prove to be superior concerning the detail and clarity they provide in modeling RUP. In Chapters 5 and 6, we will give our conclusions on the problem studied and present the future work in this field.

Chapter 2 : Background

From the software engineering perspective, the term "process modeling" is associated with the modeling of the dynamic behavior of organizations, businesses or systems. Such systems can be thought as operating or behaving as a number of interrelated processes. In order to understand the systems, we build "process models" according to particular viewpoints and using particular modeling techniques. We present in this chapter a historical overview of process modeling, some of the techniques and notations used to represent process models. Finally, we present SPEM, the Software Process Engineering Meta-model, used to develop operational process models.

2.1 History

Historically, most results of software process modeling available today have their origins in concerns for the software life cycle and the software development process raised by the emergence of the "software crisis" in the 1960s [33]. In the 1970s it was clear that the production of computer based systems, and of software specifically, presented problems that were not usually present in more familiar 'manufactured' products. As computer systems became more complex and more pervasive, this contrast became increasingly more marked. The first conference on the 'Software Process' was held in England in 1984 and has been followed by many others since [35]. Various models of the software development process have been suggested and a number of modeling techniques developed, frequently associated with some form of computer support to provide assistance for software developers in following such development processes.

Whilst work to model and support the software development process is clearly important, a major development has been to apply the emerging ideas to other processes, in particular processes underlying the operation of businesses and other organizations. Here a link has been made with techniques ([33]-[34]) used in such business contexts; in particular techniques and tools used for understanding and analyzing the business operation itself and with computer systems such as workflow, Computer Supported Cooperative Work (CSCW) and more general groupware. These links draw in concepts such as Soft Systems, Systems Dynamics, role based modeling, procedure mapping and more familiar techniques such as data flow models and activity decompositions. Computer systems are now beginning to emerge which are designed to provide support for the running of a business according to its specific operational requirements.

2.2 Process Modeling Diagrams

Many diagramming techniques, such as Petri Nets ([22]-[24]), data flows diagrams [25], can be used to represent process models. We present here a selection of these techniques.

2.2.1 Activity Diagrams (AD)

Activity diagrams represent the business and operational workflows of a system [6], [7]. An activity diagram is a dynamic diagram that shows the activity and the events that causes the object to be in a particular state.

The purpose of an activity diagram is to focus on flows driven by internal processing, as opposed to external events. They are prepared for each scenario that represents a sequence of activities following one or more events occurring in a system. The activity is triggered by one or more events and it may result in one or more events that may trigger other activity or processes.

Events represent message flows which start from start symbol and end with finish marker, having activities in between. The activity diagrams represent decisions, iterations and concurrent / random behavior of the processing.

In order to identify the behavioral elements of the activity diagrams, we use several symbols and representation rules, which we will briefly describe. The initial activity (starting point) is represented by a solid circle and the final activity by a bull's eye symbol. All the other activities are represented by a rectangle with rounded edges. The logic of the process is represented by the diamond (for the decision making) or by synchronization bars (for concurrent activities). In between the activities we use arrows which represent the events that trigger the activities.

In Figure 1, we can see an example of a complete diagram that uses the symbols and rules mentioned earlier [3].

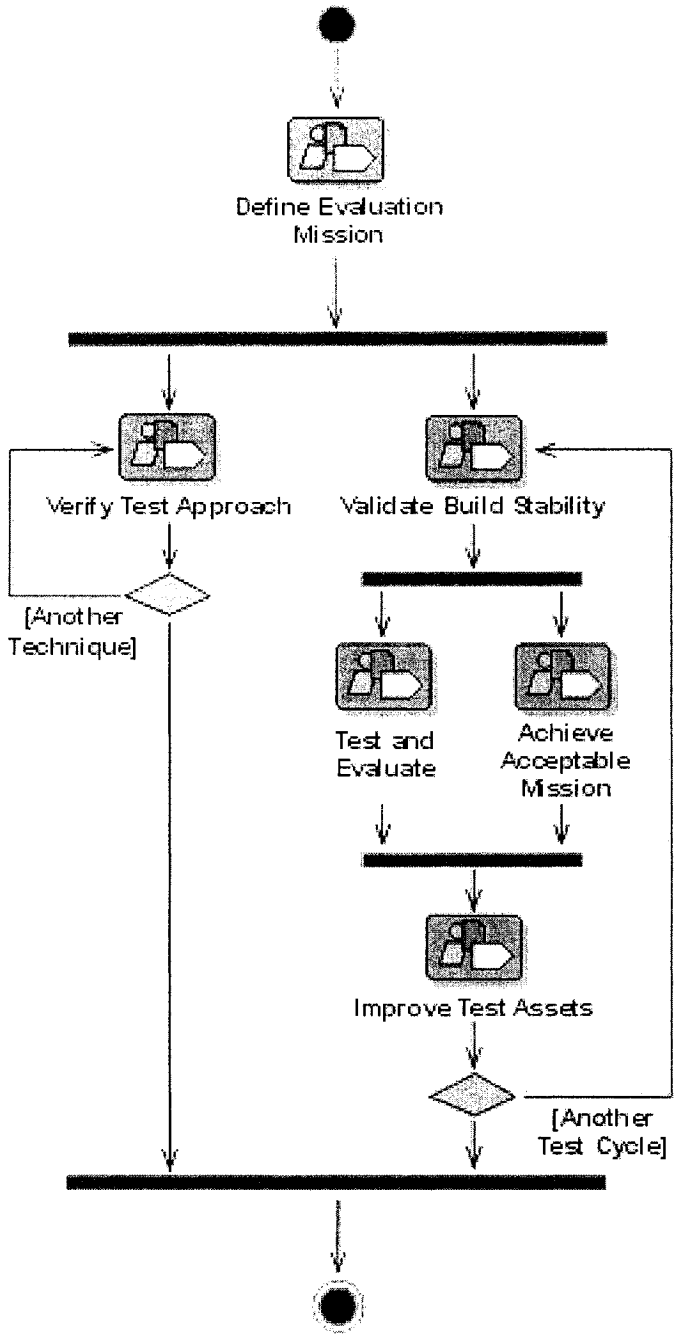


Figure 1: Activity diagram for the *Testing* discipline

Activity diagrams [6], [7] are used for documenting existing process, analyzing new process concepts and finding reengineering opportunities. They are also used to model the disciplines in RUP (see Chapter 3).

The activity diagrams focus on activities as is the case for flow charts supporting compound decisions. However, they differ from the latter ones, by supporting parallel activities and their synchronization.

The drawback of these diagrams is that they do not make explicit which objects execute which activities, and the way that the messaging works between them. However, it is possible in the activity diagrams to represent the different roles or actors that are responsible for the activities in the process. Vertical columns are made for each actor, separated by thick vertical lines, called "swim lanes", which contain the activities performed by a certain actor.

Another drawback of these diagrams is that they don't support the representation of the artifacts. In order to represent the collaboration between roles, activities and artifacts, RUP uses other kind of representation, which is not standard (see appendix A.1.5, page 114). This was the main reason of our research, to find an alternative to these diagrams (see Chapter 4, RAD and XRAD diagrams).

2.2.2 Business Process Modeling Notation (BPMN)

In this section, we will briefly refer to the Business Process Modeling Notation described in reference [10].

The Business Process Management Initiative (BPMI) has developed a standard Business Process Modeling Notation (BPMN). The BPMN 1.0 specification was released to the public in May 2004. The primary goal of the Business Process Modeling Notation effort is to provide a notation that is understandable by all business users (business analysts that create the initial drafts of the processes, technical developers responsible for implementing the technology, and the business people that manage and monitor these processes).

BPMN defines a Business Process Diagram (BPD), based on a flowcharting technique, tailored for creating graphical models of business process operations. Therefore, a Business Process Model appears as a network of graphical objects, which designate activities and the flow controls to define their order of performance.

In BPD diagrams, we distinguish four basic categories of element, namely flow objects, connecting objects, swim lanes and artifacts.

There are three types of flow objects:

- ***Events***, represented by a circle, expressing something that happens during the course of a business process. They affect the flow of the process and usually have a cause (trigger) or an impact (result). There are different types of events, based on when they affect the flow: Start, Intermediate and End.
- ***Activity***, represented by a rounded corner rectangle, designating a generic term for work. An activity can be atomic or compound. The types of activities are: Task and Sub-Process (differentiated by a small plus sign in the bottom center of the shape)
- ***Gateway***, represented by the diamond shape, used to control the divergence and convergence of sequence flow. It determines the decisions, the forking, the merging, and the joining of paths. Internal markers indicate the type of behavior control.

The flow objects are connected together in a diagram in three different ways to create the basic skeletal structure of business process:

- ***Sequence flow***, represented by a solid line with a solid arrowhead to designate the order in which the activities perform.
- ***Message flow***, represented by a dashed line, with an open arrowhead, designating the flow of messages between two separate Process

Participants (business entities and business roles), contained in separate pools.

- ***Association,*** represented by a dotted line with a line arrowhead, used to associate data, text, and Artifacts with flow objects (inputs and outputs of activities).

As in Activity diagrams, the BPMN notation supports the concept of *swimlanes* as a mechanism to organize activities into separate visual categories to represent different functional capabilities or responsibilities. There are two types of swimlane objects:

- ***Pool,*** representing a participant in a process, acting as a graphical container for partitioning a set of activities from other pools. It is used when the diagram involves two separate business entities or participants, physically separated in the diagram. The activities in separate pools are considered self-contained processes; therefore the sequence flow may not cross the boundary of a pool. It is only the message flow that allows the communication between the two participants and must connect the two pools (or the objects between the pools).
- ***Lane,*** representing a sub-partition within a pool, to organize and categorize activities. Lanes are used to separate the activities associated with a specific company function or role. The *Sequence flow* may cross the boundaries of lanes within a pool, but *Message Flow* may not be used between flow objects in lanes of the same pool.

We present here an example of a BPD that uses pools (Figure 2). It shows how the Message Flow enables the communication between the pools (this is something new with respect to the activity diagrams).

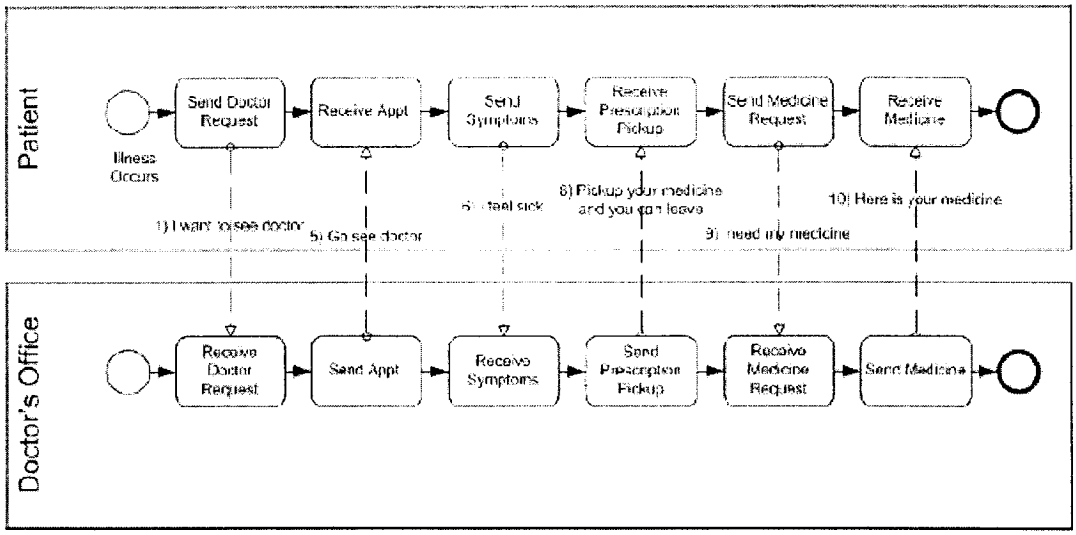


Figure 2: Example of a BPD with Pools

BPMN has the ability to add any number of artifacts to a diagram to adapt it to the context of the business processes being modeled.

The different types of artifacts are:

- ***Data objects,*** connected to activities through associations, represent a mechanism which shows how data is required or produced by activities
- ***Group,*** represented by a rounded corner rectangle, drawn with a dashed line, is used for documenting or analyzing the process.
- ***Annotation,*** used to provide additional text information for the reader of a BPMN diagram

Artifacts add more detail about how the process is performed, in order to show inputs and outputs of activities in the process. However, the basic structure of the process (determined by the Activities, Gateways and Sequence Flow) is not changed with the addition of the Artifacts in the diagram.

However, there is still more work to do in terms of standardization of the sets of Artifacts to support business Modeling and this is the future work towards which the BPMI is turned.

The diagrams we will present in the next section use many of the concepts already mentioned in the previous sections. However, they focus on the roles, the interaction between them and the activities these roles perform in a software process.

2.2.3 Role Activity Diagrams (RAD)

The basic concepts of RAD were first introduced in 1983 by Holt et al [12], and later enriched in 1995 by Ould [13]. The Ould variant of RAD is called STRIM (Systematic Technique for Role and Interaction Modeling).

STRIM is an approach to the elicitation, modeling and analysis of organizational processes. The method for modeling and analysis uses three languages with which a process is described. Two of those languages are used to capture the results of the elicitation – one that concentrates on the process – Role Activity Diagrams (RADs) and one that concentrates on the business entities involved (entity models). The STRIM analyst might also use a textual language, SPML (STRIM Process Modeling Language), to describe the organization's process in a form that has well defined semantics in order to facilitate consistency checking and enactment.

The STRIM method includes techniques for both qualitative and quantitative analysis of the process, which represents an important feature when the technique is used as part of a business process re-engineering or restructuring activity. We can find the complete description of RAD notations and symbols in Chapter 4 (section 1), together with examples from RUP disciplines flows.

2.3 Process Frameworks

2.3.1 Software Process Engineering Meta-model (SPEM)

The SPEM is used to describe a concrete software development process or a family of related software development processes [14]. It is an object-oriented approach which uses the UML notation. The Object management Group (OMG) defines four-layered architecture of process modeling as presented in Figure 3.

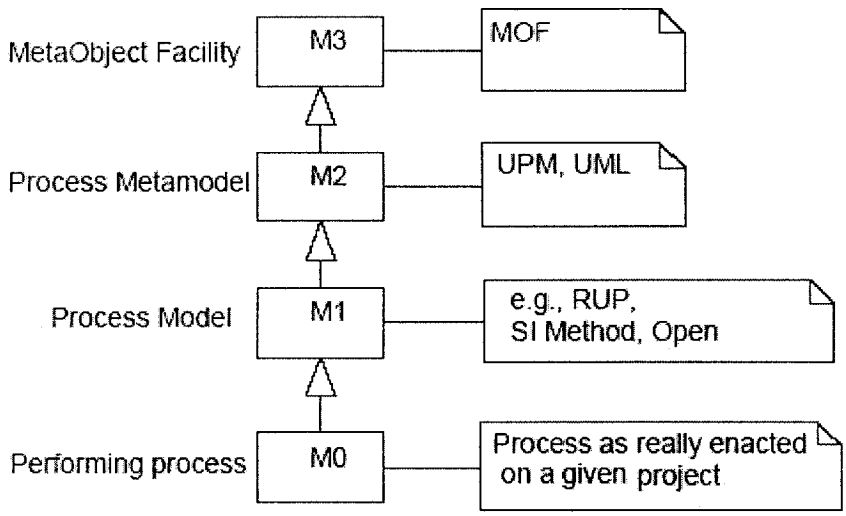


Figure 3: Levels of modeling

In this figure, level M0 represents the performing process (the real world production process), level M1 represents the process model such as RUP, level M2 represents the meta-model, such as UML, which serves as a template for level M1. The last level, M3, represents the meta-object facility.

The SPEM defines only the process modeling elements necessary to describe the software development process, without adding specific models or constraints for any specific area. It is not concerned with the actual enactment of processes (planning or executing project). The purpose of the SPEM is to support the definitions of Software Development Processes which involve UML, such as RUP. SPEM is built by extending a subset of the UML 1.4 physical meta-model. This UML subset is called SPEM foundation [14].

The conceptual model of SPEM is based on the idea that a software development process is a collaboration between active entities, called process roles, that perform operations called activities, on concrete, tangible entities, called work products, represented in Figure 4.

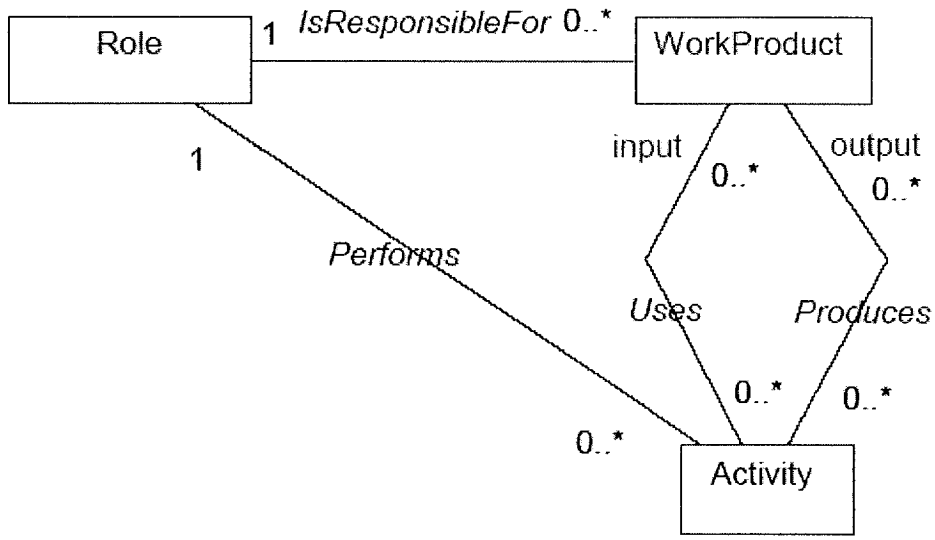


Figure 4: Reifying the Conceptual Model: Roles, Work Products, and Activities

Multiple roles interact or collaborate by exchanging work products and triggering the execution or enactment of certain activities. The overall goal of the process is to bring a set of work products to a well defined state.

Basic UML diagrams can be used to present different perspectives of a software process model. In particular, the following UML notations are used:

- Class diagrams (Figure 5) allow the representation of the following aspects of the software process: inheritance, dependencies, simple associations, comments to point to the guidance, relation between ProcessPerformer, or ProcessRole and WorkProduct, structure, decomposition and dependencies of WorkProducts. However, it is not allowed the use of the following elements: interface, template, white diamond, qualified associations, N-ary associations

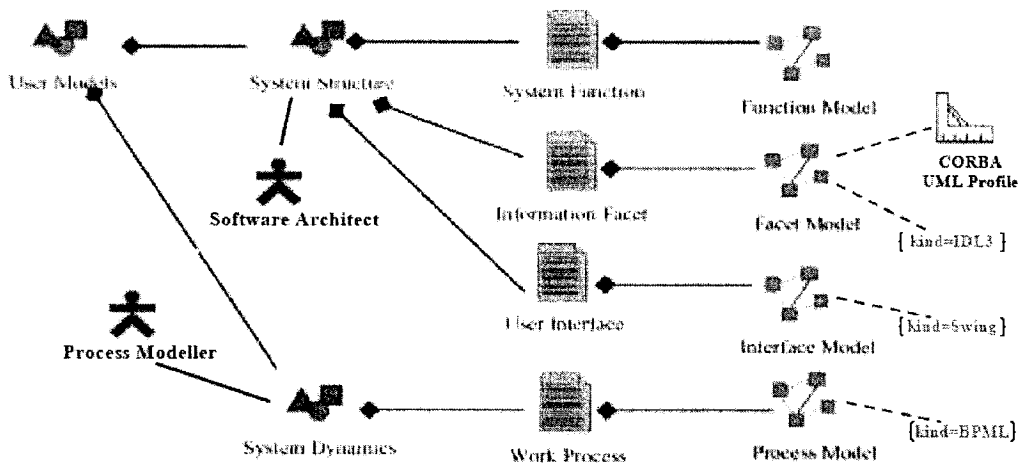


Figure 5: Example of Class Diagram in SPEM

- Package diagrams (Figure 6) allow the representation of process, process components, process packages and disciplines. They allow the use of nested and non-nested forms.

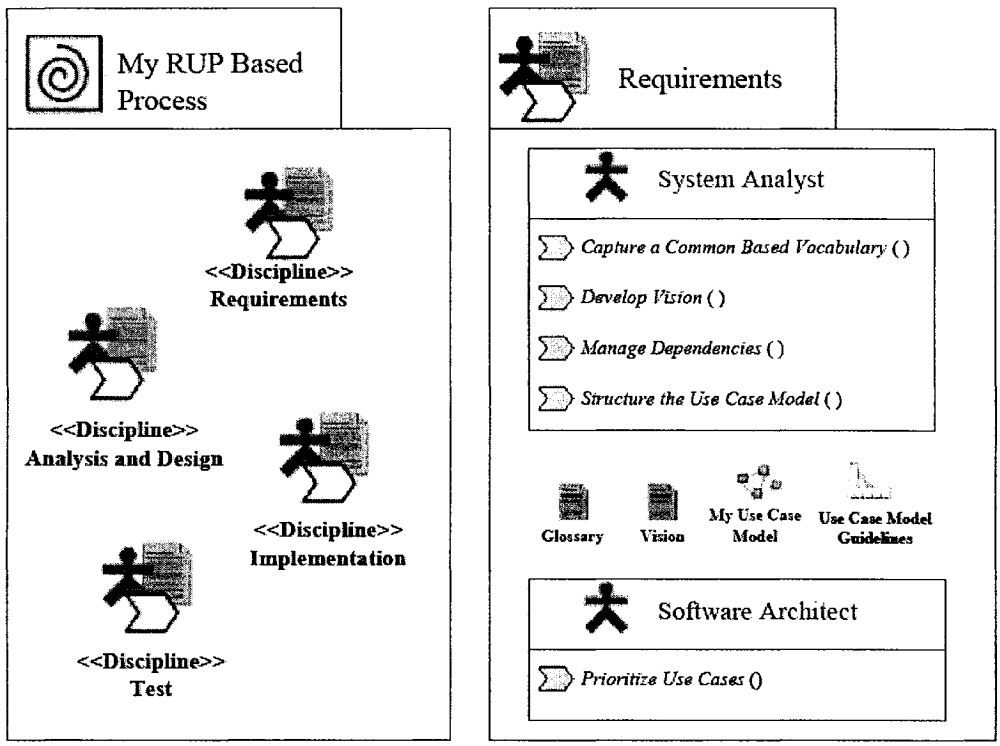


Figure 6: Example of Package Diagram in SPEM

- Activity diagrams (Figure 7) allow presenting the sequencing of the activities with their input and output work products, as well as object flow states. Swim lanes can be used to separate the responsibilities of different process roles.

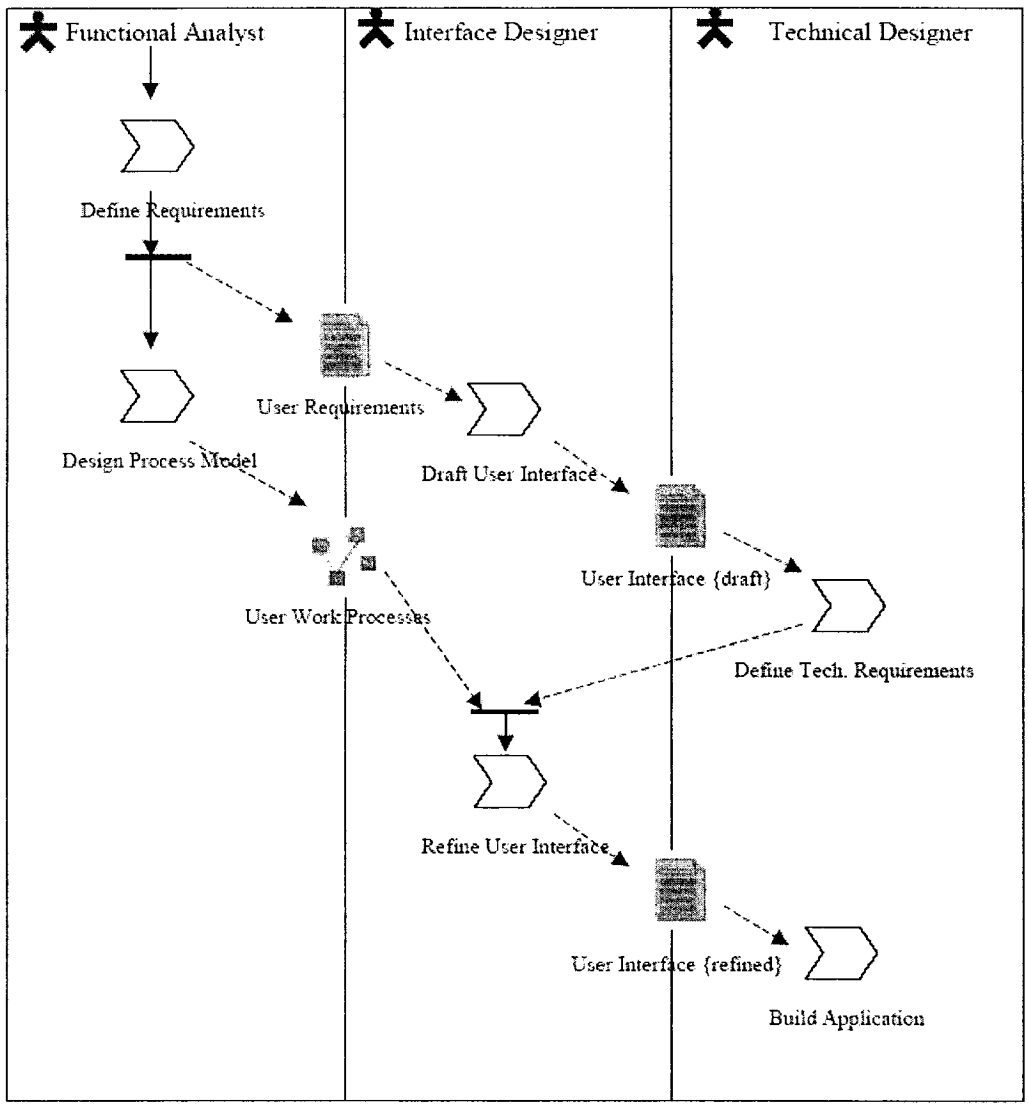


Figure 7: Example of Activity Diagram in SPEM

- Use-case diagrams (Figure 8) show the relationship between process roles and the main work definition

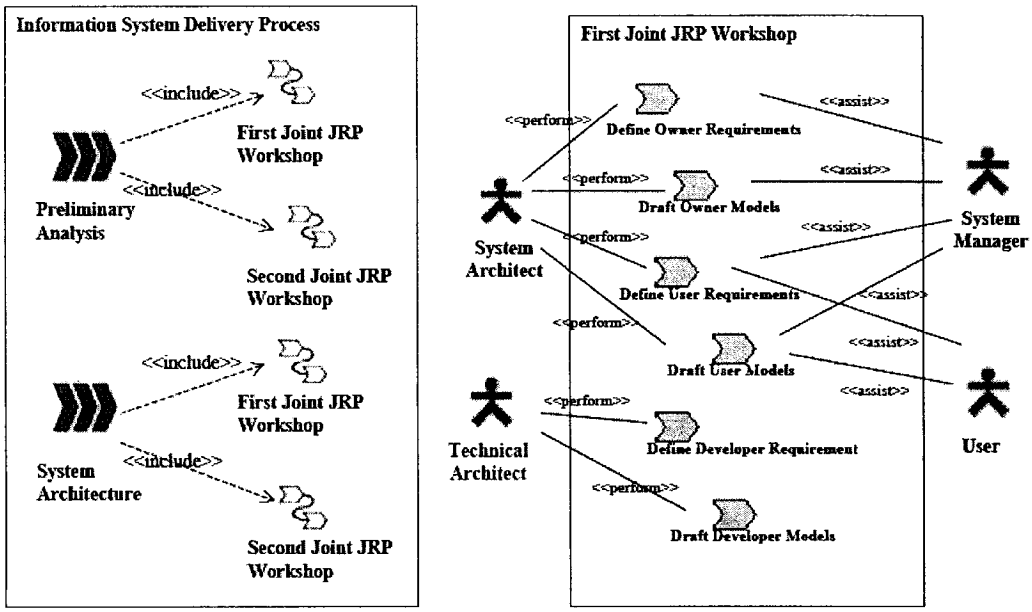


Figure 8: Example of use-case diagram in SPEM

- Sequence diagrams are used to illustrate the interaction patterns among SPEM model element instances
- Statechart diagrams are used to illustrate the behavior of SPEM elements

2.4 Summary

In this chapter, we reviewed some process modeling techniques and frameworks, namely activity diagrams, business process modeling notation (BPMN), Role Activity Diagrams (RAD) and Software Process Engineering Meta-Model (SPEM).

In the activity diagrams, we cannot explicitly show the association between the roles, artifacts, and activities. However, we can use swimlanes to separate the responsibilities of different roles who participate in the process.

BPMN defines a Business Process Diagram (BPD), based on a flowcharting technique, tailored for creating graphical models of business process operations. BPMN has the ability to add any number of artifacts to a diagram to adapt it to the context of the business processes being modeled.

RADs focus on the roles, their interactions and the activities that occur during the process. We'll see in Chapter 4 that these diagrams provide additional features to support the business modeling compared to the activity diagrams. An object-oriented approach, using UML, is developed in SPEM. The purpose of the SPEM is to support the definitions of Software Development Processes which involve UML, such as RUP.

Chapter 3 : The Rational Unified Process

3.1 Overview of the Rational Unified Process

In order to develop complex software systems and to develop them rapidly, we need a process that guides the software development teams. The unified software development process encapsulates the experience of 30 years in the field of software development [1]. It is a generic process framework that can be specialized. It includes the Objectory Process, The Rational Objectory Process and the Rational Unified Process, which is the most popular of the family of object oriented software development processes [36], [37].

The Rational Unified Process (RUP) provides a disciplined approach to software development. It is a process product, developed and maintained by Rational Software, a subsidiary of IBM Corporation. It comes with several out-of-the-box roadmaps for different types of software projects [19]. The Rational Unified Process also provides information to help the developers use other Rational tools for software development, but it does not require the Rational tools for effective application to an organization; integrations with other vendors' offerings are possible.

The Rational Unified Process is aimed at providing guidance for all aspects of a software project. It does not necessarily require you to perform any specific activity or produce any specific artifact. It does provide information and guidelines for you to decide what is applicable to your organization. It also provides some guidelines that help you tailor the process if none of the out-of-the-box roadmaps suits your project or organization. However, limited provisions are given as to

how one can ensure that a custom-tailored process is consistently or completely described [19].

The Rational Unified Process emphasizes the adoption of certain best practices of modern software development, as a way to reduce the risk inherent in developing new software [19]. These best practices are:

- Develop iteratively
- Manage requirements
- Use component-based architectures
- Model visually
- Continuously verify quality
- Identify and control risk factors
- Identify and control change

These best practices are woven into the Rational Unified Process definitions of:

- **Roles** — sets of activities performed and artifacts owned
- **Disciplines** — focus areas of software engineering effort such as Requirements, Analysis and Design, Implementation, and Test
- **Activities** — definitions of the way artifacts are produced and evaluated
- **Artifacts** — the work products used, produced or modified in the performance of activities.

RUP is an iterative process that identifies four phases of any software development project. Over time, the project goes through Inception, Elaboration, Construction, and Transition phases (see Figure 9, [19]). With the possible exceptions of the Inception phase, each phase is broken down into a number of iterations where each iteration produces an executable which constitutes a subset of the overall system. As a result, the system grows incrementally over time, iteration by iteration. During each iteration, you perform activities from several disciplines in varying levels of detail.

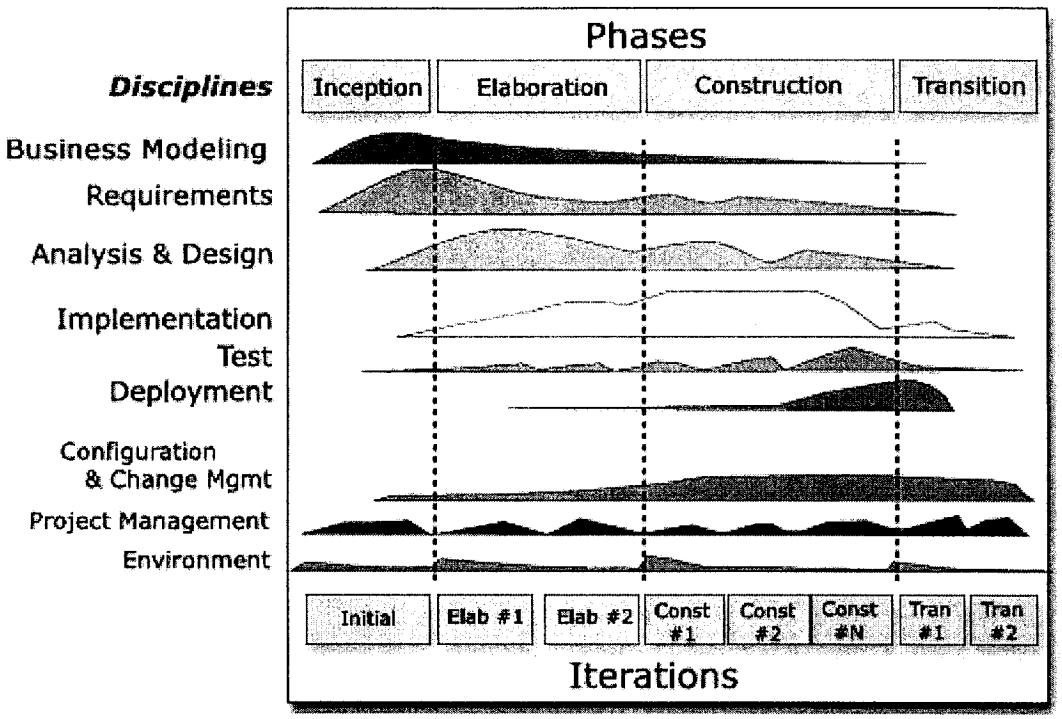


Figure 9: Overview of the phases and disciplines of the Rational Unified Process

The Rational Unified Process has three important characteristics that make it a unique process [1]:

- use-case driven
- architecture-centric
- iterative and incremental

We are presenting these characteristics in detail in order to stress their importance.

The **use case driven approach** focuses on the use case model as the basis for constructing the other successive models – the analysis, design, implementation, deployment and test models. These models are related to one another through dependencies called traces [1].

The Rational Unified Process is **architecture-driven.** It is widely recognized that architecture is an important part of the software development, because it gives us a perspective of the whole system. Architecture encapsulates the most important use cases (5-10% of the use case mass) and is developed in parallel with the use cases through several iterations [1].

The third important aspect of the RUP is **iterative and incremental** approach. The project is developed in small steps called iterations, which result in increments, which represent a growth in the product. The Rational Unified Process repeats over a series of cycles. Each cycle results in a new release of the system. Each cycle is divided into four phases: inception, elaboration, construction and transition. Each phase terminates with a *milestone*, which represents the objectives pursued in the phase.

In each phase we cover more or less all the core disciplines: business modeling, requirements, analysis and design, implementation, testing and deployment, with emphasis on only some of them. We may go several times through the disciplines in the same phase, i.e. there can be several iterations in the same phase. Having more iterations allows for a better validation and verification of the results, better distribution of feedback from the client, and a more focused risk analysis and risk reduction [1].

Such a development process gives assurance to people working on a project because it assesses the project feasibility early on and explores risks in early phases. It increases the work tempo by allowing constant feedback from customers and stakeholders.

3.2 Characteristics of the Rational Unified Process

3.2.1 Use-case driven approach

3.2.1.1 Capture the requirements

In the RUP, we capture only the requirements that add value to the user and the business [1]. In order to get the use cases that reflect this, we ask ourselves the following question: what is the system supposed to do *for each user*? [18] Developers together with the users and customers capture use cases. The developers help them express their needs concerning the system. The use cases capture the functional requirements of the system and the nonfunctional requirements (performance, availability, accuracy and security) specific to a use case.

The actors represent the different users of the system, and different users are associated with different use cases. Actors can be humans, other systems or external hardware that interacts with the system. The use cases (the functionality of the system) and the actors (the environment of the system) make up the use case model. The use case model is easy to understand by users and customers, because the use case descriptions are in English and don't use a formal language. For the developers, it is easy to divide the use cases and use them as input for the analysis, design, implementation, and testing [1], [2].

3.2.1.2 Drive the development process

Each use case in the use case model will be realized as a use case realization in the *analysis model*. The software development process is realized through several iterations (see Section 3.2.3). For each iteration, we select a set of use cases, which we realize in the analysis model. For each use case, we identify the classes that participate in the realization of the use case and we allocate the responsibilities of the use case to the class. In later iterations, we may reuse

existing classes. We also need to define the interactions between the objects of the classes in order to perform the use case realization. UML collaboration diagrams can be used to show the instances and links between them. Sometimes text can accompany the diagram to make it clearer.

The analysis model "works as a first initial cut on the design model" [1]. There is a direct mapping between the analysis model and the design model; only the latter is more detailed in order to take into account the implementation environment (object request broker, GUI, database management, legacy systems or other frameworks). The design classes are refined, and the result is that the design model will be larger and contain more classes. That's why we need to organize them, by grouping the classes in subsystems, and identifying interfaces between them. The design model works "as a blueprint for the implementation" [1]. The design classes, subsystems and interfaces are implemented as file components. The order in which components are implemented is determined by the use cases.

3.2.2 Architecture-centric approach

In the previous section, we talked about the importance of use cases in the development of software. In this section, we'll see another key element necessary to control the development of software: the architecture.

3.2.2.1 Architecture in brief

The architecture gives us a clear perspective of the whole system, by presenting different views of the models of the system. The architecture contains the most important elements, which are high-value and high-risk. The architecture is developed over several iterations in the inception and elaboration phases. The result of the elaboration phase is an executable architectural baseline which constitutes a sound architecture on which the system will be built.

The architecture also contains an architecture description that is an extract of the models of the system (use case, analysis, design, deployment and implementation models). It does not contain a view of the test model, since it is of no use to the architecture description; however it is useful to verify the architecture baseline [1], [2].

3.2.2.2 Top reasons for needing an architecture

Understanding the system. We need to make the system understandable to all people involved (developers, managers, customers and other stakeholders), in order to facilitate their participation. This is a challenge in the case of modern systems because of the complexity of their behavior, of the technology and environment in which they operate. They combine distributed computing, commercial products, reusable components, or they may be divided among geographically distributed projects, which adds to the difficulty of coordination [1].

Organizing Development. The communication overhead among developers increases with the size of the software project organization and if the project is geographically dispersed. In order to reduce this overhead, the architect divides the system into subsystems with clearly defined interfaces [1].

Fostering reuse. Developers use the components which are suitable for the problem domain and find out if they meet system requirements. Reusable components are designed and tested to fit together, so construction takes less time and costs less. A good architecture helps developers know where to look for reusable elements cost effectively [1].

Evolving the system. A system should be easy to change to fit the environment in which it operates. Developers should be able to change parts of the design and implementation, without a dramatic impact on the system. The system should be resilient to change and be capable of evolving gracefully [1].

3.2.2.3 Use cases and architecture

There is an interplay between use cases and architecture: one influences the other. We will see how this interplay occurs, by first looking at how use cases influence the architecture and then at how architecture influences the use cases. The architecture is developed in iterations in the elaboration phase. We start by deciding on a high-level design for the architecture, such as a layered architecture. Then we fashion the architecture in a couple of builds within the first iteration. In the first build, we work with the application general parts.

In the second build, we work with application specific aspect of the architecture. We select the high-risk/high-value use cases, and then we capture the requirements, we analyze, design, implement and test them. We get new subsystems implemented as components developed to support the use cases we picked. We may need to make changes to the components developed in the first build. The architecture is adapted to suit the use cases. We make another build, and so on, until we finish the iteration. If the end of the iteration corresponds to the end of the elaboration phase, the architecture is stable. After building a stable architecture, we can implement the complete functionality, by realizing the rest of the use cases during the construction phase.

The use cases developed during the construction phase use customer and user requirements as input, but are also influenced by the architecture selected in the elaboration phase. When we capture new use cases, we use our knowledge of the architecture already in place. We negotiate with the customer if we can change the use cases to make implementation simpler, by aligning the uses cases and the design with the existing architecture. By aligning the use cases with the architecture, we create new use cases, subsystems and classes cost-effectively from what already exists.

The use cases and the architecture are developed in parallel and over several iterations. We will talk more in detail in the following section about the third important aspect of RUP: iterative and incremental development.

3.2.3 Iterative and incremental approach

The strategy of the iterative and incremental process develops a software product in small, manageable steps [1], [2]:

- Plan a little
- Specify, design and implement a little
- Integrate, test and run each iteration a little

If you are happy with a step, you take the next step. Between the steps, you get the feedback that permits you to adjust your focus to the next step. The iterations in the early phases are concerned with scoping the project, removing critical risks, and baselining the architecture, whereas the iterations in the later phases result in additive increments that eventually make up the external release.

Each iteration is a "miniproject". It is not a complete project because it is not by itself what the stakeholders have asked us to do. It is a "miniwaterfall", because it proceeds through the waterfall activities [1] – planning, requirements, analysis, design, implementation, and test, which show development flowing one way. In the waterfall approach, the project would have all of its developers involved when it reached implementation, integration and testing [1].

The planners try to order the iterations to get a straight path where the early iterations provide the knowledge base for the later iterations. The ideal case is a sequence of iterations that always moves forward; that is, it never goes back two or three iterations in order to patch up the model because of something learned in a later iteration.

3.2.3.1 Top reasons for iterative and incremental development

Iterative and incremental development creates better software. We will see here some of the reasons we use this approach.

Mitigate Risks. "Risk is inherent in the commitment of present resources to future expectations [15]." In the RUP, we identify risks early in the development (inception and elaboration phases). As a result, unidentified risks don't show up later and endanger the project. The difference between the waterfall model and the iterative approach lies in the fact that the first one identifies the problems (risks) late in the project, when it is difficult and expensive to make changes. The waterfall model [1] goes through the disciplines just once. The implementation, integration and testing will show big problems which will be hard to manage. In the iterative approach, when development time reaches the construction phase, no serious risks remain to deal with.

Get a Robust Architecture. We already saw in the previous section how the architecture is developed, through iterations in the inception and elaboration phases. In the inception phase, we seek a core architecture that satisfies the key requirements, overcomes critical risks, and resolves the central development problems. In the elaboration phase, we establish the architecture baseline that guides further development. The investment in these phases is still small, and we can afford the iterations that assure the architecture is robust.

Handle Changing Requirements. From the point of view of the Stakeholders, it is better to evolve the product through a series of executable releases (or builds) than on documentation. A build is an operational version of the part of a system that demonstrates a subset of the system capabilities [1]. It allows Stakeholder feedback in the early phases. The plan – budget and schedule – is not yet set in stone so the developers can easily accommodate revisions. In the waterfall model [1], users see an operational system late, when changes will add to the budget and schedule. Therefore, the iterative approach helps customers see the

need for changes early, which will enable the developers to implement them right away.

Achieve Continuous Integration. In the iterative approach, we achieve continuous integration, by providing regularly, at the end of each iteration, a build, which is a working part of the system. In this way, the Stakeholders can also follow the progress of the project.

Attain Early Learning. The iterations helps the team understand better how the process works. After several iterations they also get better acquainted with the new technologies and tools. The team fine-tunes the process and tools before other developers join them.

3.2.3.2 Risk-driven approach

"If you do not actively attack the risks in your project, they will actively attack you [16]." This is what the experience doing the waterfall model proves. The waterfall model masks the real risks to a project until it is too late. The mistakes from earlier phases will be discovered late, when it's costly to undo them; they could even create project cancellation. The iterative and incremental approach is a better alternative to software development since it enables the identification of risks early in the lifecycle, when it's possible to attack them in a timely and efficient manner [2]. This approach is building on the work of Barry Boehm's spiral model [17].

The iterations are carried out on the basis of risks and their order of importance. Risks are classified in many categories [18]. We can distinguish the risks related to new technologies, risks related to architecture, risks related to building the right system, one that supports the mission and the users, and risks related to performance. In the elaboration phase, we establish a robust architecture which accepts changes gracefully, so it eliminates the risk of having to redesign everything [1].

There are four ways of dealing with risks: avoid it, confine it, mitigate it or monitor it. Avoiding or confining a risk takes re-planning or rework. Mitigating a risk might require the team to build something that exposes the risk. Monitoring a risk involves choosing a monitoring mechanism, setting it up and executing it. Mitigating or monitoring risks takes time, that's why a project organization can not address all risks at the same time. From here comes the idea of prioritizing the iterations.

3.3 The RUP Phases

A RUP project is extended over four phases: inception, elaboration, construction and transition. Each phase ends with a major milestone, at which managers make crucial decisions, decide on schedule, budget, and go, no-go requirements. A major milestone is a set of objectives that need to be met in order to be able to go to the next phase. In order to reach these objectives, the process moves through a series of iterations and increments. The iterations follow the core disciplines, with emphasis on only some of them. Some activities are common for all phases, like planning an iteration, setting the evaluation criteria, establishing a risk list, prioritizing the use cases and assessing the iterations.

3.3.1 Inception

In the inception phase, activity is concentrated in the business modeling and the requirements disciplines, with a little work carrying over to the analysis and design discipline. This phase seldom carries work as far as the final two disciplines, implementation and test. The goal of the inception phase is to establish the business case, identify and reduce the critical risks, move from a key subset of the requirements through use-case modeling into a candidate architecture, make an initial estimate of cost, effort and schedule.

3.3.2 Elaboration

In the elaboration phase, the activity is focused on the two disciplines: requirements and analysis and design, as they underlie the creation of the architectural core. In order to reach an executable architectural baseline, there is necessarily some activity in the final disciplines, implementation and test. The primary product of the elaboration phase is a stable architecture, to guide the system through its future life.

3.3.3 Construction

In the construction phase, the requirements discipline diminishes, analysis and design lightens, and the work focuses on the last three disciplines (Implementation, Test and Deployment). The major milestone is initial operational capability. Other activities also include maintaining the integrity of the architecture. This phase employs more staff over a longer period of time than any of the other phases. It is also generally carried out through a greater number of iterations than the other phases.

3.3.4 Transition

In the transition phase, the percentage of work in the disciplines depends on the feedback from acceptance or beta test. If the beta tests uncover defects in the implementation, there will be considerable activity in implementation and test disciplines. Main activities include: site preparation, preparation of manuals and other documentation for product release, correction of defects. The transition phase ends with a formal product release. However, before the team is done with the project, the team leaders meet to discuss and record for future reference "lessons learned", which can be of use for the next release.

3.4 The RUP Core Process Disciplines

RUP has an overall of nine disciplines: six of them are the *core disciplines* and the other three are the *supporting disciplines*. In each discipline there are workers that collaborate together and perform activities, whose result are *artifacts*. We will present briefly in this section the goals of the core disciplines as well as the existing way of representing the disciplines and the collaborations. The disciplines are represented using activity diagrams, and the collaborations are represented using other types of diagrams. For a complete description of the workers involved, artifacts produced and activities performed in each of the core disciplines, please refer to the annexes A1 to A6.

3.4.1 Business Modeling Discipline

The main goal of business modeling is to understand the dynamics and structure of an organization, the possible problems that exist and possibly find a solution. In order to achieve this goal, Stakeholders and customers work very closely with the software developers (see workflow details in appendix A.1.5, page 114) to derive the system requirements needed to support the target organization. The *Business Modeling Discipline* (see Figure 10) is represented by an activity diagram. It shows that first we need to assess the status of the organization in which the eventual system is to be deployed (the target organization), as defined in *Assess Business Status*.

Based on the results of this assessment, we can choose which path from the discipline we need to follow. If we determine that no business models are necessary, we develop a domain model. In the case of business modeling, we have several cases possible: if we determine that no changes need to be made to the current process, we don't need to describe the current process, but focus on the target organization; if we want to improve or re-engineer an existing process, we need to model both the old and the new business; if we want to model a new business from scratch, we again don't need to *Describe Current*

Business [3]. When we explode the workflow details, we get several diagrams which show the workers, the activities they perform and the artifacts that they produce together (see Figure 61 to Figure 68 in appendix A.1.5, page 114).

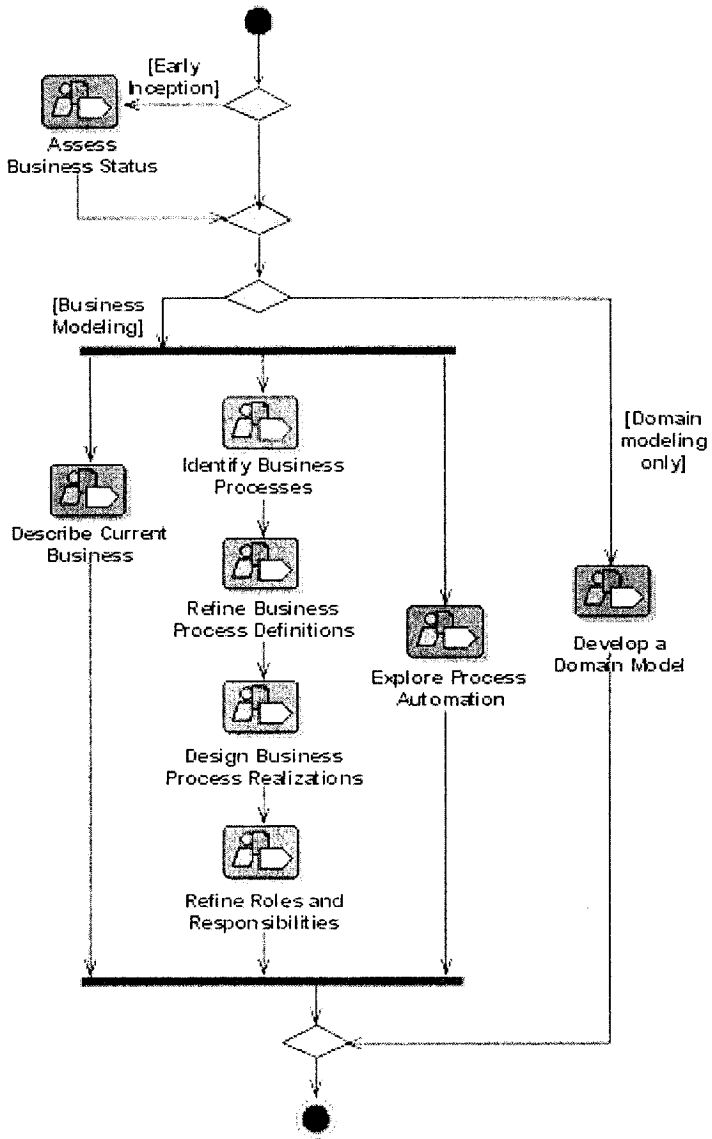


Figure 10: Activity diagram for the *Business Modeling* discipline

3.4.2 Requirements Discipline

In the *Requirements* discipline, developers and stakeholders are working together in order to understand better the system requirements. The stakeholder is involved in almost all the collaborations in the discipline (see the workflow details in appendix A.2.5, on page 130). The goals of the developers in the requirements discipline are: to define the boundaries of the system, to provide a basis for planning the technical contents of iterations, to provide a basis for estimating cost and time to develop the system and to define a user-interface for the system, focusing on the needs and goals of the users.

In order to achieve these goals, developers need to perform effective requirements management, at every step in the requirements discipline (see Figure 11). During the Inception phase of a project, they *Analyze the Problem* and *Understand Stakeholder Needs*, during the Elaboration phase, they *Define the System* and *Refine the System Definition*, and they *Manage the Scope of the System* and *Manage Changing Requirements* , continuously throughout the project [3].

As for the Business Modeling, we will present the activity diagram that shows the main workflow of the Requirements discipline (the secondary diagrams that show the workflow details, as collaborations between workers, activities and artifacts, are presented in appendix A.2.5, page 130).

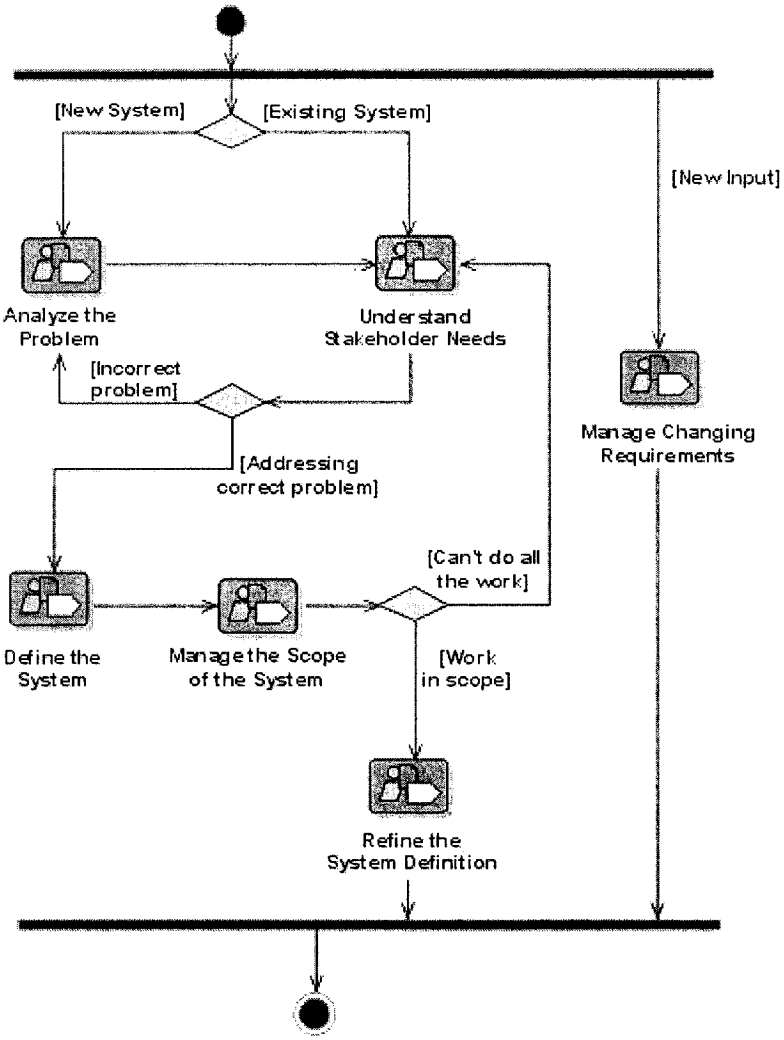


Figure 11: Activity diagram of the *Requirements* discipline

3.4.3 Analysis and Design Discipline

The purposes of analysis and design discipline are: to translate the requirements into a specification that describes how to implement the system, to establish a robust architecture early in the project, in order to design the system, and finally to adjust the design to match the implementation environment, designing it for performance, robustness, scalability and testability.

In order to achieve these goals, we follow the steps presented in the activity diagram of the *Analysis and Design* discipline (see Figure 12). Some of these steps are phase-dependant. For example, the workflow detail *Perform Architectural Synthesis* is done in the *Inception* phase. It is concerned with establishing whether the system as envisioned is feasible, and with assessing potential technologies for the solution. The workflow detail *Define a Candidate Architecture* is done early in the *Elaboration* phase. It focuses on creating an initial architecture for the system. If the architecture already exists, the work focuses on refining the architecture and analyzing behavior.

The workflow detail *Design Components* produces a set of components which provide appropriate behavior to satisfy the requirements on the system. Parallel to this activity, persistence issues are handled in the workflow detail *Design the Database* [2], [3]. We will present the activity diagram of the Analysis and Design discipline (the secondary diagrams that represent the workflow details are presented in appendix A.3.5, page 147).

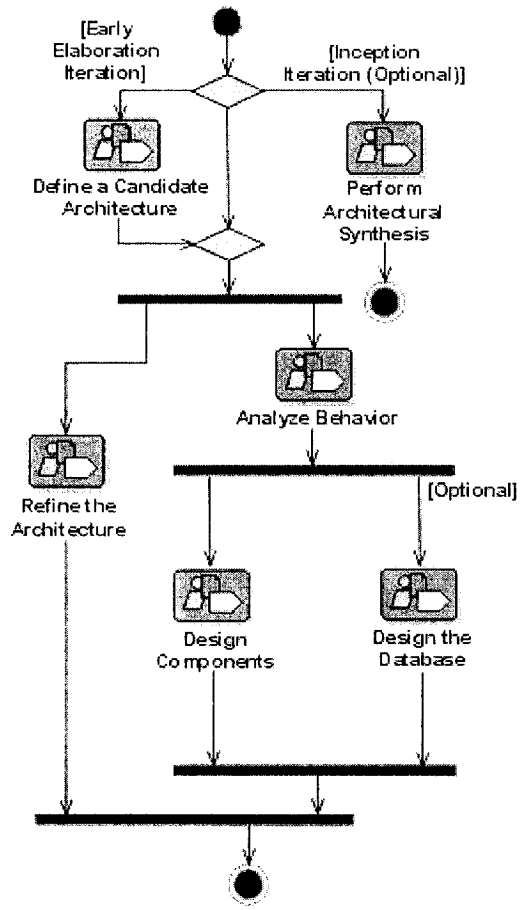


Figure 12: Activity diagram of the *Analysis and Design* discipline

3.4.4 Implementation Discipline

The four purposes of the *implementation* discipline are: to define the organization of the code in terms of the implementation subsystems, to implement classes and objects in terms of components (source files, binaries, executables and others), to test the developed components as units and to integrate into an executable system the results produced by individual implementers or teams. The scope of test within the implementation discipline is limited to unit test of individual components (which is the responsibility of the implementer). System test and integration test are described in the test discipline (see next section).

In order to achieve the purposes previously mentioned, we follow the activities in the implementation discipline (see Figure 13). The main work to *Structure the implementation model* is done in the elaboration phase. Its purpose is to ensure the implementation model is well-organized and prevents configuration management problems and allows the product to be built up successively from integration builds. In *Plan the Integration*, the work focuses on planning which subsystems should be implemented and the order in which subsystems should be integrated in the current iteration.

The implementers then implement the classes and objects in the design model during the activity labeled *Implement Components*. During this activity they also fix code defects and perform unit tests to verify changes. Following this activity, the components are integrated into a subsystem (*Integrate each Subsystem*), the result of which is a build. The build is integration tested before it is handed to the System Integrator to integrate it to the system, in *Integrate the system* [2], [3]. We will present the activity diagram of the Implementation discipline (see appendix A.4.5, page 158, for the complementary diagrams that show the workflow details).

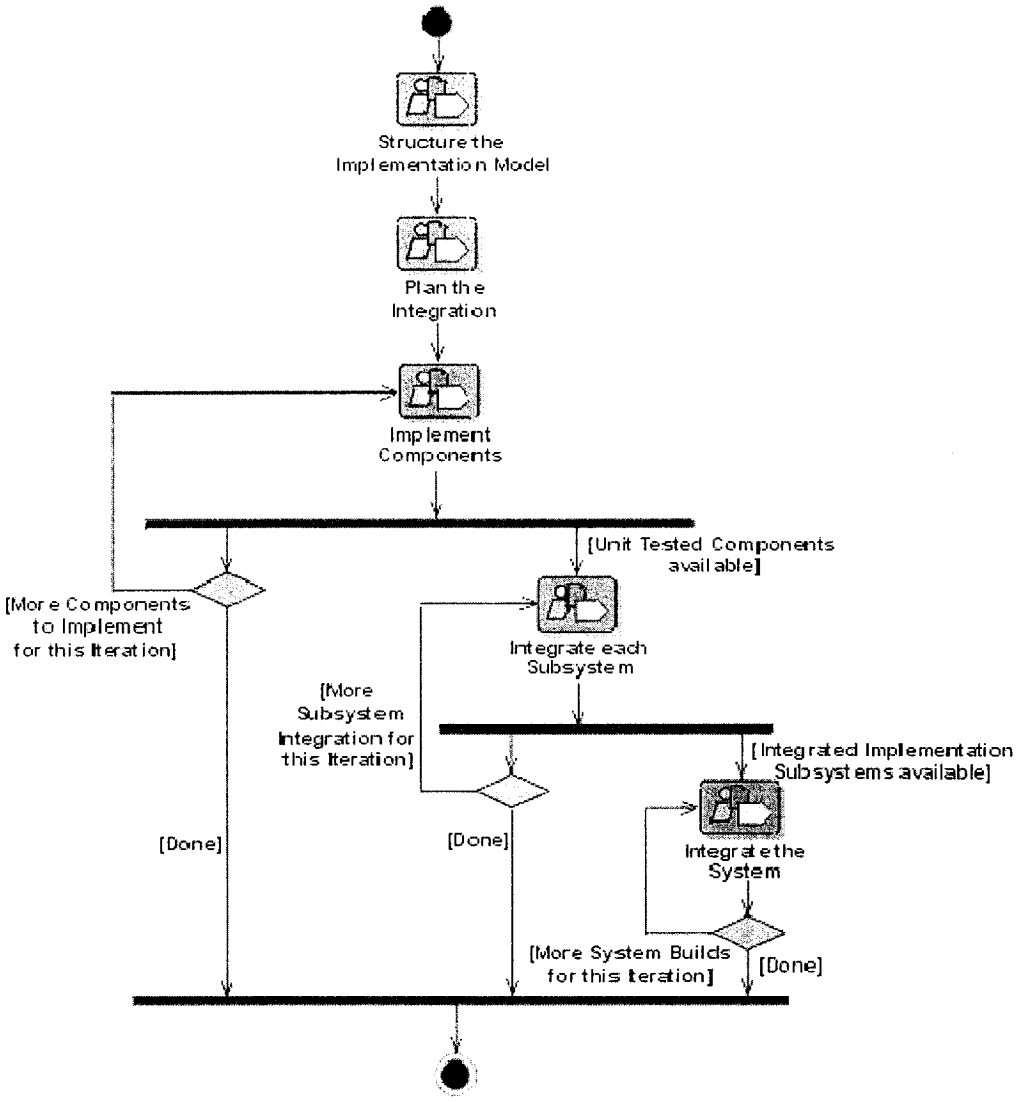


Figure 13: Activity diagram of the *Implementation* discipline

3.4.5 Test Discipline

Testing focuses on evaluating or assessing product quality, using the following practices [2]:

- Find and document failures in the software product: defects, problems
- Advise management on the perceived quality of the software
- Evaluate the assumptions made in design and requirement specifications through concrete demonstration
- Validate that the software works as designed
- Validate that the requirements are implemented appropriately

Test activities are normally divided into *verification* and *validation*. Verification is the work involved in checking whether the result agrees with the specification, whereas validation is the work necessary to check whether the end result is what was actually wanted [26]. In [27], validation is concerned with building the right thing and verification with building the thing right.

These practices are achieved by the intermediate of the activities in the discipline (see Figure 14). In *Define Evaluation Mission* we identify the appropriate focus of the test effort and gain agreement with the Stakeholders on the corresponding goals that direct the test effort. In *Verify Test Approach*, we demonstrate that the various techniques outlined in the Test Strategy will facilitate the planned test effort. Parallel to this activity we *Validate Build Stability*, before entering a test cycle for a new build. We validate that this build is stable enough for detailed test and evaluation effort to begin.

We then do the activity *Test and Evaluate*, which involves implementing, executing and evaluating specific tests and the corresponding reporting of incidents that are encountered. Parallel to this activity, we deliver a useful evaluation result to the Stakeholders in *Achieve Acceptable Mission*. And finally, we *Improve Test Assets*, such as Test Ideas List, Test Cases, Test Data, Test

Scripts [3]. We will present the activity diagram of the Test discipline (see appendix A.5.5, page 170, for the complementary diagrams that represent the workflow details).

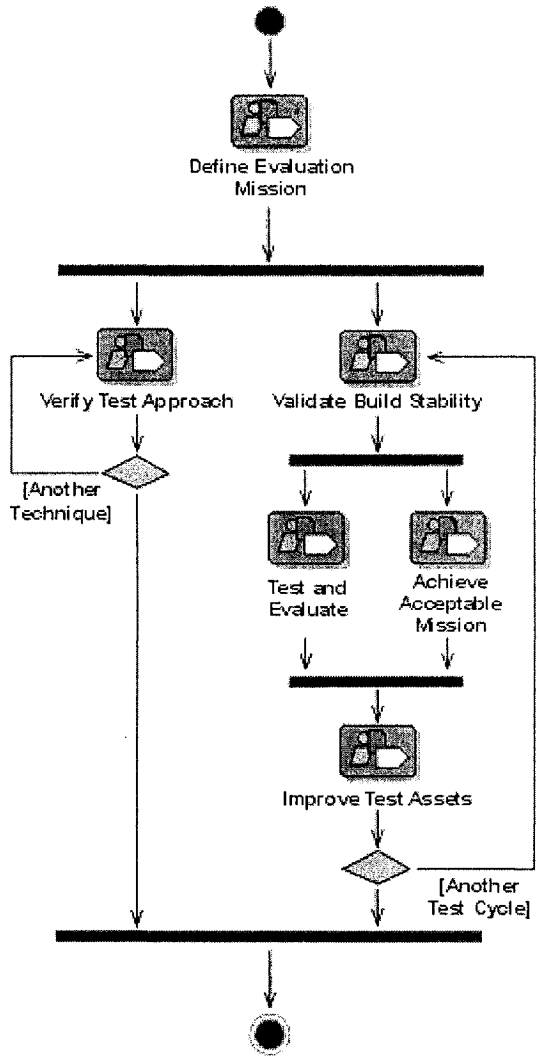


Figure 14: Activity diagram of the *Test* discipline

3.4.6 Deployment Discipline

The purpose of the *Deployment* discipline is to deliver the finished product over to its users. This discipline involves the activities of: *testing* the software in its final operational environment (*beta test*), *packaging* the software for delivery, *distributing* the software, *installing* the software, *training* the end users and the sales force and migrating existing software or converting databases [2]. We will present (see Figure 15) the activity diagram of the Deployment discipline (for the complementary diagrams of the Deployment details, see appendix A.6.5, page 182).

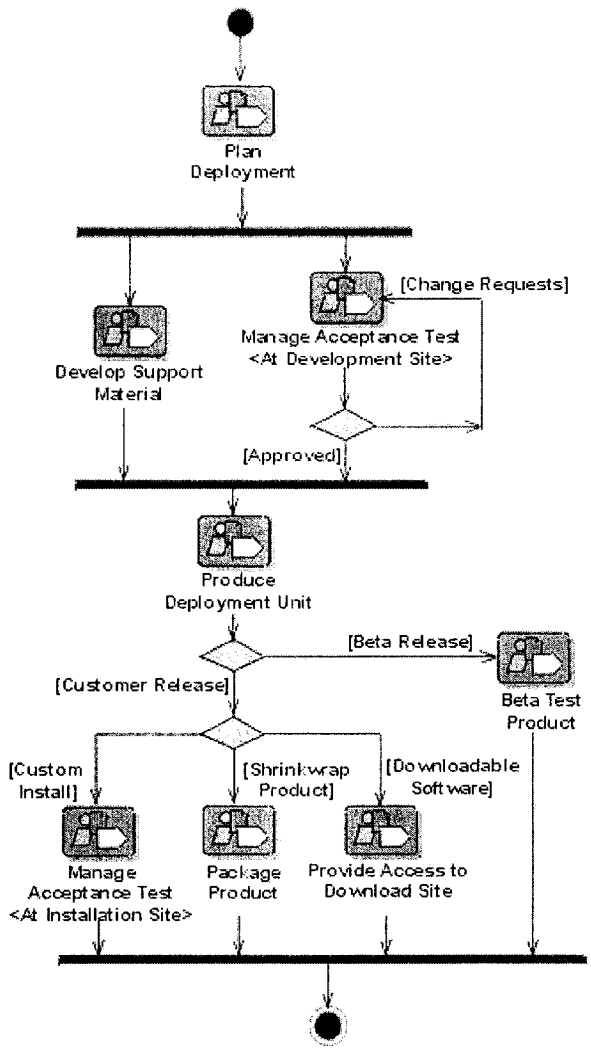


Figure 15: Activity diagram of the *Deployment* discipline

3.4.7 Limitations with the existing model

The existing representation of the RUP, as we saw in the previous sections, doesn't allow the representation of the roles, artifacts and activities on the same diagram. We use activity diagrams to represent the main workflow, and complementary diagrams to represent the collaborations between workers, activities and artifacts. The complementary diagrams are not activity diagrams. They are a non-standard way of representing the collaborations. We need for a more standardized way of representing RUP disciplines, the workers, the activities they perform and the artifacts they produce.

We have tried to use only activity diagrams to represent the process disciplines, including the dependency between the roles, activities and artifacts. When we represent the different roles in the process, everything works fine, even if the diagram has duplication from the role interaction (see Appendix A.7, Figure 114 to Figure 116, page 187-189). We represent each role by using swimlanes, and role interaction, by duplicating the activity in the swimlane of each role that performs it.

The activity diagrams don't support the notion of artifacts. We have introduced a way to represent them. We include a column for the INPUTS artifacts and a column for the OUTPUTS artifacts. If an activity needs an artifact from the INPUTS column, there is a dotted line with arrowhead from the input to the activity. If an activity produces an artifact, there is a dotted line with an arrowhead from the activity to the artifact in the OUTPUTS column. This notation is inspired by the BPMN notation (see section 2.2.2, page 8), that uses associations to link a flow object (activity, etc.) to a data object (artifact). It is slightly different from the BPMN, because it focuses more on the inputs and outputs of activities, than on the artifacts flow.

Since in RUP, there are a lot of artifacts involved, the diagram becomes at a certain point unreadable, as we can see in the following figure:

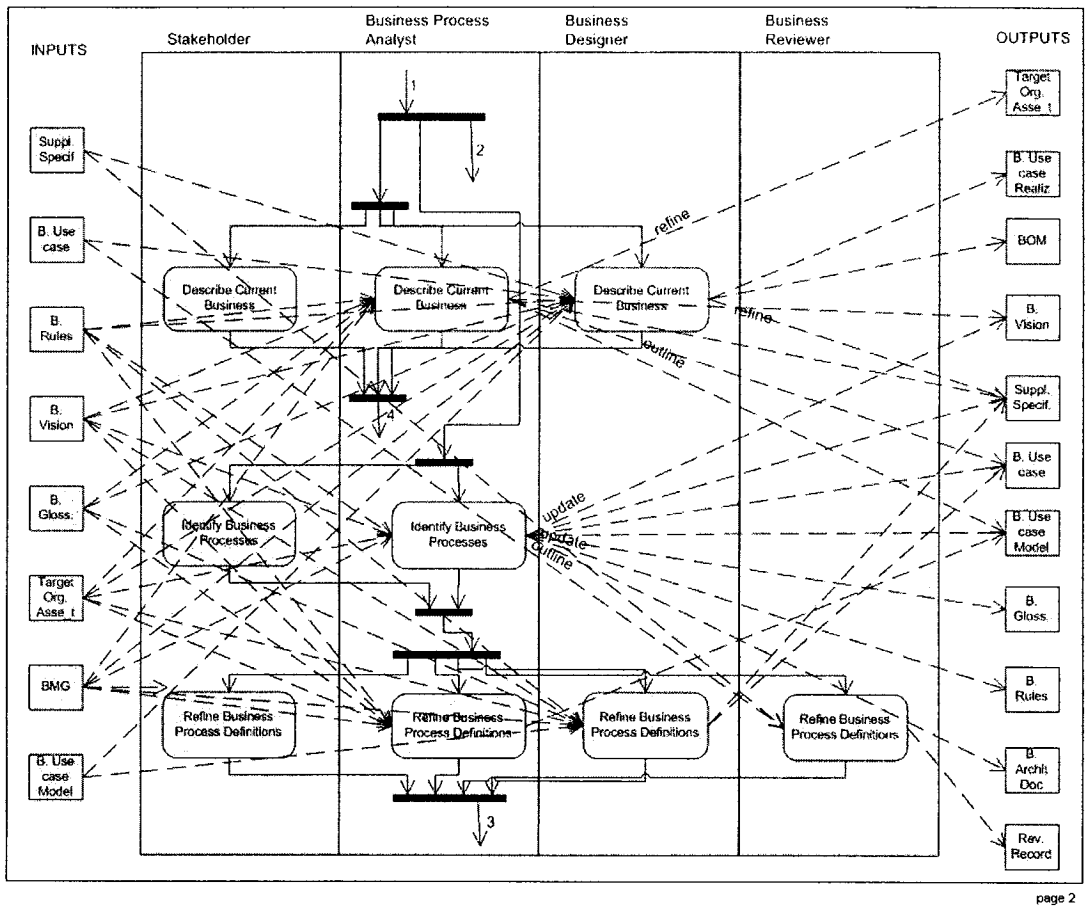


Figure 16: Excerpt from the Activity diagram for the Business Modeling discipline featuring roles, activities and artifacts (Figure 117 to Figure 119, page 190-192)

This diagram superimposes different views of the activities that require and produce artifacts. Should we have decided to represent multiple views, for different actors or activities, we would not have such complicated diagrams. We need single view diagrams to get an overall picture of the complexity of the process. The activity diagrams are just not optimal for the representation of single views of a complex process. The XRAD notation (an extension of the RAD – Role Activity Diagram – notation), which we will present in chapter four, is more suited for the representation of a large number of artifacts, as is the case in RUP.

3.5 Summary

In this Chapter we presented an overview of RUP, its main characteristics and RUP core disciplines. RUP is a use-case driven, architecture-centric and iterative and incremental approach. It focuses on the use cases which provide value to the users and Stakeholders. It also focuses on the architecture as the basis to build the system. The use cases and the architecture are developed in parallel, through iterations. Each iteration ends with a functional release of the system (build), which enables constant Stakeholder feedback and gives assurance to developers that they work in the right direction. It also helps address risks early in the lifecycle.

Each iteration goes through the core disciplines, with emphasis on some of them, depending on the phases. Each discipline is a collaboration of workers, activities and artifacts. We have presented in this chapter the purposes of the disciplines and the main activities. We will refer to the annexes A1 to A6 for a complete description of the workers, artifacts and activities that are present in RUP.

Chapter 4 : RAD applied to RUP

In this chapter, we are proposing the *Role Activity Diagrams* (RAD) as a better alternative to model the RUP, than what already exists to date. We will present the RAD notation and symbols using examples from the diagrams that we have developed during our research. We have developed our diagrams sequentially, starting with limited versions, and finishing with improved and complete new versions. By this fact, we can show that our diagrams are a good means of modeling a process. We will present the three version diagrams, which represent the modeling of the core RUP disciplines, by showing how we built on older versions in order to construct better ones.

4.1 Modeling with RAD

The RAD diagrams are used to model a process. They show the roles, their component activities and their interactions, together with external events and the logic that determines what activities are carried out when. In this section, we will refer to [4] and [11] in order to describe the notations and symbols used to construct the RAD diagrams and to figures from appendix A.8 in order to give some examples.

4.1.1 Representing Roles and Activities

Each role in the process is represented by the contents of a shaded block. In Figure 123 (see appendix A.8, page 196), there are four roles with the names *Stakeholder*, *Business Process Analyst*, *Business Designer* and *Business Reviewer*. All activities and interactions take place within those four roles, as far as this model of this process is concerned.

Within a role, there are a number of activities indicated by black boxes. The annotation against each black box describes the activity. However, in the previous model, there is no atomic activity performed by a single role alone. In Figure 136 (see appendix A.8, page 209), the Integrator role carries out an activity "Plan the Integration".

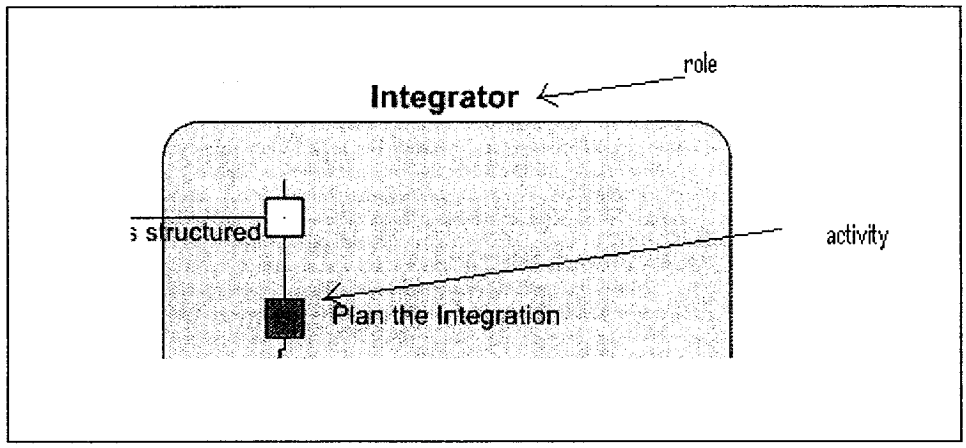


Figure 17: Representing roles and activities (excerpt from Figure 136, page 209)

4.1.2 Representing Roles Being started – Role Instantiation

One role can be instantiated, i.e. a new instance of that role can be started: this is indicated by a square with a cross inside it. When we want to represent the "ending" of a role instance, we use the "Stop" symbol.

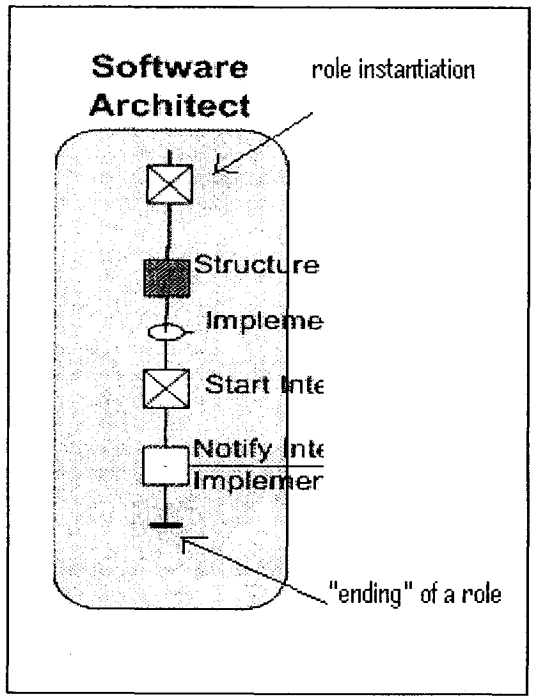


Figure 18: Representing role instantiation and role "ending" (excerpt from Figure 136, page 209)

4.1.3 Representing interactions

An interaction between roles is shown as a white box in one role connected by a horizontal line to a white box in another role. The white boxes in each role are called "part-interactions". An interaction can involve any number of roles and signifies that the roles involved must synchronize. In Figure 123 (see appendix A.8, page 196), the Business Process Analyst interacts with Business Designer, Business Reviewer and Stakeholder to Develop Domain Model.

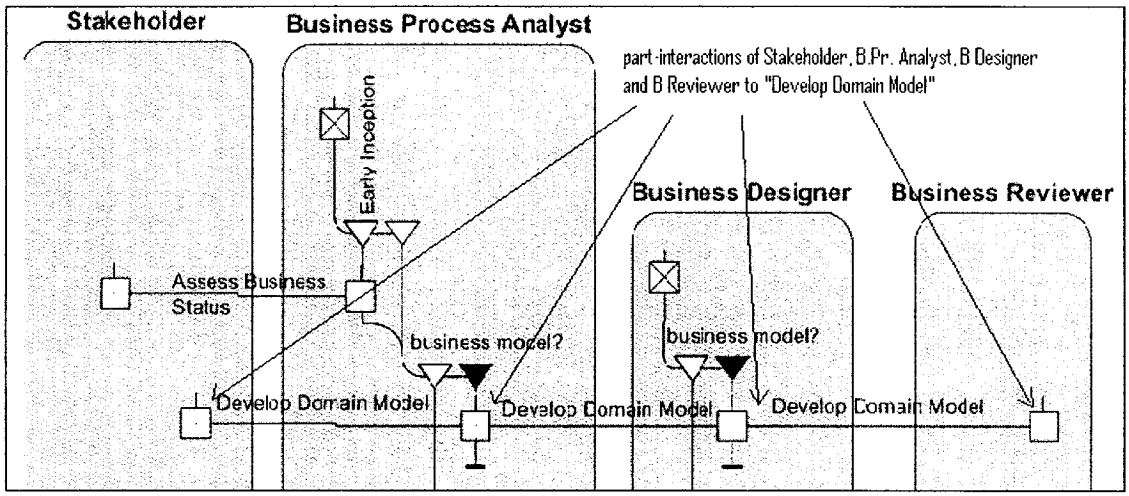


Figure 19: Representing part-interactions (excerpt from Figure 123, page 196)

In the original RAD diagrams, the interaction lines do not carry arrows to indicate the "flow" of materials; they are placed appropriate annotations (see Figure 124 to Figure 127 in appendix A.8, pages 197-200).

We will propose (see section 4.3) the use of arrows to show the entities flow in the extended version of RAD (called XRAD).

4.1.4 Representing Roles and States

In RAD, the state is represented by a vertical line which connects the activities within a role. A state can represent a condition which the role can be in, and which is necessary for an activity to occur. We represent this condition by putting a "magnifying glass" on the state line and annotating it.

For example, in Figure 130 (see appendix A.8, page 203), we need to show the state of the System Analyst, "Work in Scope", because it is a pre-condition for the activity "Refine System Definition" to be performed by the Requirements Specifier and UI Designer roles.

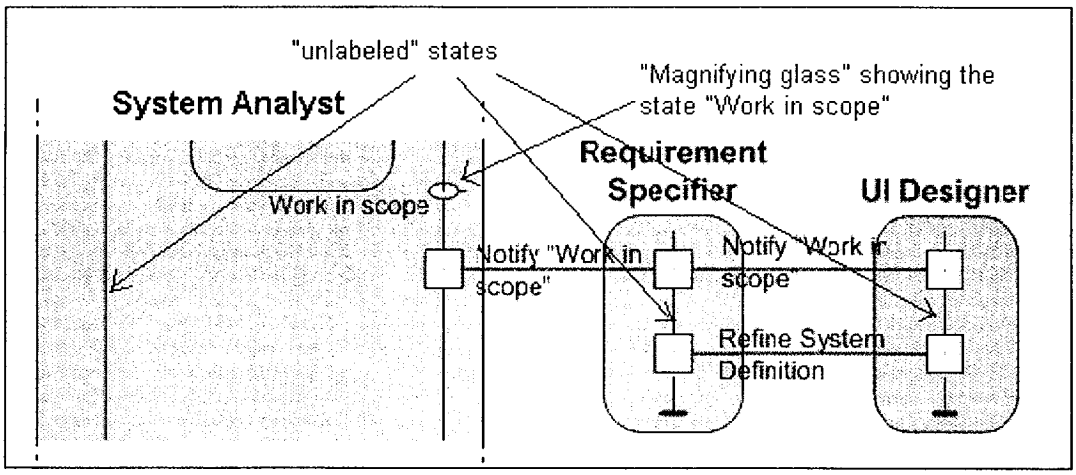


Figure 20: Representing states (excerpt from Figure 130, page 203)

4.1.5 Representing Alternative Courses of Action

At some points in the process, what happens next might depend on some condition or state. We represent such *alternative courses* of action with the notation shown in the stencil (see Figure 120 in appendix A.8, page 193) for case refinement. We are refining the state of the process according to different predicates or "cases".

For example, in Figure 133 (see appendix A.8, page 206), the software architect takes different courses of action depending on which of the phases the process is in at the moment: "Inception", "Early Elaboration" or "Other".

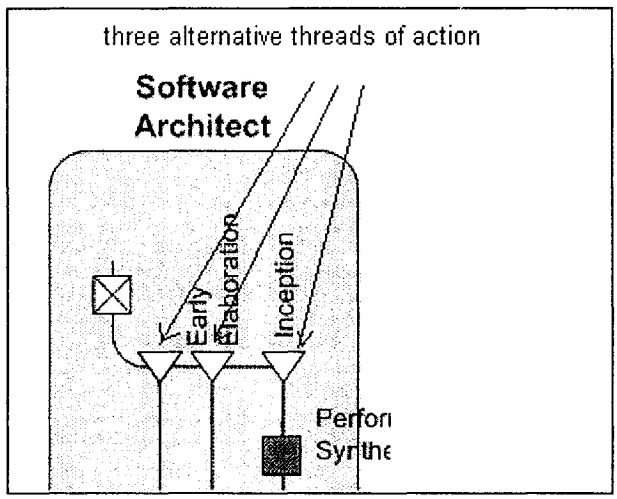


Figure 21: Representing alternative threads (excerpt from Figure 133, page 206)

In Figure 130 (see appendix A.8, page 203), we have the *predicate* "New System?" in the System Analyst Role. The System Analyst will take the thread that corresponds to the predicate that is true. The black case refinement corresponds to the negative answer, while the white one corresponds to the positive. If this is a new system, the System Analyst together with Stakeholder will "Analyze the Problem"; otherwise, he will skip this activity and go directly to "Understand Stakeholder Needs".

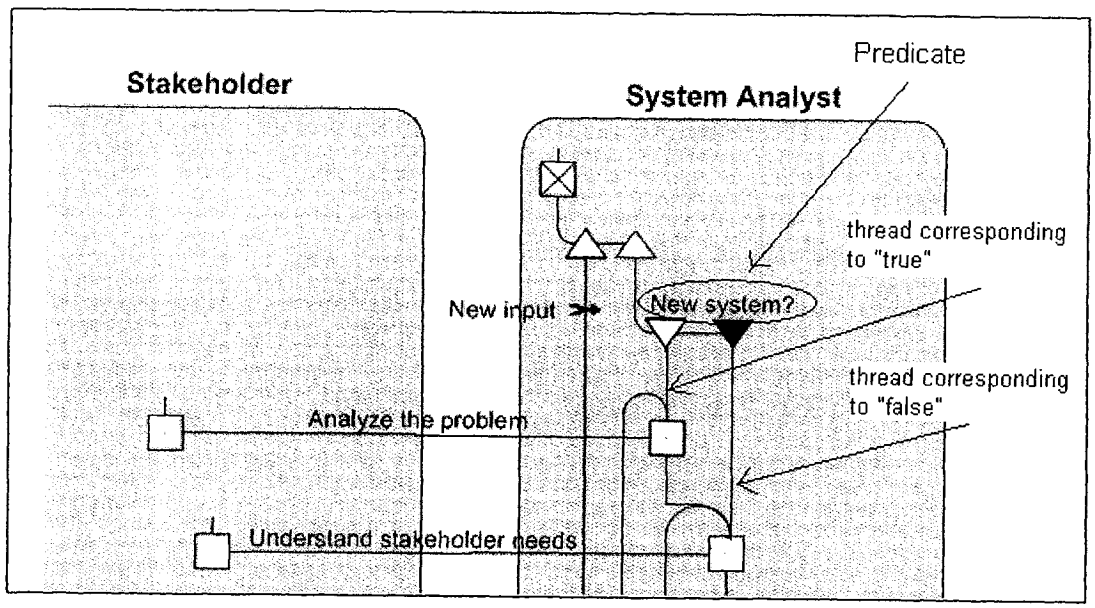


Figure 22: Representing a predicate and two alternative threads (excerpt from Figure 130, page 203)

It is important to note that no person or machine is doing anything to make the decision: the process is simply going in different directions depending on the state it is in.

Case refinement allows us to represent conditional iteration within a role. Let's consider again the Figure 130 (see appendix A.8, page 203).

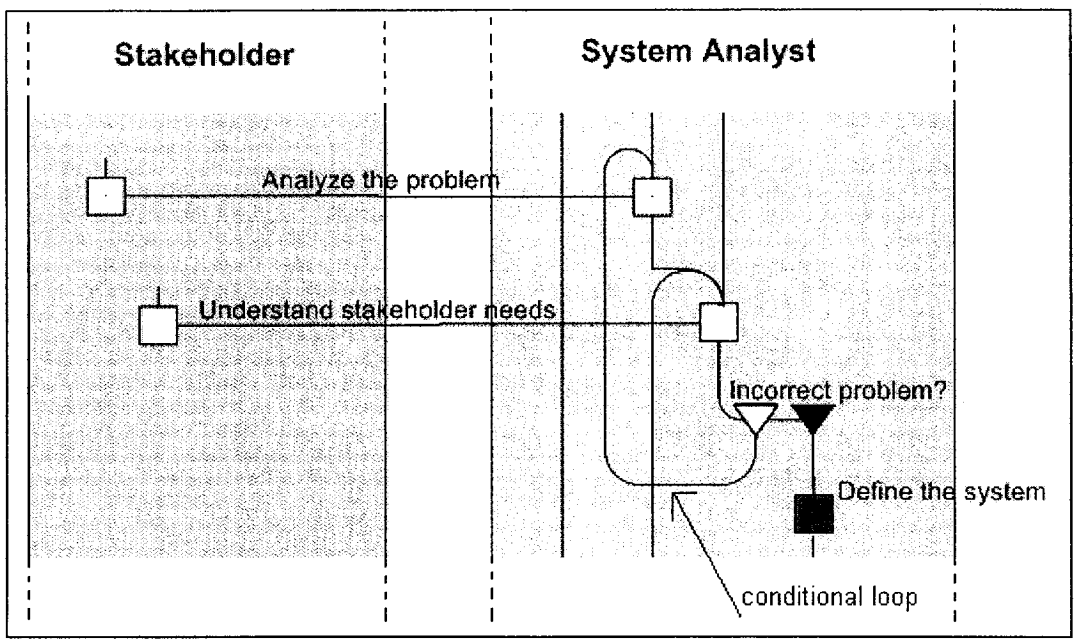


Figure 23: Representing conditional iteration (excerpt from Figure 130, page 203)

The predicate "Incorrect Problem" in the System Analyst role allows the System Analyst to go back and "Analyze the Problem" until it is the correct one. When the problem is correct, it can go on with the process. We use the conditional loop that gets out the case refinement corresponding to positive answer to the predicate question, and going in the activity "Analyze the problem".

In some situations, whichever of the alternative threads of activity is followed, we finally want to "return" to some "main" thread of activity. In this case, we use the "*refinement closing*" symbol, as illustrated in Figure 142 (see appendix A.8, page 214) with the threads joining up again when they have finished (i.e. the case states are recombined).

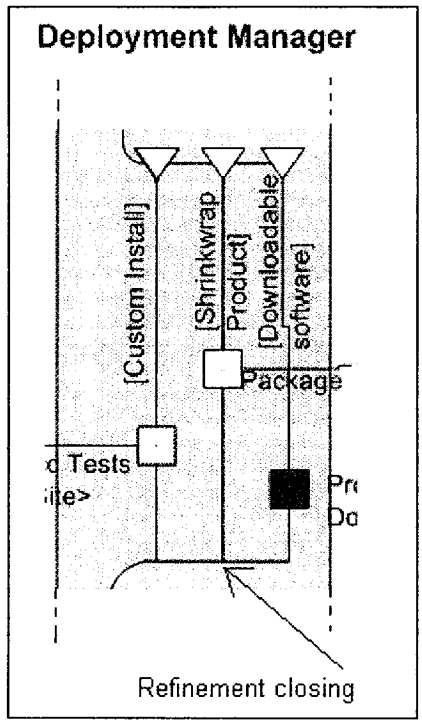


Figure 24: Joining-up alternative threads (excerpt from Figure 142, page 214)

In other situations, a process does not operate this way. The example is in Figure 130 (see appendix A.8, page 203): if the problem is correct, we continue with the process, otherwise, a rework is necessary, we go back to "Analyze the problem". The two alternative threads *do not recombine.*

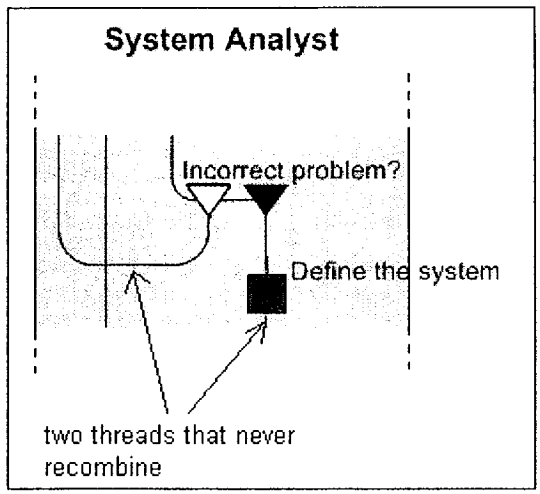


Figure 25: Representing threads that do not recombine (excerpt from Figure 130, page 203)

4.1.6 Representing concurrent threads of action

Sometimes, a role can start a number of separate threads of activity that can be carried out *concurrently*. This is represented in the RAD by the symbol for "part refinement". The state of the role is divided into a number of separate parts. Part refinement can involve any number of threads of concurrent activity, depending on how much concurrency is possible in the work of the role.

In Figure 123 (see appendix A.8, page 196), the Business Process Analyst and the Business Designer both carry out three concurrent threads of activity.

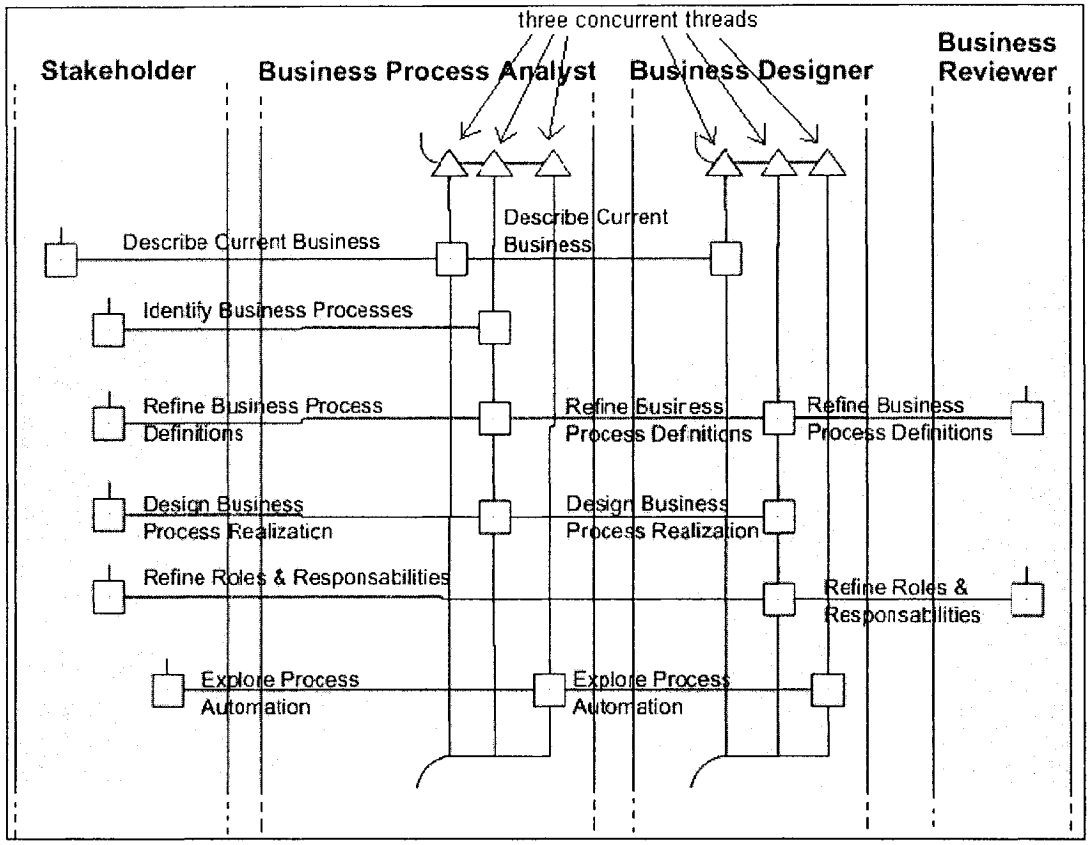


Figure 26: Representing three concurrent threads (excerpt from Figure 123, page 196)

On one thread, the Business Process Analyst, together with the Business Designer and the Stakeholder, they "Describe Current Business". On the second thread, he carries out three activities: "Identify Business Processes" (with Stakeholder), "Refine Business Process Definitions" (with Stakeholder, Business Designer and Business Reviewer), and "Design Business Process Realization" (with Stakeholder and Designer). On the third thread, he carries out the activity "Explore Process Automation" (with Stakeholder and Business Designer).

We can use the same symbol as we have done for case refinement to join all threads together once they are finished. It is what happens in the previous example, once the Business Process Analyst and the Business Designer have resumed the three threads.

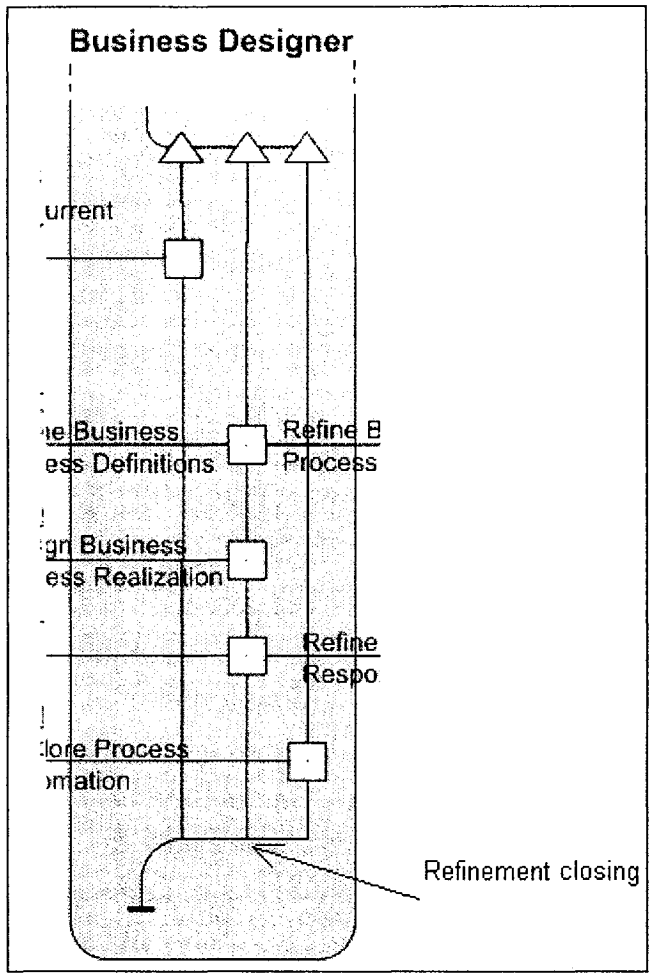


Figure 27: Joining-up concurrent threads (excerpt from Figure 123, page 196)

In some cases, however, a process might not operate in this way. Some threads of a part refinement may never need to recombine. For example, in Figure 139 (see appendix A.8, page 212) the Test Manger, after interacting with the Test Analyst, Test Designer and Tester, on the second thread, he can continue somewhere else in the process (it's not interesting for the purpose of this model); the thread never recombines with the first one.

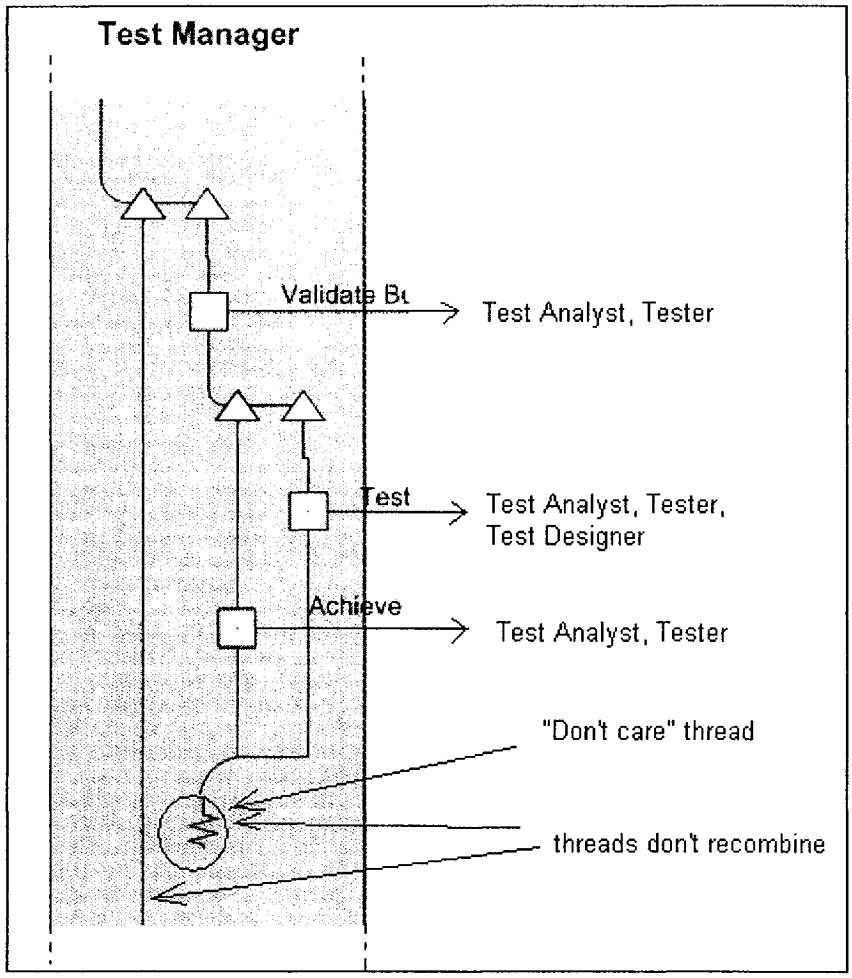


Figure 28: Another case of threads that don't recombine (excerpt from Figure 139, page 212)

We can note the fact that the case refinement and the part refinement closings are equivalent to the synchronization bars in the AD diagrams.

4.1.7 Representing external events

We show an event by an arrow placed on the state line and annotated with a brief description of the event concerned. In Figure 130 (see Appendix A.8, page 203), we see that as soon as there is "new input" coming in, the other thread of the System Analyst is activated, which is a triggering event for the activity

"Manage Changing Requirements" to be carried out together with the Stakeholder and the Requirements Reviewer.

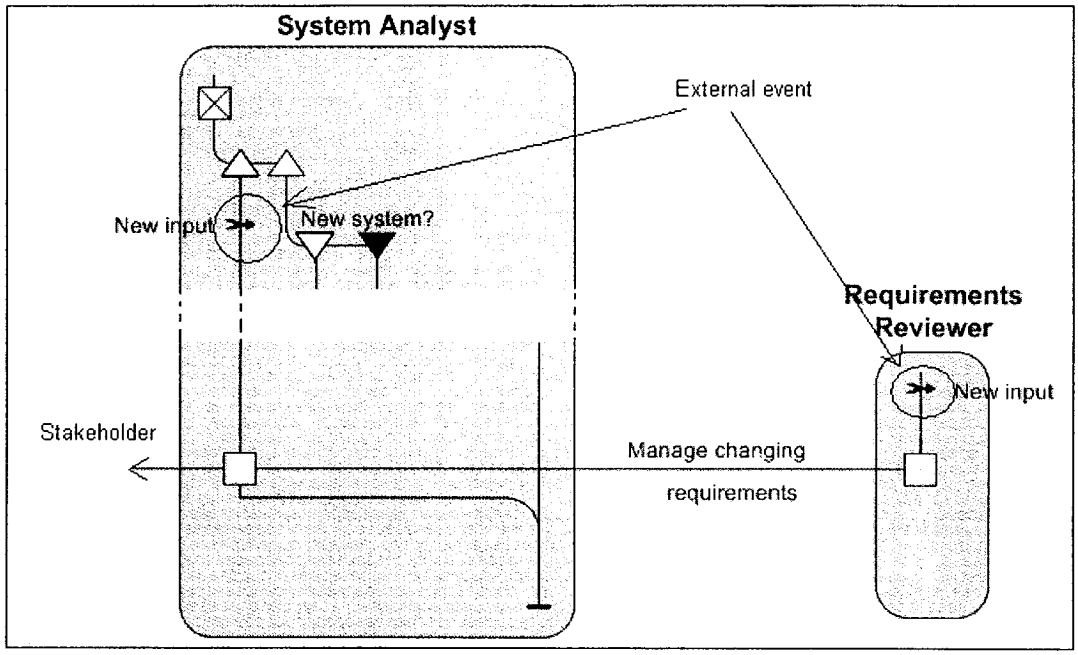


Figure 29: Representing an external event (excerpt from Figure 130, page 203)

4.1.8 Modeling the materials in the process

There are different ways to insert the materials flow into the already existing role model. One of them is the use of the *central repository*. When a role, for example Business Process Analyst, produces an artifact, like Business Use Case Model, this artifact can find its way into the central repository. From there, it can be accessed by another role, Business Designer, without explicit transfer from the role body of Business Process Analyst to the role body of Business Designer. In this case, the artifact "Business use case model" is shared. However, if we wanted to establish that the materials flow was coherent, we would need to show the explicit passage of the artifact from one role to another.

In the original RAD diagrams, the materials required by the process are represented as shown in Figure 124 to Figure 127 (see appendix A.8, pages 197 to 200). They appear as the "grams" passing between roles at interactions.

In order to show the production of an artifact, we draw the black box of the activity that creates it. If an entity is needed as input to an activity, there are two cases. The first case is when the same role produced it and later needs it. In this case, nothing is specified in the diagram. Everything a role produced before, it can be used later by that same role.

Let's consider Figure 127 (see appendix A.8, page 200). When the Business Designer and the Business Model Reviewer interact to "Refine Roles and Responsibilities", the Business Designer does two "micro-activities", *Detail a Business Worker* and *Detail a Business Entity*.

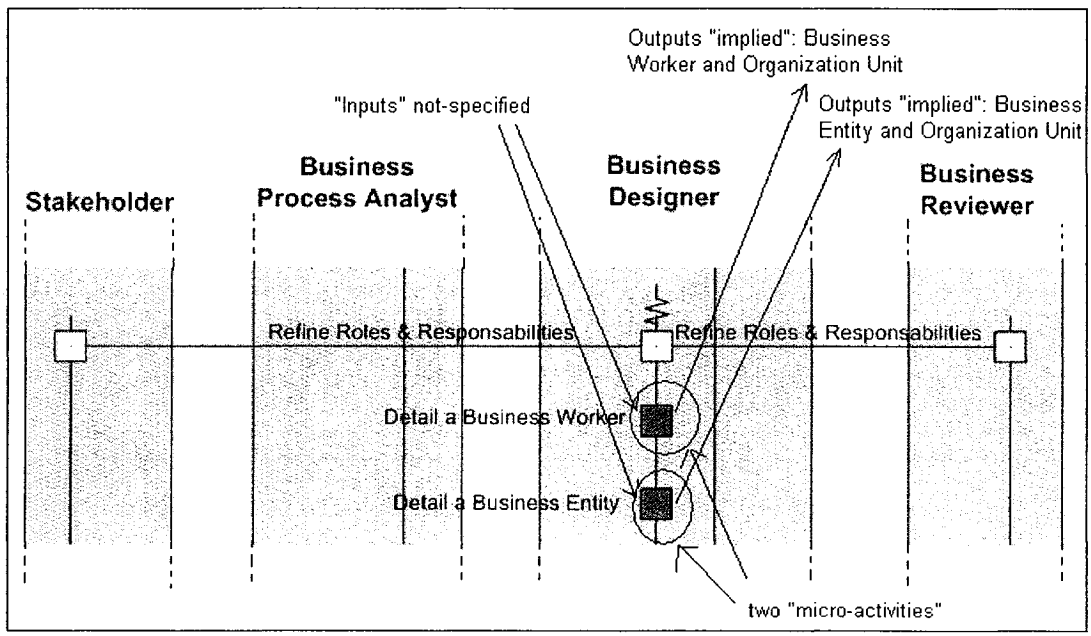


Figure 30: Representing the production of artifacts during the "micro-activities" (excerpt from Figure 127, page 200)

He uses artifacts that were previously sent to him by the Business Process Analyst during the interaction "Design Business Process Realizations", and artifacts that he produced also during the same interaction. So, that's why these input artifacts don't show up on the diagram.

During the activity "Detail a Business Worker", he will produce two artifacts: *Business Worker* and *Organization Unit*. During the activity *Detail a Business Entity*, he will produce two artifacts: *Business Entity* and *Organization Unit*, as well. We don't mention which artifacts are produced, because we can "guess" by the annotation placed next to the black box, which artifact will be produced. The second case is when a role needs an artifact produced previously by another role; the latter has to send it to the first by an interaction (white box). We can consider the previous example.

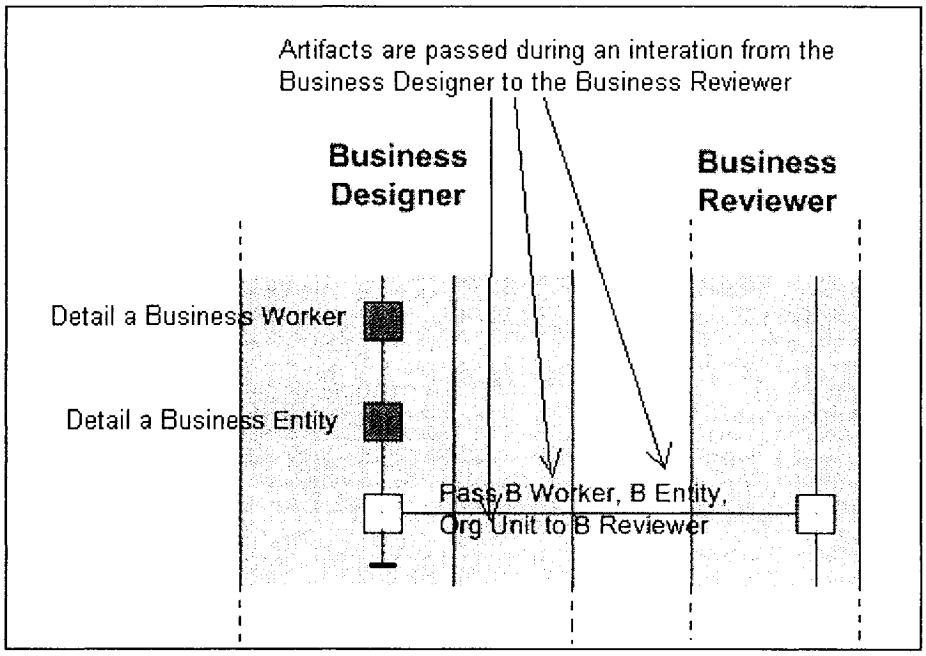


Figure 31: Representing the transfer of artifacts from one role to another at interaction time (excerpt from Figure 127, page 200)

After having finished the two activities, the Business Designer will pass the three artifacts produced to the Business Reviewer by the intermediate of an interaction. The Business Reviewer, together with the Stakeholder will Review the Business Object Model and produce a Review Record.

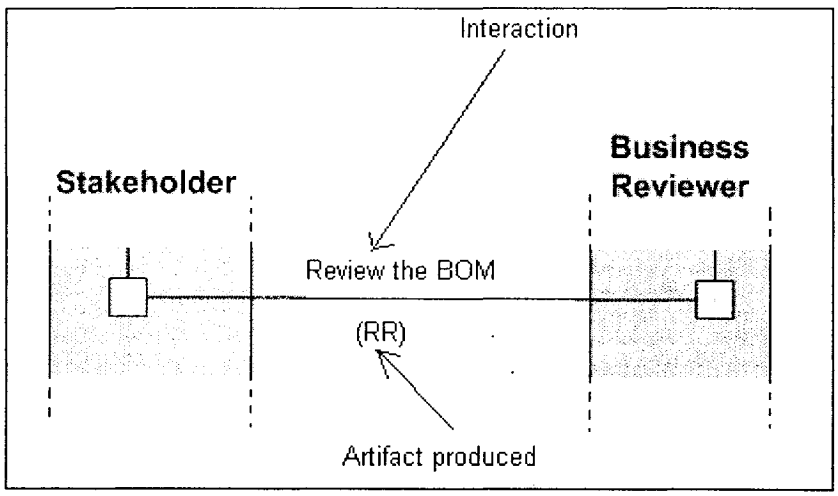


Figure 32: Representing the production of an artifact during an interaction (excerpt from Figure 127, page 200)

4.2 RADs applied to RUP

As we could see in the previous section, RAD diagrams are a convenient approach when modeling the interactions between different roles. The activity diagrams which are presently used to represent the RUP disciplines allow the representation of interactions, but when we add the artifacts, the diagrams become very complicated and unreadable (see example in section 3.4.7, page 44). That is why we developed XRAD (an extension to RAD) to represent RUP artifacts and to replace the existing activity diagrams.

We developed three versions of the model. The first two versions are RAD diagrams. They only express the relationship between the activities and the roles in the RUP. The first version is inspired by references [1] and [2] and the second

version by reference [3]. The second version is more complete and tries to correct mistakes found in the first version. The third version, called "XRAD", is an extension to RAD. It builds on the second version RAD and adds to it the *artifacts* involved in RUP. It introduces the use of the arrows to express the entities flow.

4.2.1 First version RADs

This first version tries to combine information coming from different diagrams and documents. It first uses the original activity diagrams for each discipline (as they appear in Chapter 3). Then, it uses a document from reference [2], which describes the activities performed and the artifacts produced during a workflow detail. Since we can associate to each worker, the activities he can perform and the artifacts he produces ([2]: appendices), we can try to guess the actors that collaborate during a workflow detail. That's how we obtain the RAD diagrams, for each core discipline, involving interactions among workers.

4.2.1.1 Business Modeling

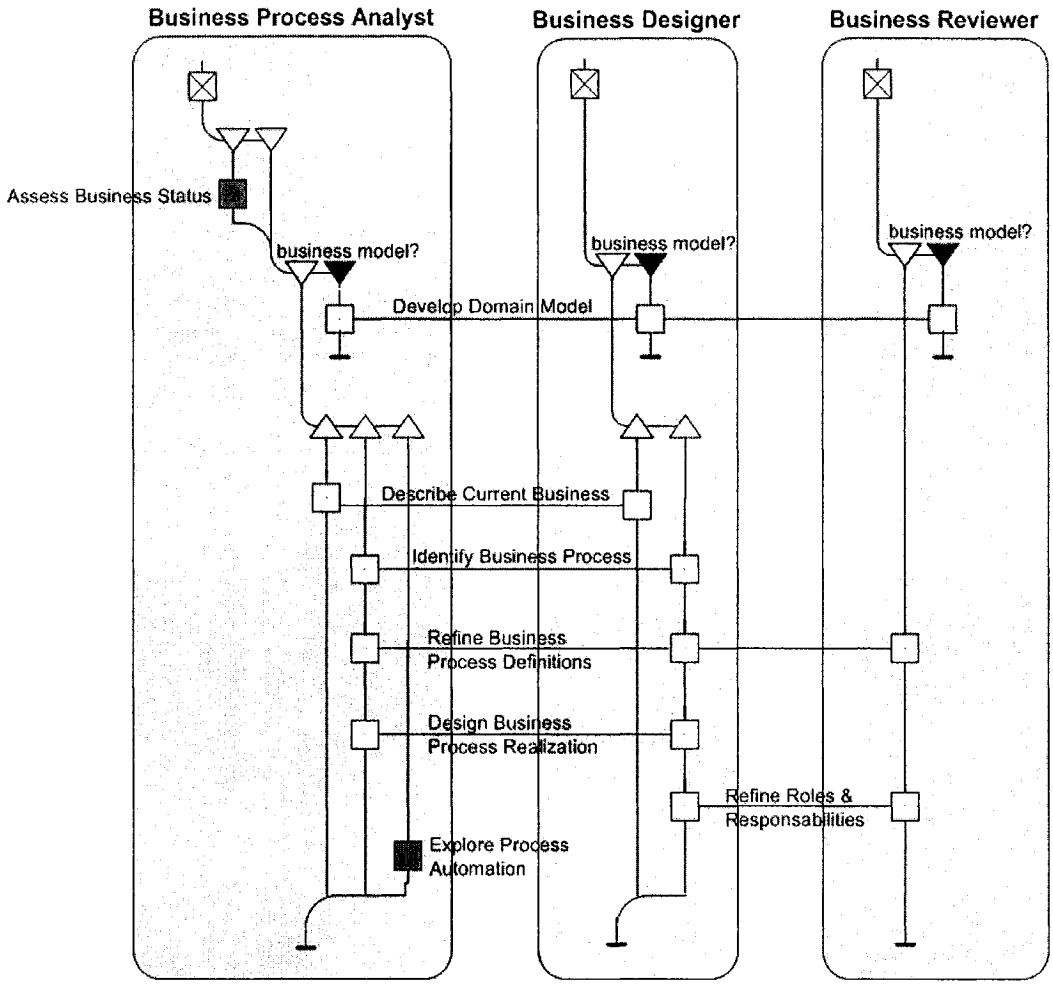


Figure 33: Role Activity Diagram for the *Business Modeling* discipline (v. 1)

The problem with this diagram is that the Business Reviewer role is idle from the time it is instantiated and until he is involved in the activity "Refine Business Process Definitions". Another problem with this diagram is that we understand that the Business Reviewer is also responsible for deciding if it is a business model. This can not be the case, since he is only responsible for reviewing the different models. The third problem is that the Stakeholder is not shown as being part of the process, while he is strongly involved in many activities.

4.2.1.2 Requirements

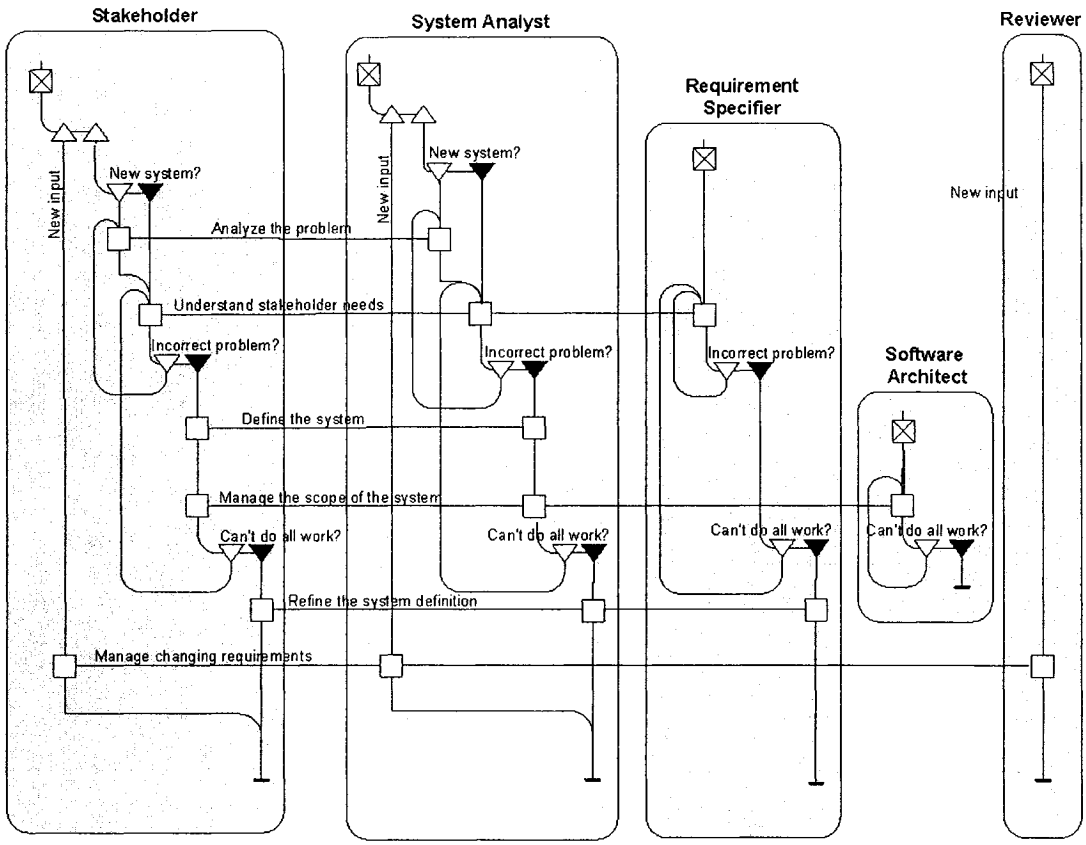


Figure 34: Role Activity Diagram for the *Requirements* discipline (v. 1)

The first problem we can notice in this diagram is that the Stakeholder is involved in the process, at the same level as the System Analyst. Even if he plays an important part in the different activities, he will not play the same role as a worker in the process. We need to find another way of expressing the Stakeholder's involvement. Another problem is that some minor roles (Requirements Specifier and Reviewer) are idle a long time, waiting for the System Analyst to perform some activities together. We should find a way to modify that.

4.2.1.3 Analysis and Design

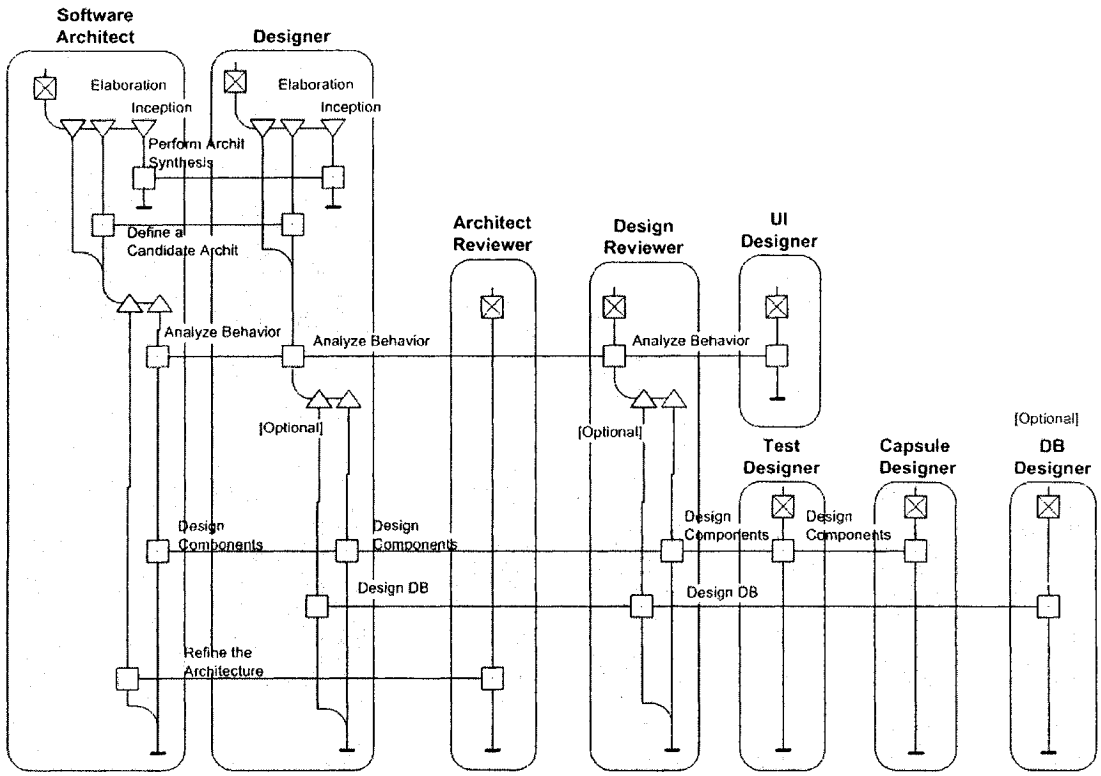


Figure 35: Role Activity Diagram for the *Analysis and Design* discipline (v. 1)

There is no major problem with this diagram, except for the Architect Reviewer, who is idle for some time.

4.2.1.4 Implementation

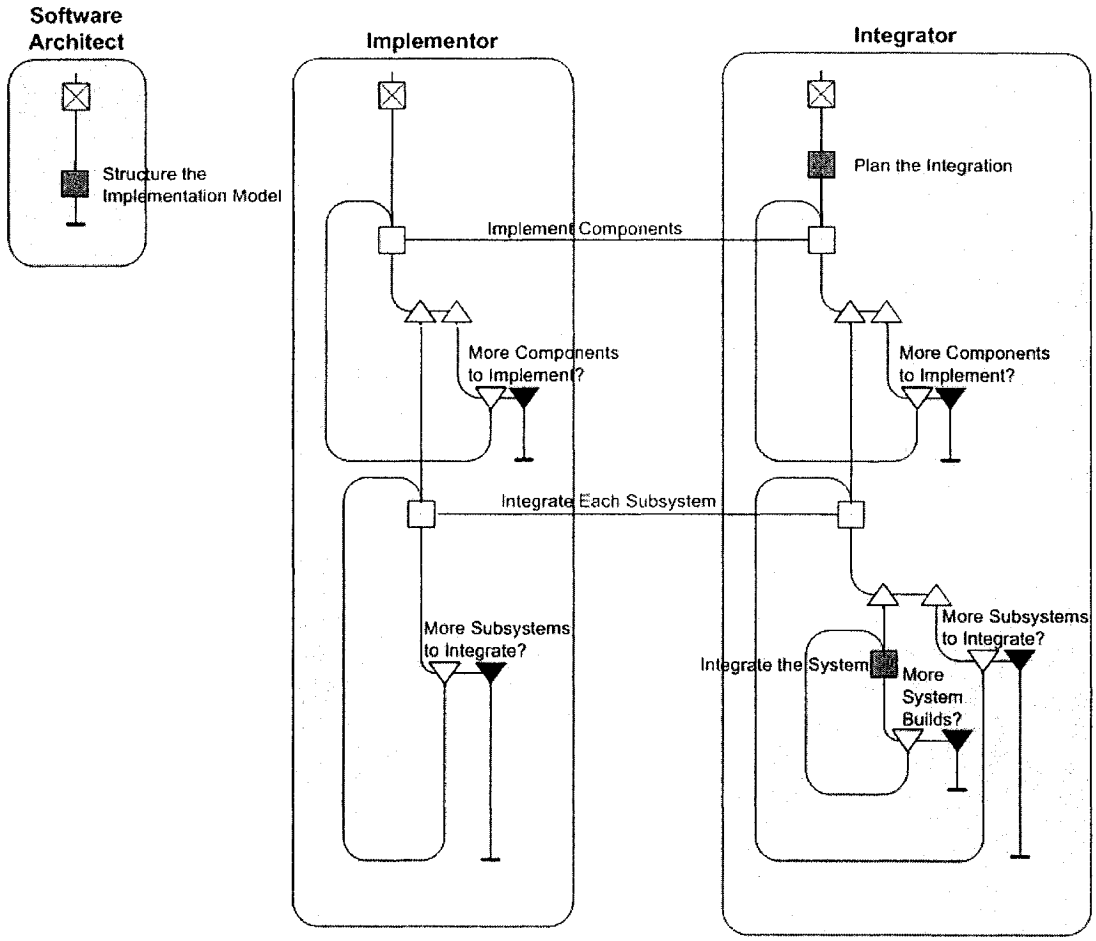


Figure 36: Role Activity Diagram for the *Implementation* discipline (v. 1)

The main problem with this diagram is that the Software Architect is not linked in any way to the process. The Integrator role, as well as the Implementer, depends on the artifacts produced by the Software Architect, in order to follow with the process. This is a flaw that we'll correct in the second version.

4.2.1.5 Test

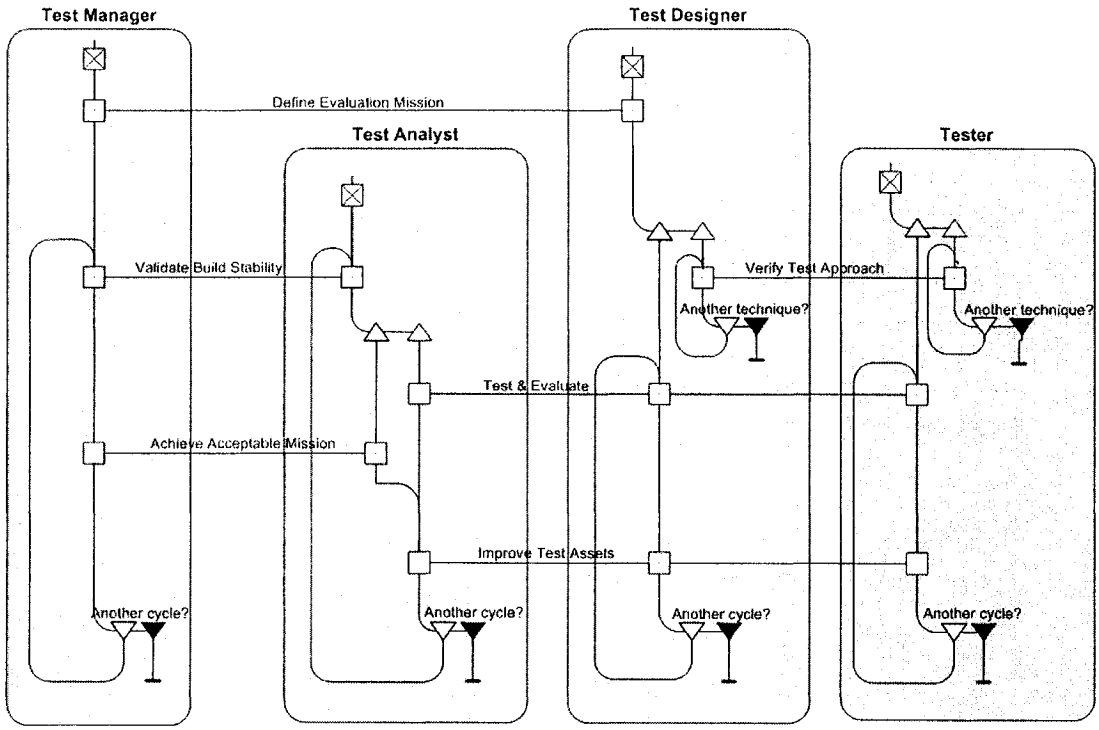


Figure 37: Role Activity Diagram for the *Test* discipline (v. 1)

The problem with this diagram is that the Test Designer is idle for some time (between the activities "Define Evaluation Mission" and "Test and Evaluate").

4.2.1.6 Deployment

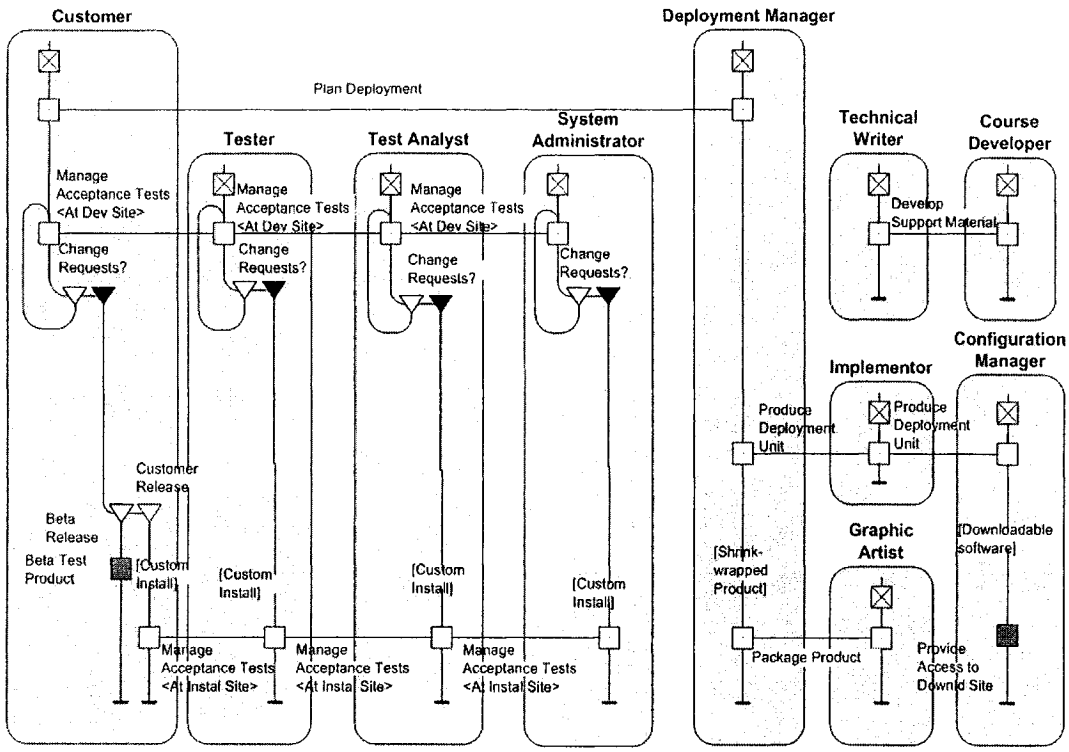


Figure 38: Role Activity Diagram for the *Deployment* discipline (v. 1)

The main problem in this diagram, as in the Implementation diagram (see Figure 36, on page 68), is that the Technical Writer and the Course Developer are not linked to the process. We need to find a way to link them to the process logic (see RAD second version of the Deployment discipline, Figure 44, on page 79).

4.2.1.7 Limitations

The limitations with the first version come mainly from the fact that we try to guess, by looking at activities performed and the artifacts produced in each workflow detail, which actors are responsible for doing these activities and producing these artifacts. Sometimes, we can have some artifacts, but not the activities by which they are produced, and sometimes the reverse. So, we can make mistakes in the actual model because we don't have complete information

about the process. These mistakes could have been corrected only after we've got a more complete reference, i.e., [3].

There are some other mistakes that we could see directly from the diagrams themselves, as pointed out previously, for each discipline. This proves that our model is reliable, and we can improve it. The second version RADs are an improved version for two reasons: they use a more complete reference for the RUP and they correct the mistakes of the model that we detected in the first version.

4.2.2 Second version RADs

We developed the second version by combining the activity diagrams for each discipline with the complementary diagrams of the workflow details. In the same time, we corrected the mistakes found in the first version.

4.2.2.1 Business Modeling

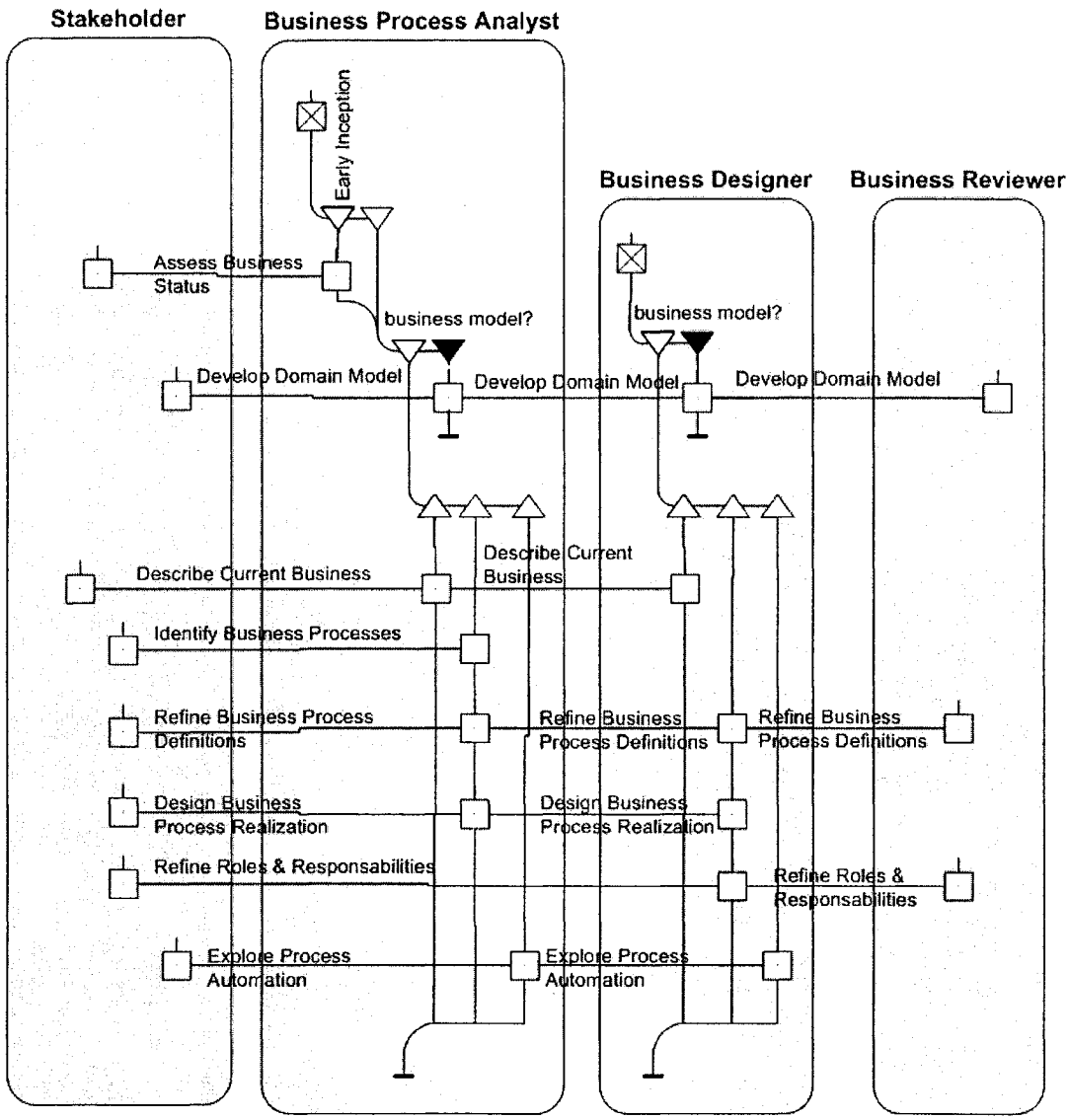


Figure 39: Role Activity Diagram for the *Business Modeling* discipline (v. 2)

In this version, we introduced the Stakeholder at every level of the process. The Stakeholder provides valuable feedback. The reason why the activities in the Stakeholder role are not connected by the state line (in this case, they are called "hanging"), is because the Stakeholder is only doing these activities on demand (just like the Business Reviewer).

We avoided the Business Reviewer role being idle for a long time, by making him act on demand of the Business Designer (he can do other activities meanwhile). In the second version, the Business Process Analyst is interacting with the Business Designer to "Explore Process Automation", while in the first version he is doing this activity alone. This is due to the fact that in the second version we used another reference [3] that showed us the Business Designer is involved.

4.2.2.2 Requirements

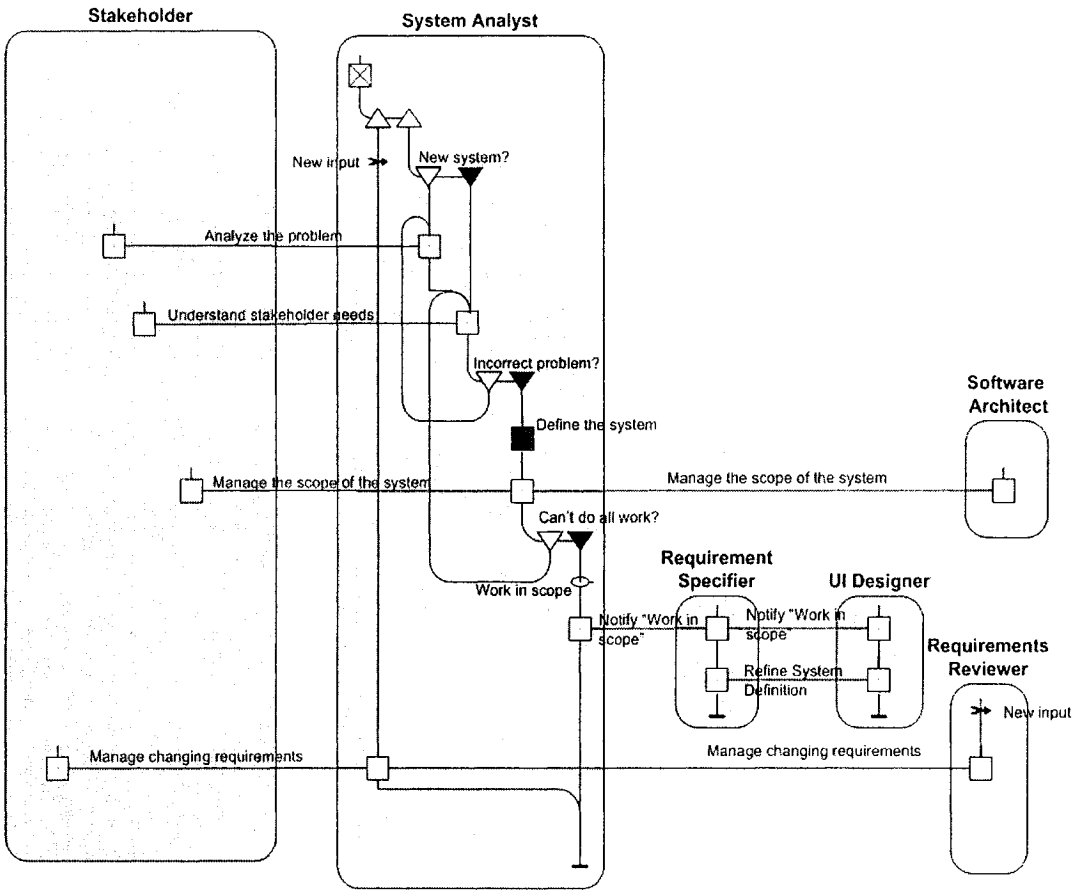


Figure 40: Role Activity Diagram for the *Requirements* discipline (v.2)

In the second version of the Requirements discipline, we decided to make just one key role for the process, the System Analyst. The Requirements Specifier, the Software Architect and the Requirements Reviewer are roles that will act on demand of the System Analyst role (for the same reason as in the preceding

example, to avoid these roles being idle for a long time, waiting for other roles to perform some activities). We also introduced in the second version, the UID (User Interface Designer) role (this role used to be part of the Analysis and Design discipline, according to the first reference we used [2]; but in the latest reference [3], he was part of the Requirements discipline).

The Stakeholder can not be part of the process (as shown in the first version, see Figure 34, page 66), even if he is present at most of the levels in the process. We make this role act on demand of the System Analyst role, by making the activities in this role "hanging". In the second version, we improve on the clarity of the process model by introducing two new notations: the "Magnifying Glass" and the "Triggering Event". The "Magnifying Glass" underlines the state "Work in scope", after it was previously established that all work could be done.

This is a necessary condition for the Requirements Specifier and the UI Designer to "Refine the System Definition". The System Analyst will notify the state the process is in to the Requirements Specifier and the UI Designer, so they can undergo the activity. "New Input" is a triggering event for the System Analyst and the Requirements Reviewer to start the activity "Manage Changing Requirements".

4.2.2.3 Analysis and Design

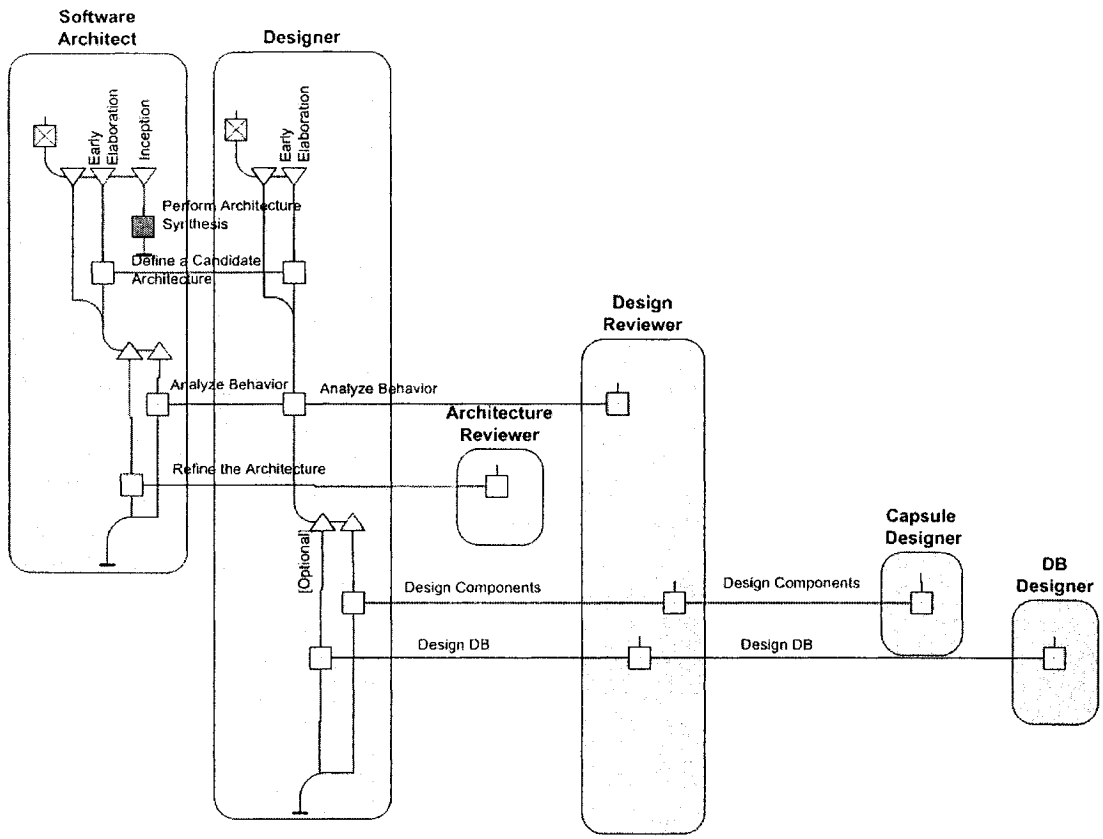


Figure 41: Role Activity Diagram for the *Analysis and Design* discipline (v. 2)

In the second version of the Analysis and Design discipline, we eliminated the UI Designer role (since we integrated it in the Requirements discipline) and the Test Designer Role (which is part of the Test Discipline). The Designer and the Design Reviewer don't interact directly with the Test Designer to "Design Components"; they only need artifacts developed in the Test discipline by the Test Designer role. We can see this by looking at the diagram containing the artifacts flow (see Figure 53, page 92).

In the second version, there are two key roles: Software Architect and Designer, while the other roles are acted upon request of the two principal roles (the activities in these roles are "hanging"). The other differences between the first

(see Figure 35, page 67) and the second version concern updates from the latest reference used [3]. Hence, the Software Architect does the activity "Perform architecture synthesis" all by himself, and he is not involved in the interaction: "Design components".

4.2.2.4 Implementation

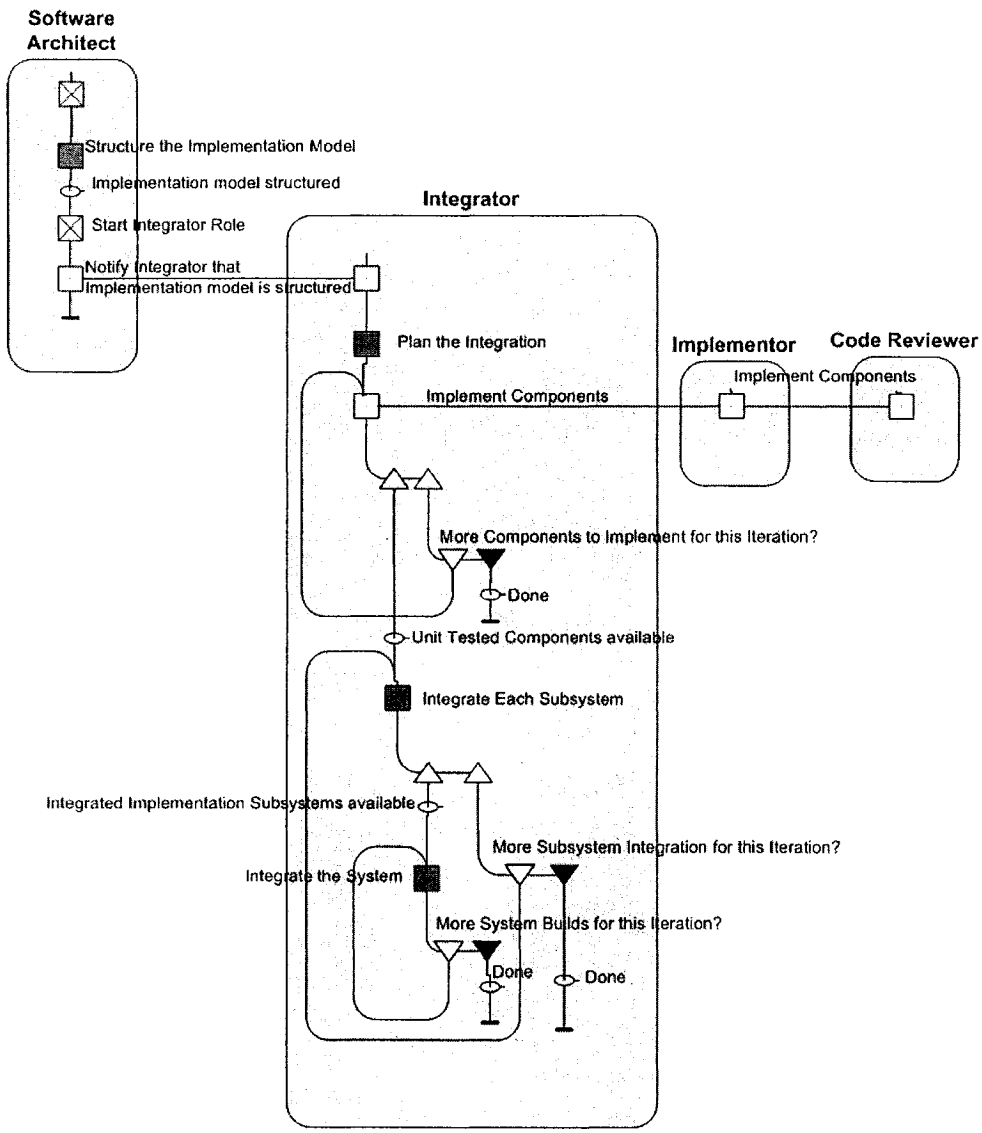


Figure 42: Role Activity Diagram for the *Implementation* discipline (v. 2)

In the first version (see Figure 36, page 68), there was a flaw. The software architect was not linked to the process in any way. We came up with a solution to correct this. The "magnifying glass" underlines the state following the activity performed by the Software Architect. This is considered a triggering condition to start the Integrator role. The Software Architect will interact with the Integrator and notify him that he can start the activities.

In the second version of the implementation discipline, we introduced a new role (Code Reviewer) as an update of the latest reference [3] used. The Implementer, as well as the Code Reviewer roles, are acted only on demand, because they only participate in the "Implement Components" interaction (in the first version, see Figure 36, page 68, the Implementer also interacts with the Integrator to "Implement Components", but according to the latest reference [3], it is no longer the case, the Integrator doing this activity alone).

Other updates concern the use of the "Magnifying glass" to underline the different states the process is at different times. For example, it is used to show some threads are finished ("Done"), or that we have the necessary condition for some activities to start ("Unit Tested Components Available", "Integrated Implementation Subsystems Available").

4.2.2.5 Test

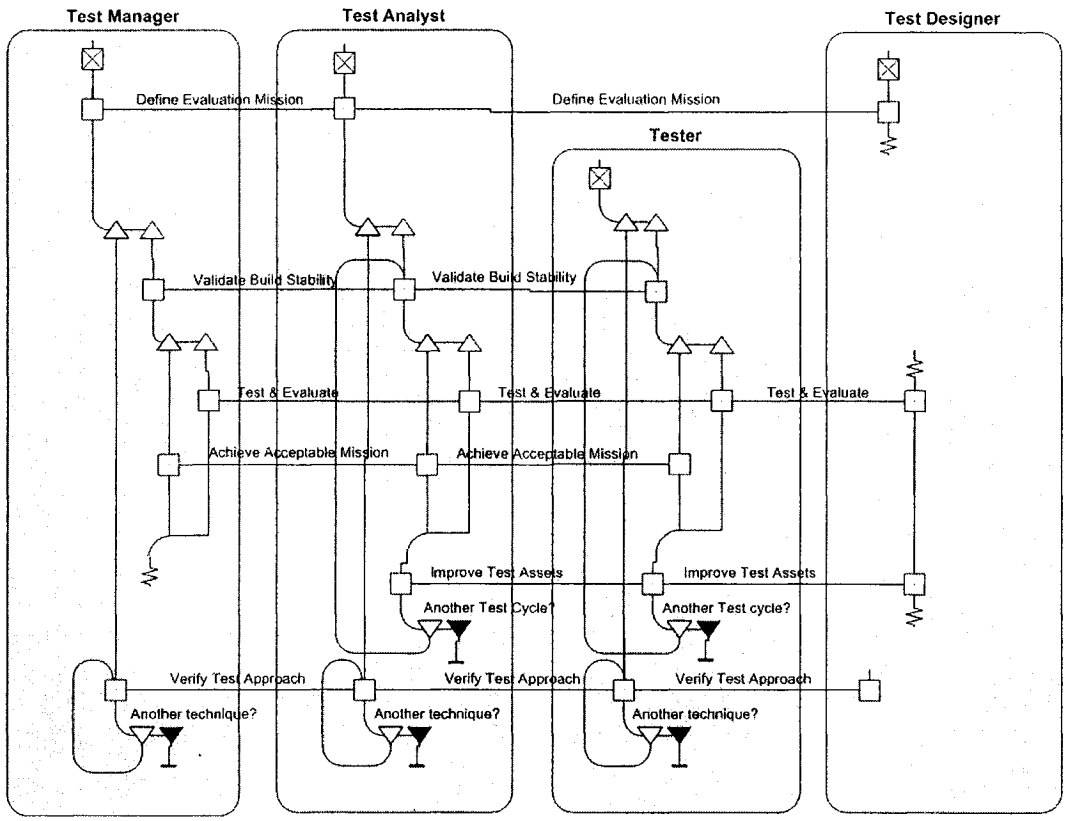


Figure 43: Role Activity Diagram for the *Test* discipline (v. 2)

In this new version, we have three key roles (Test Manager, Test Analyst and Tester), who are responsible for the process logic. The Test designer is a role that acts on demand of one of the other three roles. We chose to make the activities in the Designer role "hanging", because we don't want to keep the Designer role idle, between the time he participates in the interaction "Define Evaluation Mission" and the time he is doing the activity "Test and Evaluate". Instead of waiting during this time, he can do other activities. However, we can link the output state of the activity "Test and Evaluate" to the input state of the activity "Improve Test Assets" (i.e. link the two activities by the same state line), because the process permits it (it is also the case in the role body of the Test Analyst and of the Tester; however, they have to wait for the interaction "Achieve Acceptable Mission" to take place, before they can "Improve Test Assets").

We use a "hanging thread" in the role body of the Test Manager, after completion of the two concurrent threads in which he carries the activities "Test and Evaluate" and respectively "Achieve Acceptable Mission". The reason we don't use the "Stop" symbol for this thread is because we may need to reactivate the thread, if after the activity "Improve Test Assets" (done by Test Analyst and Tester), we find out there is another test cycle, in which case we need to come back and do some of the previous activities over again.

4.2.2.6 Deployment

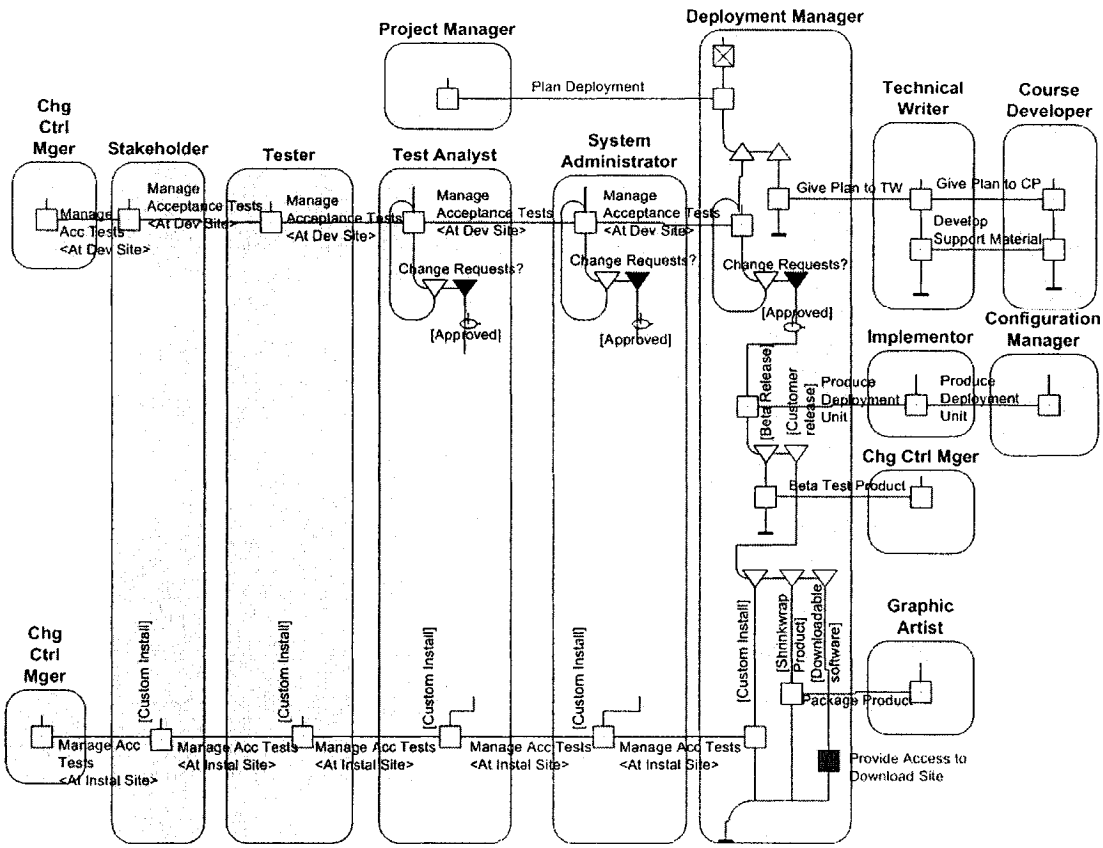


Figure 44: Role Activity Diagram for the *Deployment* discipline (v. 2)

In the first version (see Figure 38, page 70), the Technical Writer and the Course Developer were not linked to the process. They were doing the activity "Develop Support Material" parallel to the process, but we don't know when this activity is initiated. By interacting with the Technical Writer and the Course Developer, the Deployment Manager includes them (and their interaction) to the process. In this new version of the Deployment discipline, we introduce two new roles (Project Manager and Change Control Manager). The reference [3] we used for this version of the model provided more information about the interaction between the different roles involved in the Deployment discipline. As a result, the new version is improved in terms of clarity and information detail.

We can see how important the Deployment Manager role is to the process. He is responsible for the process logic (concurrent or alternative threads), he is responsible for the different decisions taken ("Change Requests") and he initiates the interactions with other roles. The other roles are all acted upon request of the Deployment Manager. The advantage with this diagram is that the process logic is visible inside the Deployment Manager Role.

4.2.2.7 Limitations

The second version of the RADs we have just developed, expresses very well the interactions between the roles, as well as the process logic. We can also express if some roles are principal or secondary, if some activities are "hanging", if there are external events, etc. There is one thing which they don't express yet: the data flow. RUP puts a lot of emphasis on the artifacts produced during activities and interactions. Our model can be complete, if it contains the artifacts produced and consumed by the roles at different steps of each discipline. We constructed a RAD for the flow of the artifacts of the Business Modeling discipline, to see if the present notations that RAD uses for expressing entities flow (as described in section 4.1.8, page 60) are suited to model the RUP artifacts. We obtained the following diagram:

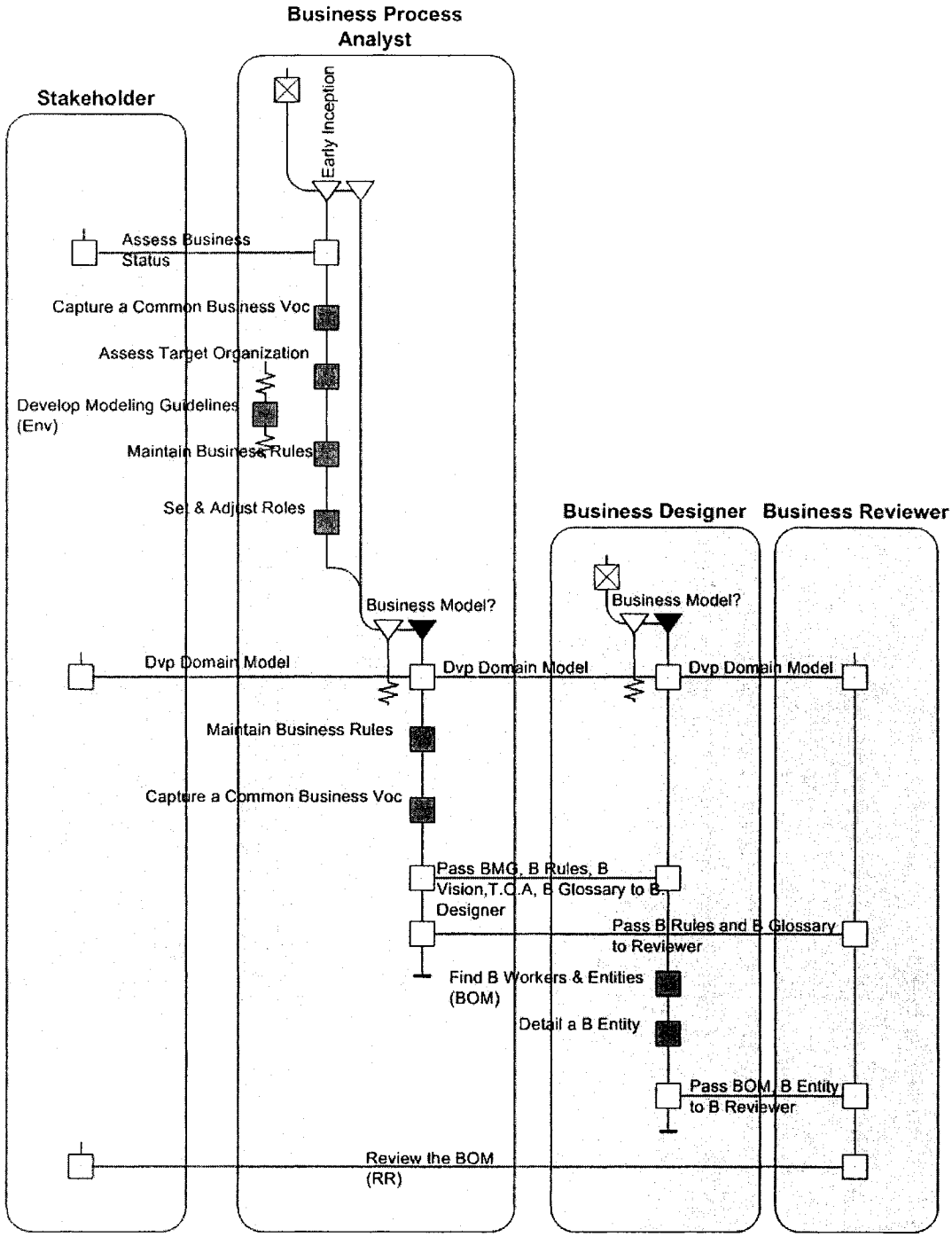
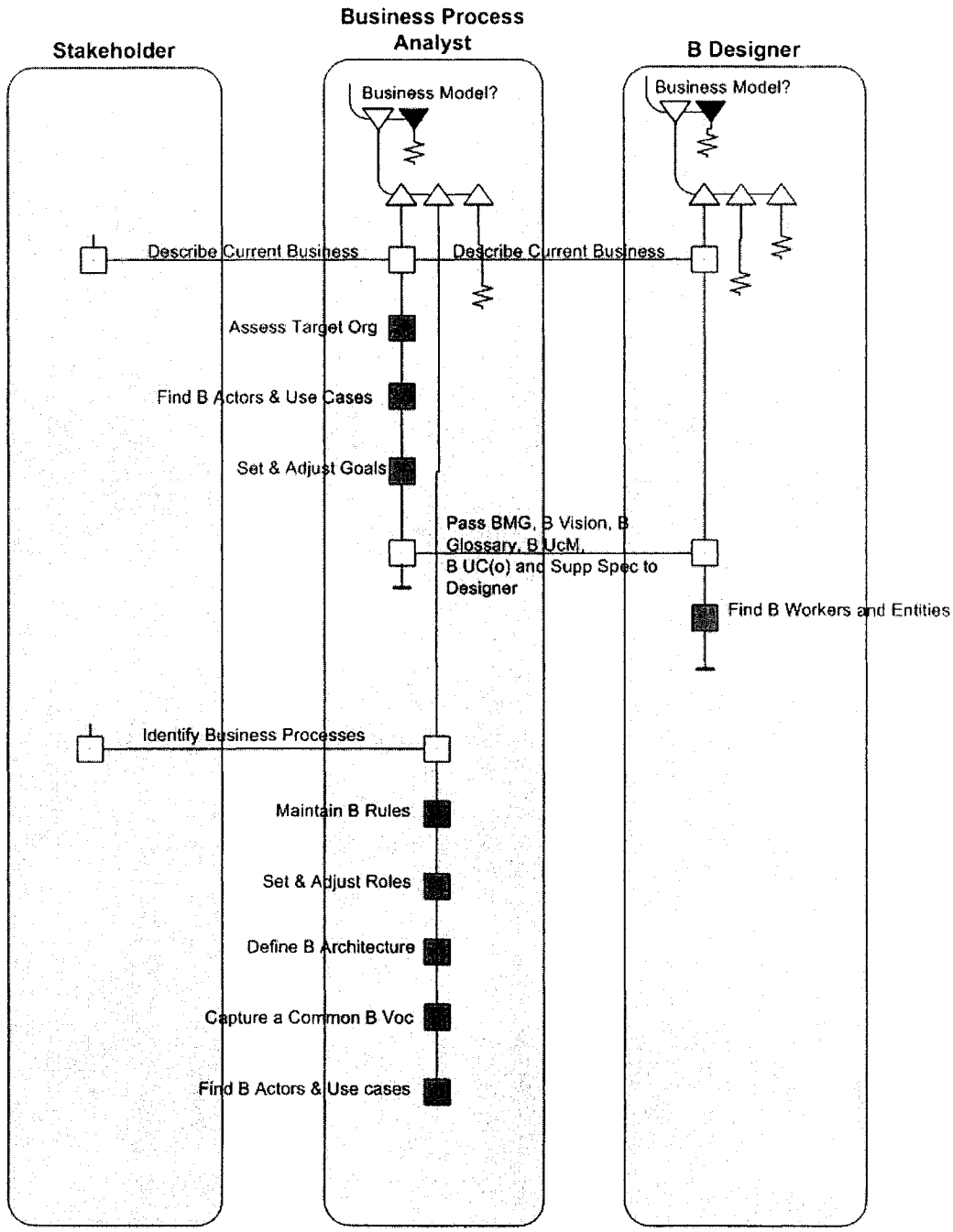
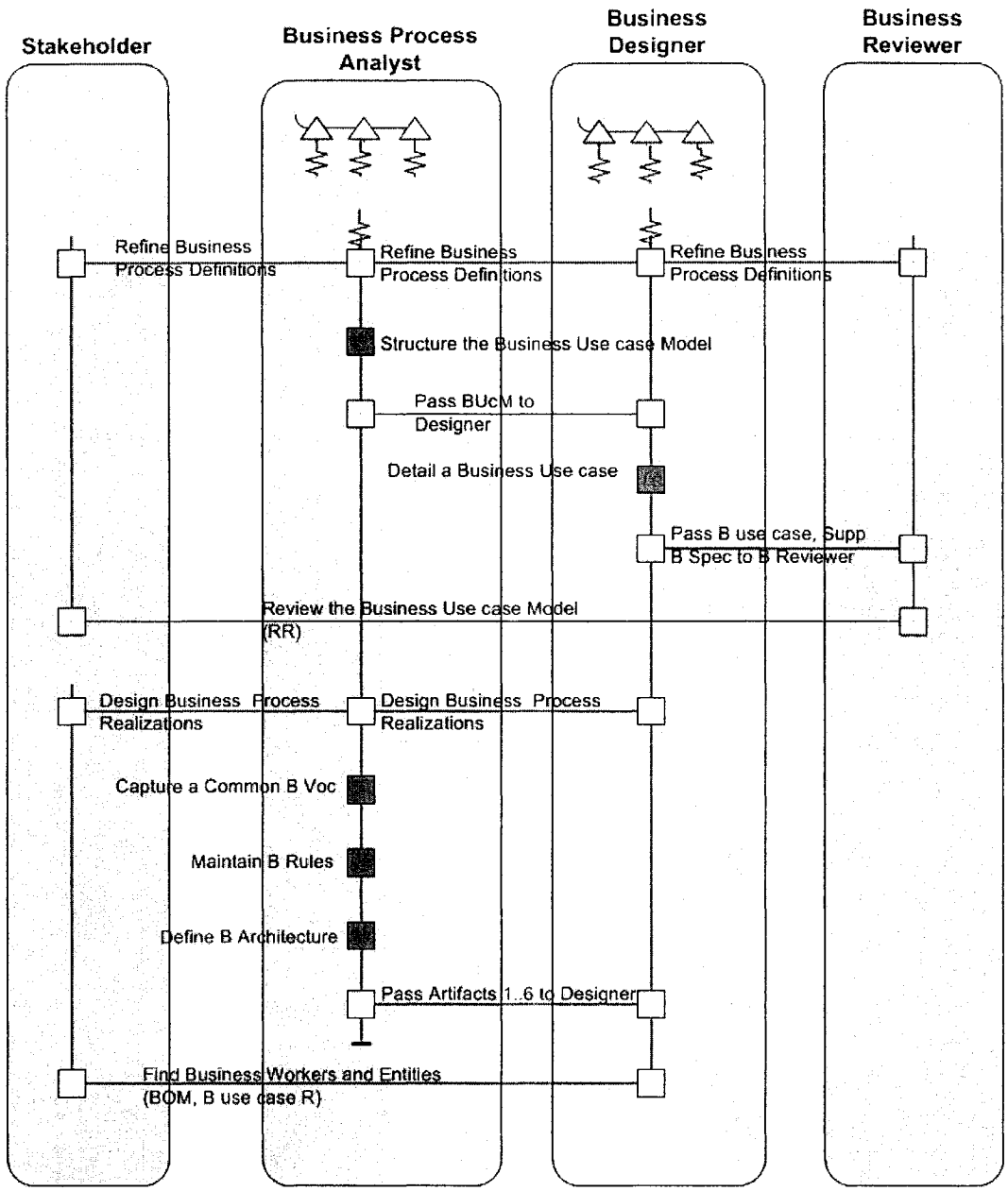


Figure 45: Role Activity Diagram for the *Business Modeling* discipline entities flow



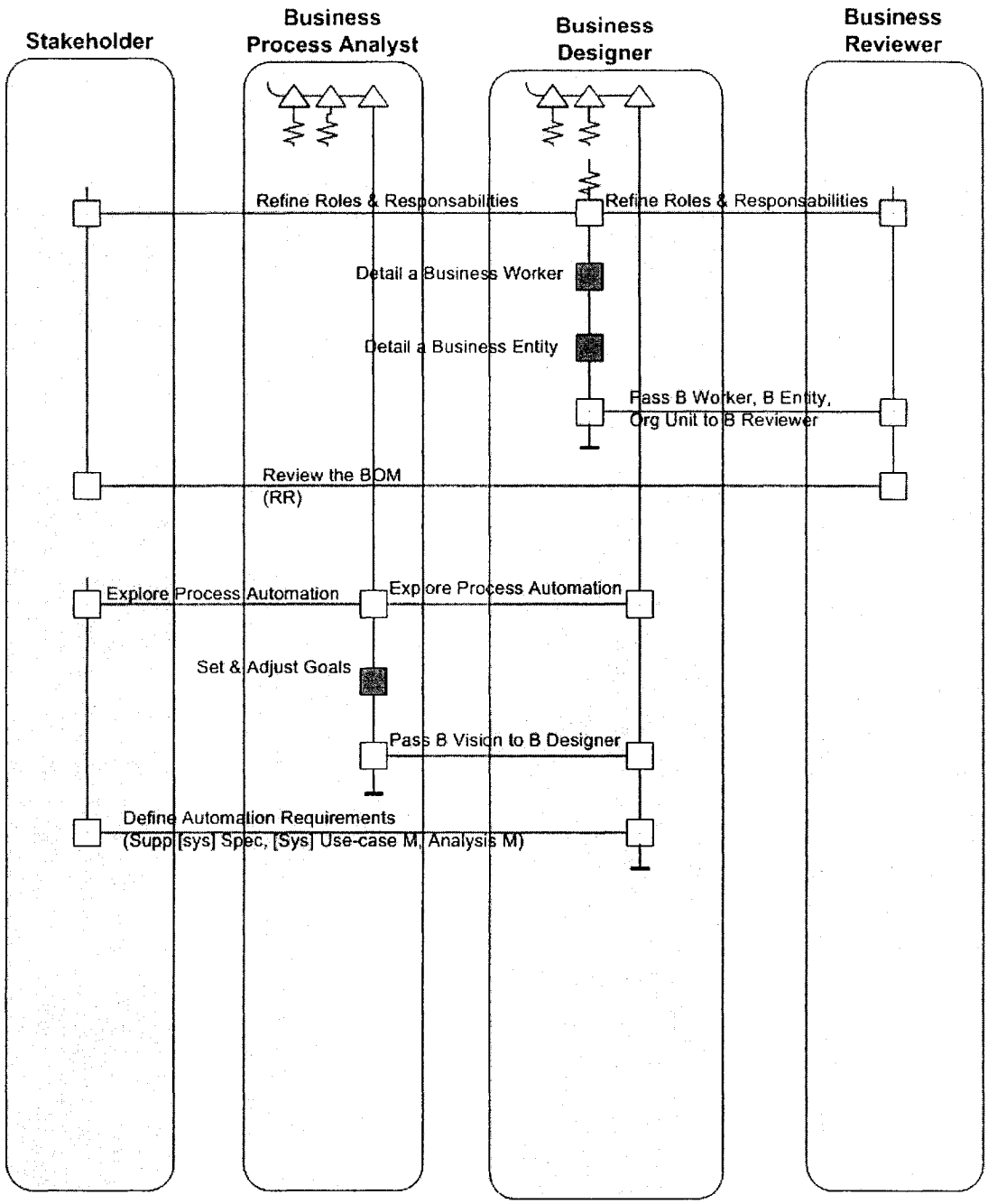
Business Modeling (entities flow), p2

Figure 46: Role Activity diagram for the *Business Modeling* discipline entities flow (contd.)



Business Modeling (entities flow) p3

Figure 47: Role Activity Diagram for the *Business Modeling* discipline entities flow (contd.)



Business Modeling (entities flow) p4

Figure 48: Role Activity Diagram for the *Business Modeling* discipline entities flow (contd.)

After having constructed the RAD to express the entities flow, we came up with several arguments that show that the original RAD diagrams are not suited to express the flow of artifacts.

They have the following limitations:

1) For each input of an activity, we need to refer to the role that produced it (if different from the one that needs it), and pass it through a part interaction (white-box) to the role that needs it. If many artifacts are involved, coming from many disciplines and roles, it would complicate the diagram, or make it very long.

2) For each output, we need to show the activity that produces it (black box). For 5 outputs, we need 5 black boxes connected by the role state. This again makes it very long.

3) If an activity needs inputs that were previously produced by the same role, they will not be specified. It is implied that anything that was produced before can be used after by the same role. But not knowing which artifacts are necessary for the present activity is not helpful.

4) We cannot express the entities flow of the discipline, because RADs don't allow the use of arrows going in and out the part-interactions.

4.2.3 Third version RADs (XRAD)

In order to make a more compact model and to include the artifacts to the discipline, we came up with a very simple solution that makes use of arrows to express the entities flow throughout the process. The new elements used in XRAD representation seem to have different semantics compared to the original RAD version. These differences occur in the way we represent the artifacts flow. In RAD, when an artifact is produced, a black box activity illustrates it, and when an artifact is required by an activity, it has to be passed to the role by an interaction (white box). In XRAD, input artifacts are represented by arrows going in an activity (or part-interaction), and the output artifacts by arrows going out of the activity (or part-interaction).

Our approach uses the idea of a central repository, which contains all the artifacts produced during a discipline. During an activity, a role produces an artifact that is referred to as a number in the repository. An activity also needs inputs, which are artifacts produced by the same role or by other roles in the same discipline, or in other disciplines. This approach, as underlined in the previous section, could be misleading, because we use the notion of "shared artifacts". However, it turns out it is clear enough, because it shows where the artifact is coming from, who produced it, and in which discipline it was created.

There is a repository for each discipline, *Business Modeling* (see appendix A.1.4, Figure 60, page 112), *Requirements* (see appendix A.2.4, Figure 71, page 127), *Analysis and Design* (see appendix A.3.4, Figure 80, page 141), *Implementation* (see appendix A.4.4, Figure 89, page 157), *Test* (see appendix A.5.4, Figure 97, page 167) and *Deployment* (see appendix A.6.4, Figure 106, page 181). However, if we refer to artifacts produced in the support or process disciplines (not covered in the thesis), we refer to their abbreviation (see appendix A.8, Figure 121, page 194). For e.g., IP is Iteration Plan from Project Management, CR is Change Requests from the Configuration and Change Discipline, etc.

While we are modeling the process and the entities flow in a discipline, we may need to refer to artifacts from other core disciplines. We do that by referring to the discipline and attaching the number that corresponds to that artifact in that discipline. For example, in Analysis and Design discipline, we need R7, which corresponds to the artifact number 7 from Requirements, which is Supplementary Specifications. An artifact can have a different status: outlined, updated, refined, etc. All the abbreviations and notations are documented in the legend (see appendix A.8, Figure 121, page 194). We add the artifacts used (input) and produced (output) during the discipline.

Most activities in the discipline are collaborations between roles. Each role does some "micro-activities" (not visible in the main workflow), the result of which are entities needed in the process (e.g. Architecture Document, Guidelines, Plans, Test Results, Scripts, etc). Some of the artifacts produced by one role are needed by another role(s) involved in the interaction. We represent the artifacts needed by a role involved in the collaboration by an arrow going in the part-interaction of the respective role, and annotated with the appropriate numbers or symbols (or both) corresponding to the artifacts. The artifacts produced by the role are represented by an arrow going out the part-interaction of the role, annotated with the respective numbers or symbols for the artifacts.

For example, in Figure 134 (see Appendix A.8, page 207), the Software Architect, the Designer and the Design Reviewer interact to "Analyze Behavior".

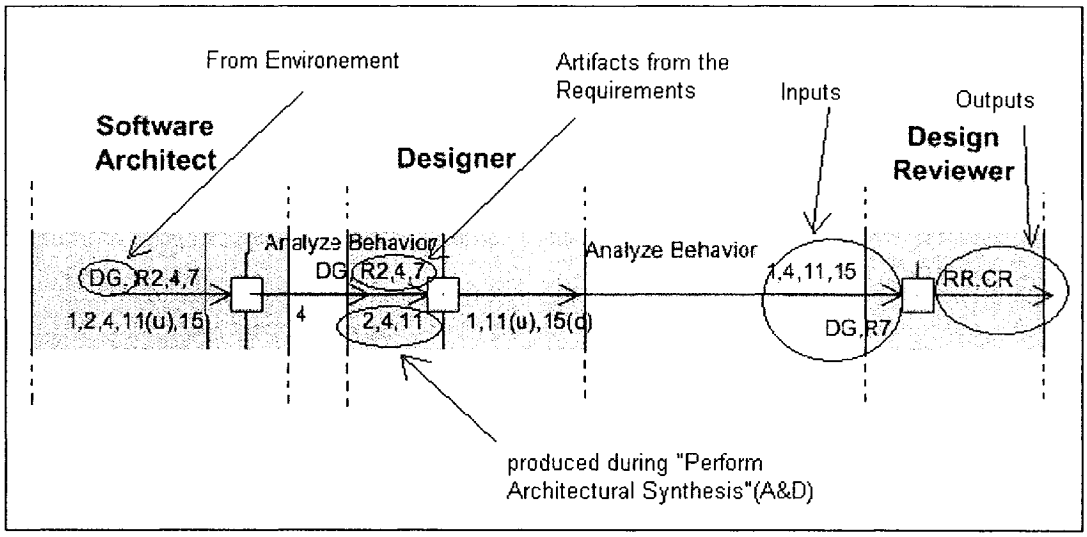


Figure 49: The flow of artifacts at interaction time between the Software Architect, the Designer and the Design Reviewer (excerpt from Figure 134, page 207)

The Designer needs artifacts produced in the *Requirements* discipline (R2 – Glossary, R4 – Use case Model, R7 – Supplementary Specifications), an artifact produced in the *Environment* discipline (DG – Design Guidelines), and artifacts produced by the Software Architect (in the *Analysis and Design* discipline, during the Inception phase and during the "Perform Architecture Synthesis" activity) - 2 (Software Architecture Document), 4 (Design Model), 11 (Use-case Realization).

The designer updates the use-case realizations (11), creates the Analysis Model (1), and details the Analysis Class (15). The designer also passes these three artifacts to the Software Architect and to the Design Reviewer. The Software Architect uses these artifacts, and the ones already mentioned for the Designer role, updates the Design Model (4), which he passes to the Design Reviewer. The Design Reviewer will produce two artifacts: CR (Change Request) and RR (Review Record).

A collaboration is not necessarily concerned with passing artifacts from one role to another involved in the collaboration; sometimes the collaboration involves several roles working together to achieve some "common goal". For example, in Figure 128 (see Appendix A.8, page 201), the Business-Process Analyst and the Business Designer, involved in the collaboration "Design Business Process Realizations", both share resources produced before (in a previous workflow detail) by the Business Process Analyst (Business Glossary, Target Organization Assessment, Business Rules, Business Vision, etc.).

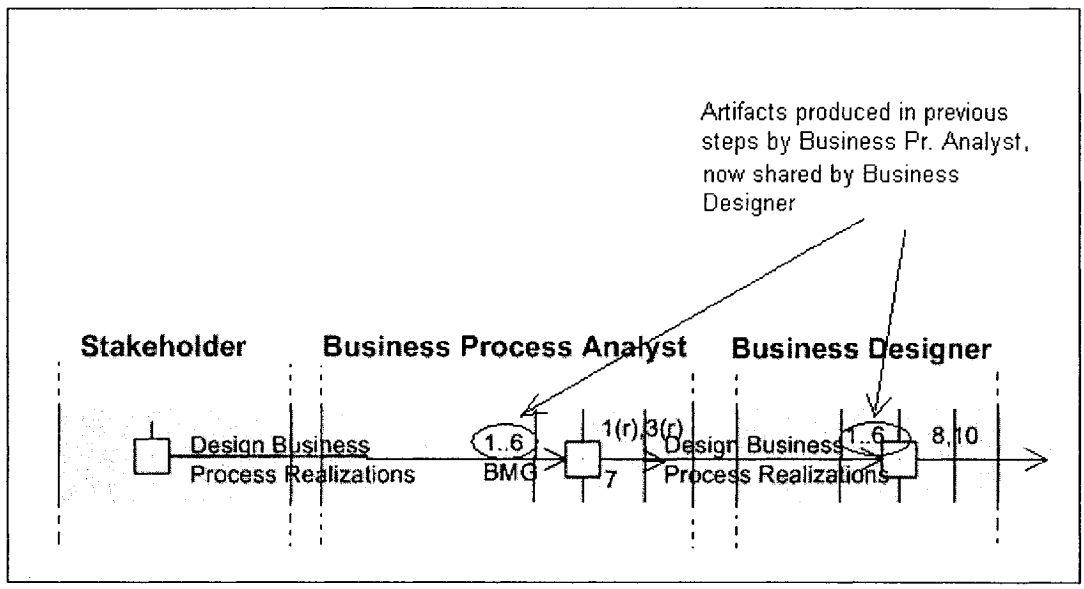


Figure 50: Representing artifacts being shared between roles (excerpt from Figure 128, page 201)

Sometimes, the workflow detail is not a collaboration; it is done by a single role; in this case it is represented by a black box activity. For example, in Figure 137 (see Appendix A.8, page 210), the Integrator is doing the activity "Plan the Integration" by himself.

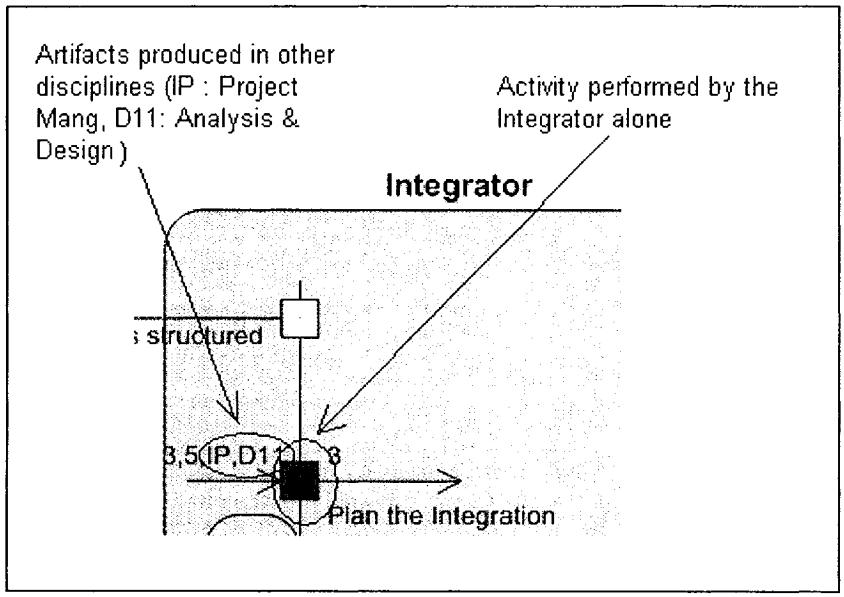


Figure 51: Representing a workflow detail as an activity performed by a single role (excerpt from Figure 137, page 210)

The Integrator uses artifacts produced previously in the Analysis and Design discipline, by Designer role, "Use-Case Realization" (D11), in Project Management discipline, "Iteration Plan" (IP), and the "Implementation Model" (5) structured by the Software Architect at the beginning of the Implementation discipline. He produces the integration plan (3). We have developed the XRADs for all the core disciplines. We present them here.

4.2.3.1 Business Modeling

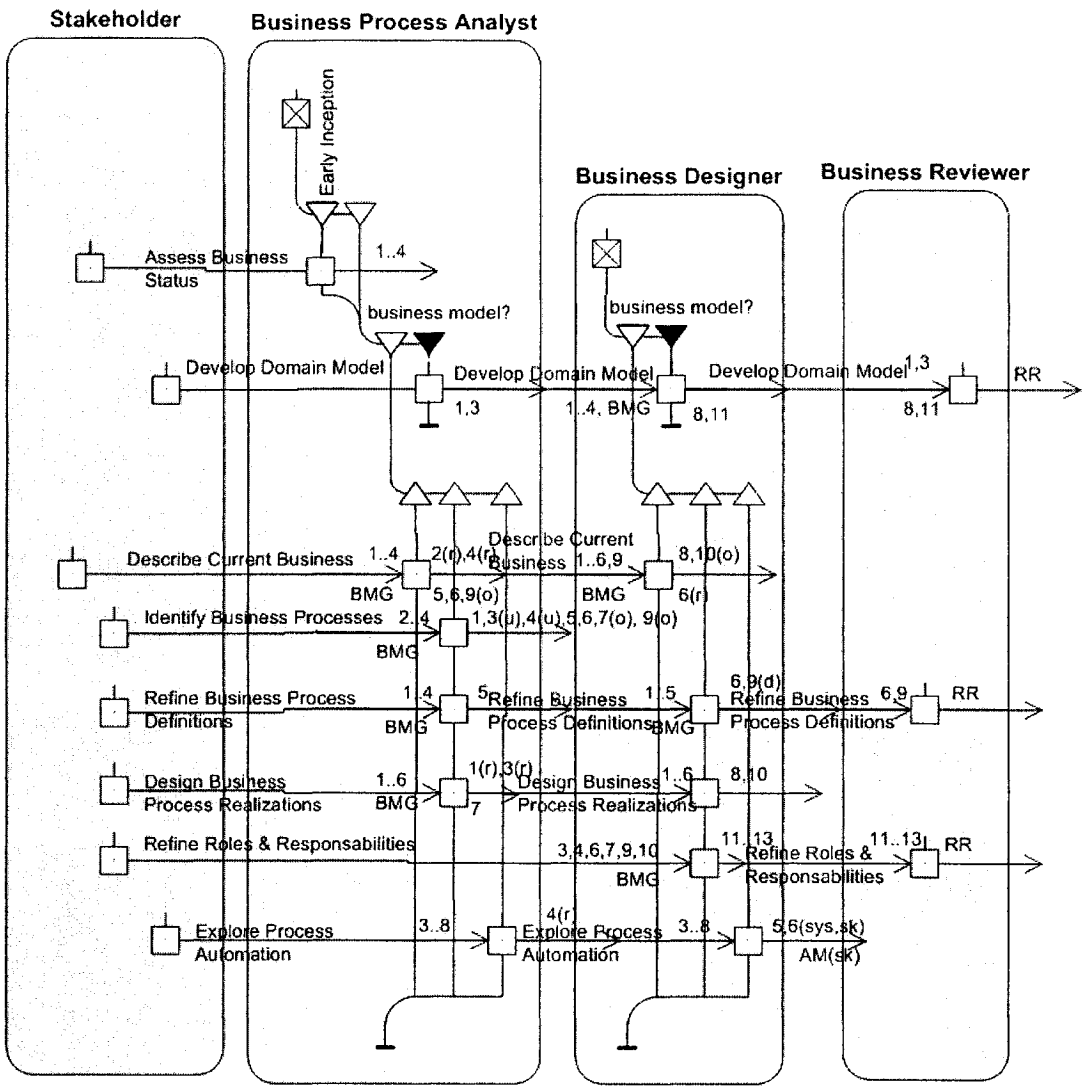


Figure 52: XRAD for the *Business Modeling* discipline entities flow

4.2.3.2 Requirements

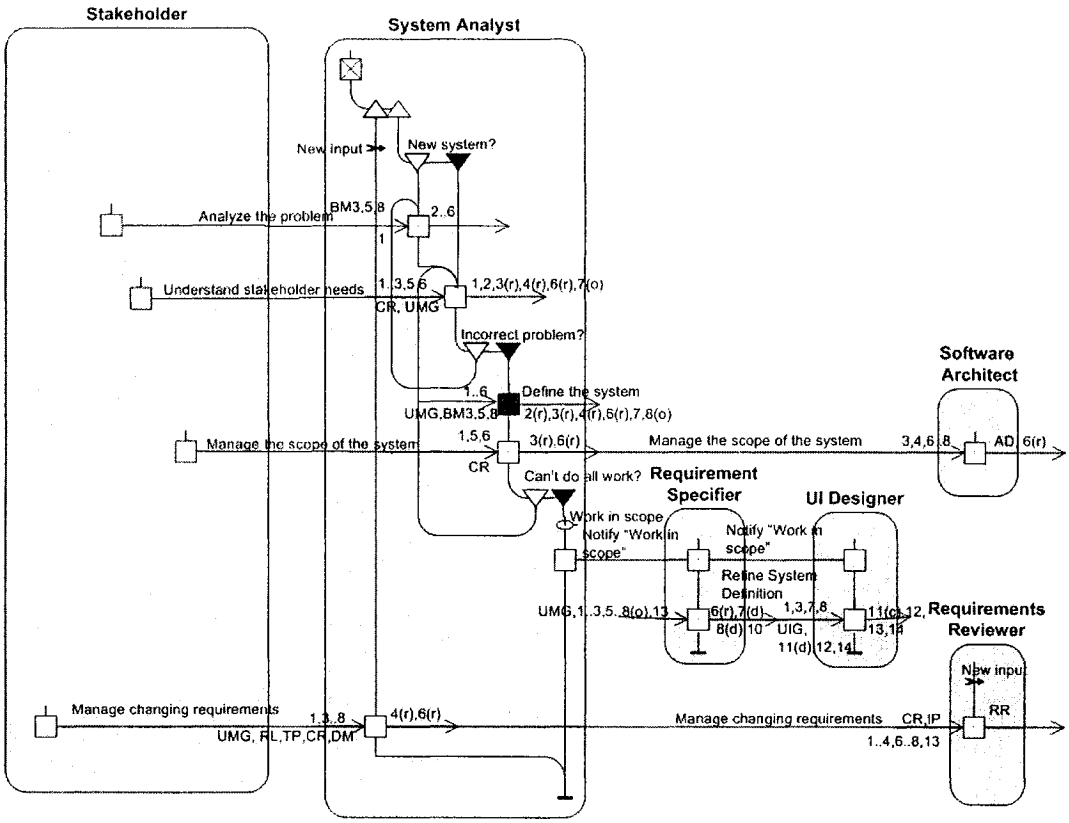


Figure 53: XRAD for the *Requirements* discipline entities flow

4.2.3.3 Analysis and Design

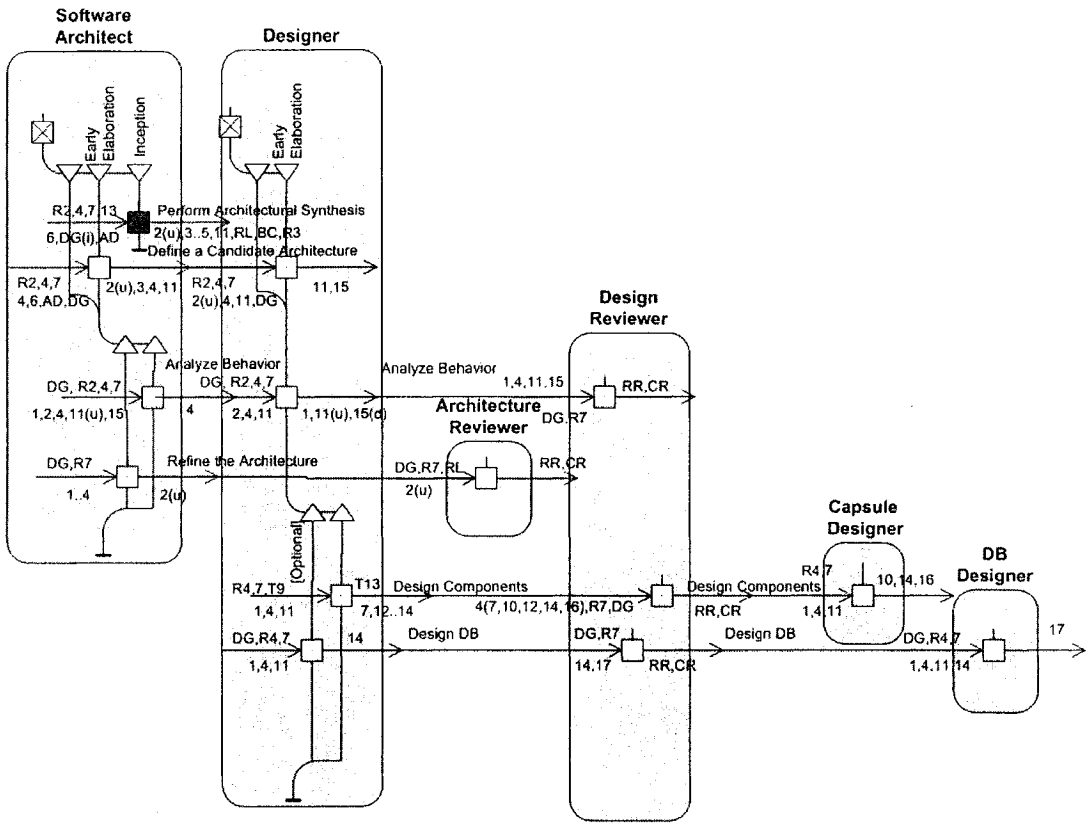


Figure 54: XRAD for the *Analysis and Design* discipline entities flow

4.2.3.4 Implementation

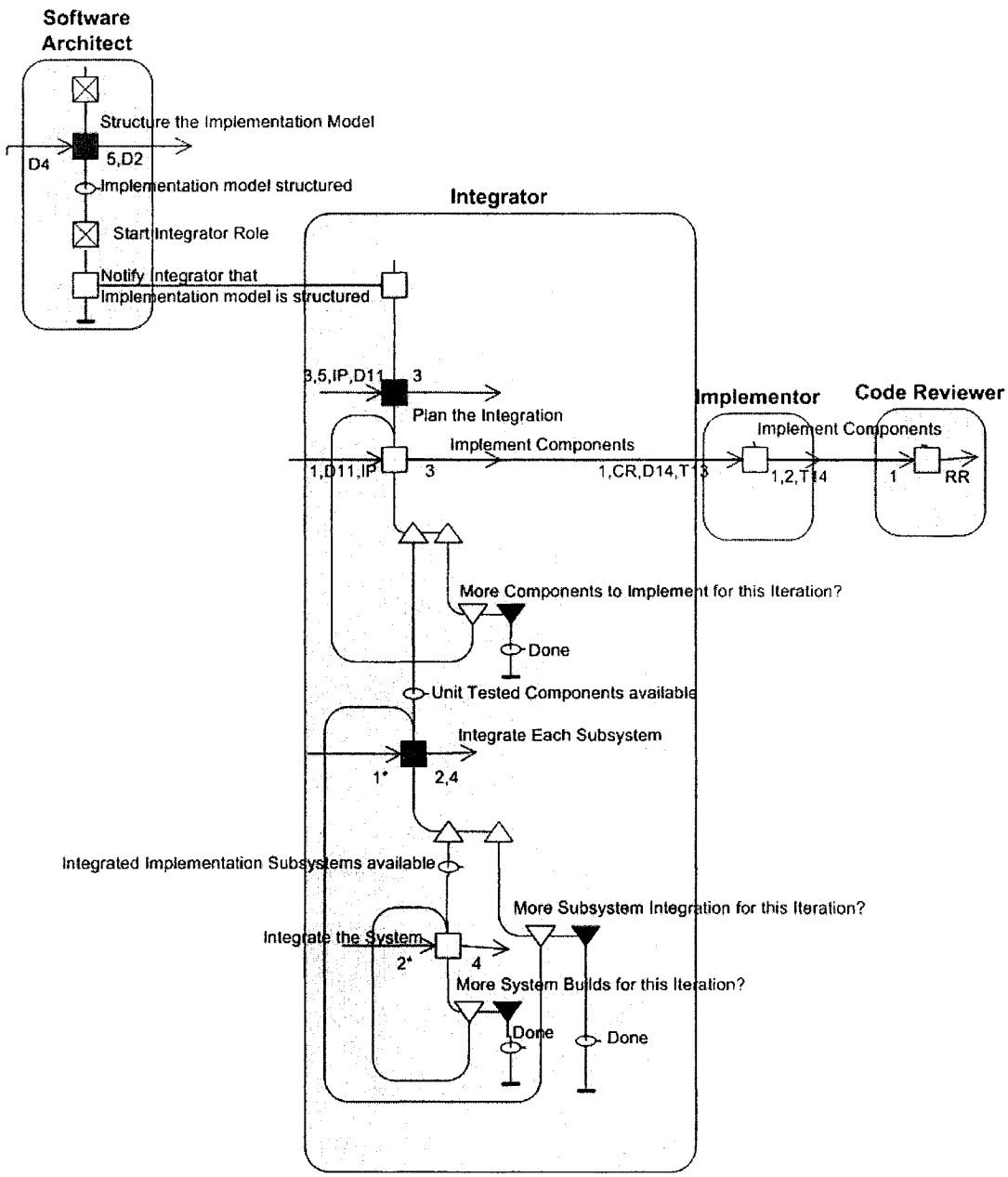


Figure 55: XRAD for the *Implementation* discipline entities flow

4.2.3.5 Test

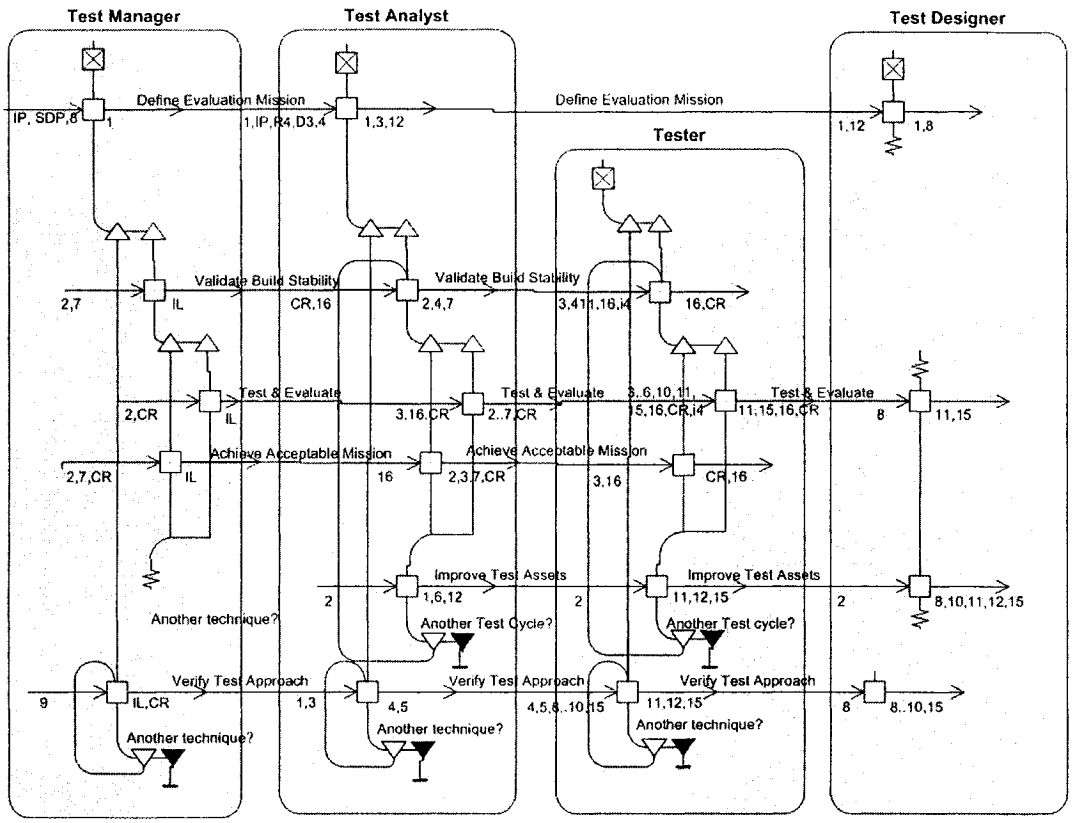


Figure 56: XRAD for the *Test* discipline entities flow

4.2.3.6 Deployment

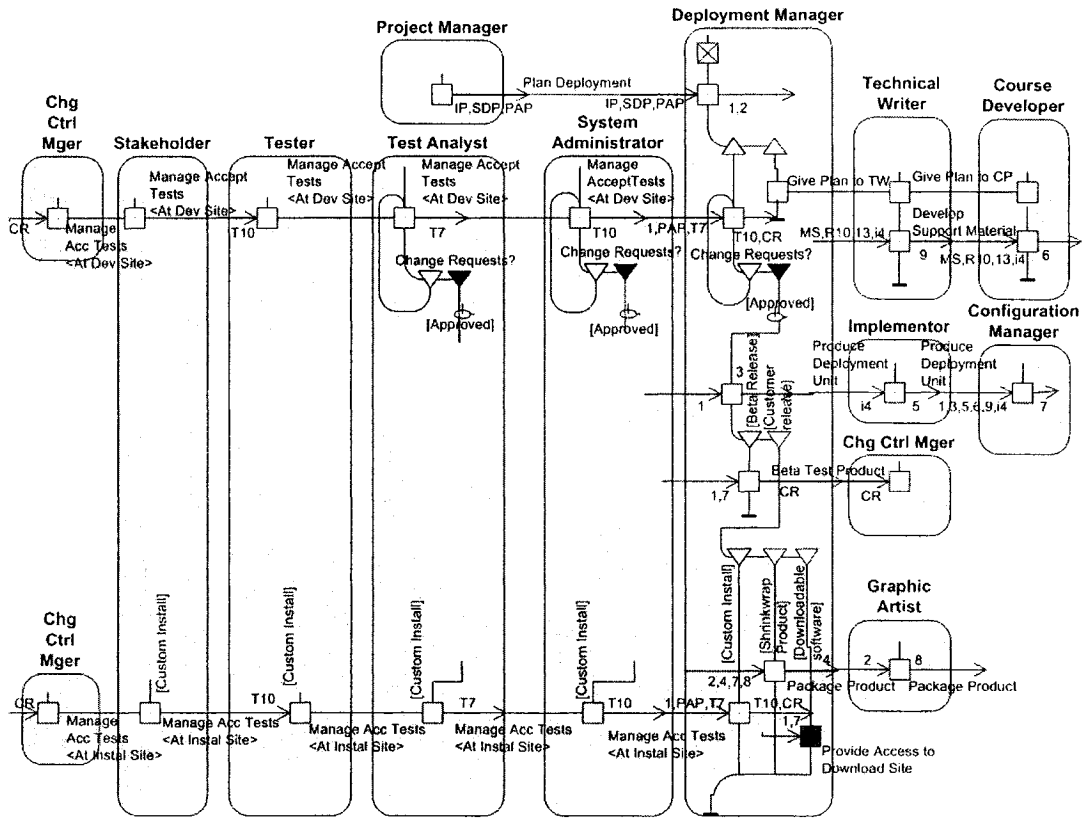


Figure 57: XRAD for the *Deployment* discipline entities flow

4.2.3.7 Benefits and limitations of the XRAD

This version will prove to be useful to all process actors, from the process designers, who create the drafts of the processes, manage and monitor those processes, to the actual developers, responsible for implementing the technology that will perform those processes. Based on this version, the process designers have a useful tool to easily develop simple diagrams that will look familiar to most process actors. The introduction of the arrows to express the input and output entities is familiar to most modelers in software engineering. The users of the XRAD can easily recognize the basic types of elements and understand the diagram.

The purpose of this new version is to create a simple mechanism for creating process models, capable of handling the high complexity of processes. This version represents an overall view of the process that could be split into many layers. A good approach can be to represent each layer by a separate view, illustrated by new and less complex XRAD diagram. This approach avoids the case where the diagrams become very dense and less readable, due to the complexity of the process represented by a single view.

There is no formal method to generate different views of the XRAD diagrams automatically, for each iteration and for each phase and taking in consideration the nature of the project. We could generate them manually, but we don't propose here a formal model to generate them automatically (see Chapter 6 : Future Work).

4.3 Summary

In this Chapter, we proposed the Role Activity Diagrams (RAD) as an alternative for modeling the RUP disciplines. These diagrams are better because they allow us to represent the roles and the artifacts.

We showed how the RAD diagrams are built, what are the different symbols they use to express the logic of the process (the concurrency and the alternative courses of action), the occurrence of an external event, the state the process is in, the instantiations of roles, the activities of roles and interactions between roles. We also showed how RAD diagrams are presently used to capture the resource consumption and production. We saw this was not optimal, and we proposed XRAD diagrams, which use arrows to express the entities flow throughout the process.

We also showed how we evolved the RAD diagrams from an old version to a new one, by using a more complete reference, which gives more detail in terms of the roles involved in the process and the resources used and produced.

Chapter 5 : Conclusion

We have thoroughly studied the RUP process model that is traditionally using various forms of UML activity diagrams for its representation. We have identified deficiencies in this representation of the process. Based on these deficiencies, we have developed a new version of the RUP process model representation for the *business modeling*, *requirement*, *analysis and design*, *implementation*, *test*, and *deployment* discipline diagrams using RAD. We then have analyzed this alternate representation and extended the RAD notation to include artifacts flow. We have named this extension of RAD, XRAD. Then we used XRAD to represent the above mentioned disciplines of the RUP. This version will be particularly useful to process designers, who create the drafts of the process, but also to actual developers who implement the process. It is a tool to create simple diagrams that will look familiar to all process actors.

From both theoretical and practical perspectives, the following observations can be made:

- The Rational Unified Process (RUP) can be described using role activity diagrams (RAD).
- In our research, we proved that with the existing activity diagrams, there exist some limits, and therefore the necessity to improve their structures. In fact, this was the main idea that led us to develop new versions of diagrams such as the first and second RAD versions and the XRAD version as a combination of RAD and RUP.

- The first and second RAD versions still have limits, but their structures reveal real practical interest and a substantial improvement compared to the diagrams designed in the literature using activity diagrams.
- The last XRAD version eliminates almost all these limits, but still remain an open field for future work.
- The RUP methodology is an iterative process that identifies four phases of any software development project. From our point of view, an increasing number of iterations in each phase allows better validation and verification of the results, better distribution of the feedback from client and a more focused risk analysis and risk reduction. In our research, we limit this number at one single iteration. It would be a good opportunity to orient the future work in the direction of increasing the number of iterations explicitly in the process description. Note that this limitation is part of the original RUP representation, and not an additional limitation raised by our RAD and XRAD process modeling.
- Our proposed diagramming technique for the RUP allows the unified description of the dependencies between activities, roles, and artifacts using the same diagrams, which is not the case for the activity diagrams traditionally used in the literature for the description of the RUP.

Chapter 6 : Future Work

A software development process is composed of actors, artifacts produced by actors, and activities undertaken by actors to produce these artifacts. Process modeling involves the modeling of all these components of the process, as well as all the relationships between these components, thus defining a *process meta-model* enabling the modeling of software processes. SPEM is the meta-model used to represent the RUP model. Such meta-models enable a formal process definition enabling process enactment, validation, verification, and improvement, which are the goals of the research project in which this thesis stands.

Process models, as represented by activity diagrams, RAD, and, most particularly XRAD, can be modeled as state machines or dataflow diagrams, where all activities are data filters, consume resources, and exchange data in the form of process artifacts. All these process artifacts are produced in different versions, depending on which project phase or iteration they are being produced. Some activities and artifacts are to be adapted to different situations. Some document templates come in different versions depending on the nature of the project in which they are used. Some activities in a process can take different forms depending on the project's application area or the technical maturity level of the actors.

We can produce a formal model which can generate different views of an XRAD (for a different role, artifact, iteration, phase, etc.). This model will be used for the verification and enactment of the process.

Thus, process adaptation is generally made according to the *context* of application of the process. All the different variants of activities and artifacts can then be modeled using a multidimensional context-driven model. The Lucid family of intentional programming languages [30] enables the representation of state machines, dataflow diagrams, in a context-driven manner and using context calculus. The GIPSY project aims at the development of compiler and execution engine components for Lucid language variants. The development of a Lucid-based context-driven process meta-model in the GIPSY ([28]-[32]) includes the following tasks:

- Development of a variant of Lucid suitable for process modeling, i.e. the declaration of process workflows and their elements.
- Development of a software component for the graphical representation of RAD and XRAD, as well as a translator to translate such graphs into their Lucid counterpart.
- Development of an execution engine adapted to the verification and enactment of process models defined in this variant of Lucid.
- Development of a context-driven version control repository system enabling the storage of versioned process artifacts.

Chapter 7 : References

[1] Ivar Jacobson, Grady Booch, James Rumbaugh, The Unified Software Development Process, *the complete guide to the Unified Process from the original designers*, Rational Software Corporation, US, 1999.

[2] Philippe Kruchten, The Rational Unified Process, an Introduction, 3rd edition, US, 2003.

[3] http://process-up.ts.mah.se/RUP/RationalUnifiedProcess/process/workflow/ovu_core.htm

[4] Martyn A. Ould, Business Processes, *Modelling and Analysis for Re-Engineering and Improvement*, UK, 2003.

[5] Stefan Bergström, Lotta Råberg, Adopting the Rational Unified Process, Success with the RUP, US, 2003

[6] Maria Ericsson, *Activity diagram: What it is and how to use*, http://sunset.usc.edu/classes/cs577a_2000/papers/ActivitydiagramsforRose Architect.pdf

[7] Ludovic Arbelet, *UML n'est pas encore entré dans les moeurs*, in 01 Informatique, 11/10/2002, http://www.01net.com/article/195306.html

[8] Thierry Jacquot, *Des approches modernes dérivées du RAD*, in 01 Informatique, 06/02/2004, http://www.01net.com/article/232427.html

[9] Philippe Billard, *L'individu au Coeur des projets de développement logiciel*, in 01 Informatique, http://www.01net.com/article/176821.html

[10] Stephen A. White, *Introduction to BPMN*, http://www.bpmi.org

[11] Odeh, M., Beeson, I., Green, S., and Sa, J., *Modeling Processes using RAD and UML Activity Diagrams: an Exploratory Study*, ACIT Conference, Doha Qatar, International Arab Conference on IT, Doha Qatar, Dec. 16th - 19th, 2002

[12] Holt A W, Ramsey H R and Grimes J D, *Coordination System Technology as the basis for a programming environment*, Electrical Communication 57-4, 1983, pp 308-314

[13] Ould M A, Business Processes – *Modeling and analysis for re-engineering and improvement*, John Wiley & Sons, Chichester, 1995

[14] *Software Process Engineering Metamodel* , An Adopted Specification of the Object Management Group, Inc., November 2002, version 1.0, http://www.omg.org/docs/formal/02-11-14.pdf

[15] Peter F. Drucker, Management: Tasks, Responsabilities, Practices, New York: Harper & Row, 1973

[16] Tom Gilb, Principles of Software Engineering Management. Harlow, UK: Addison-Wesley, 1988

[17] Barry W. Boehm, "*A Spiral Model of Software Development and Enhancement*", IEEE Computer, May 1988

[18] Capers Jones, *Assessment and Control of Software Risks*, Upper Saddle River, NJ: Prentice-Hall, 1993

[19] Gary Pollice, *Using the IBM Rational Unified Process for Small Projects: Expanding Upon eXtreme Programming*, IBM Rational Software, http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/tp183.pdf

[20] Martin, Booch and Newkirk, *The Process*, http://www.objectmentor.com/ressources/articles/RUPvsXP.pdf

[21] Sam Courtney, Senior Software Specialist, IBM Rational, *Adopting and implementing RUP for a maintenance development cycle*, October 2004, http://www-128.ibm.com/developerworks/rational/library/nov04/courtney/index.html

[22] Fahmy, H.M.A., *Petri nets analysis using small computers: piecewise method*, Microcomputer Applications, ISMM International Conference, Los Angeles, USA, December 14 - 16, 1989, pages 106-108., http://www.informatik.uni-hamburg.de/TGI/pnbib/

[23] Tittus, M., *Petri net models in batch control*, Mathematical and Computer Modeling of Dynamical Systems, Vol. 5, No. 2, pages 113-132. 1999, http://www.informatik.uni-hamburg.de/TGI/pnbib/

[24] Tiplea, F.L., *On Normalization of Petri Nets*, Proc. of the 11th Romanian Symposium on Computer Science ROSYCS'98, Iasi (Romania). May 1998, http://www.informatik.uni-hamburg.de/TGI/pnbib/

[25] Scott W. Ambler, The Object Primer 3rd Edition, *Agile Model Driven Development with UML 2*, Cambridge University press, 2004

[26] Mentoring for WayPointer, RUP and UML, http://www.jaczone.com/services/mentoring/

[27] Mohamed Fayad and Marshall P. Cline, *Aspects of Software Adaptability*, http://portal.acm.org/citation.cfm?id=236156.236170

[28] Joey Paquet , Aihua Wu, *Towards a Framework for the General Intensional Programming Compiler in the GIPSY*, Proceedings of OOPSLA 2004, Vancouver, Canada, October 24-28, 2004

[29] Ai Hua Wu, Joey Paquet and Peter Grogono, *Design of a compiler framework in the GIPSY system in Parallel and Distributed Computing and Systems* - PDCS 2003, Marina Del Rey, California, USA, 2003.

[30] Bo Lu, Peter Grogono and Joey Paquet, *Distributed execution of intensional multidimensional programming languages in Parallel and Distributed Computing and Systems* - PDCS 2003, Marina Del Rey, California, USA, 2003.

[31] Joey Paquet and Peter Kropf, *The GIPSY Architecture In Distributed Computing on the Web*, Proceedings of the Third International Workshop, DCW2000, Lecture Notes in Computer Science, Vol. 1830, Springer, 2000.

[32] Yi Min Ding, *Bi-directional translation between dataflow graphs and Lucid programs in the GIPSY environment*, Master's Thesis, Computer Science Department, Concordia University, Montreal, Canada, 2004.

[33] R. A. Snowdon, *Overview of Process Modeling*, http://www.cs.man.ac.uk/ipg/Docs/pmover.html

[34] M. Dowson and C. Fernström, *Towards Requirements for Enactment Mechanisms*, Software Process Technology, Third European Workshop, EWSPT '94, Villard de Lans, France, February 1994, edited by Brian Warboys. Springer Verlag LNCS 772, 1994

[35] *Proceedings of the Software Process Workshop*, Egham, Surrey, UK, February 1984, edited by Colin Potts, IEEE Press.

[36] Terry Bollinger, Facts and Fantasies, A Review of Two Books, http://www.computer.org/software/Bookshelf-samples.pdf

[37] Kendall Scott, *The Unified Process Explained*, Addison-Wesley, Boston, 2002

Appendix A : RUP Workflows by Discipline

The contents of this appendix are based mainly on two references: [2] and [3].

A.1 Business Modeling

A.1.1 Business Modeling Workflow

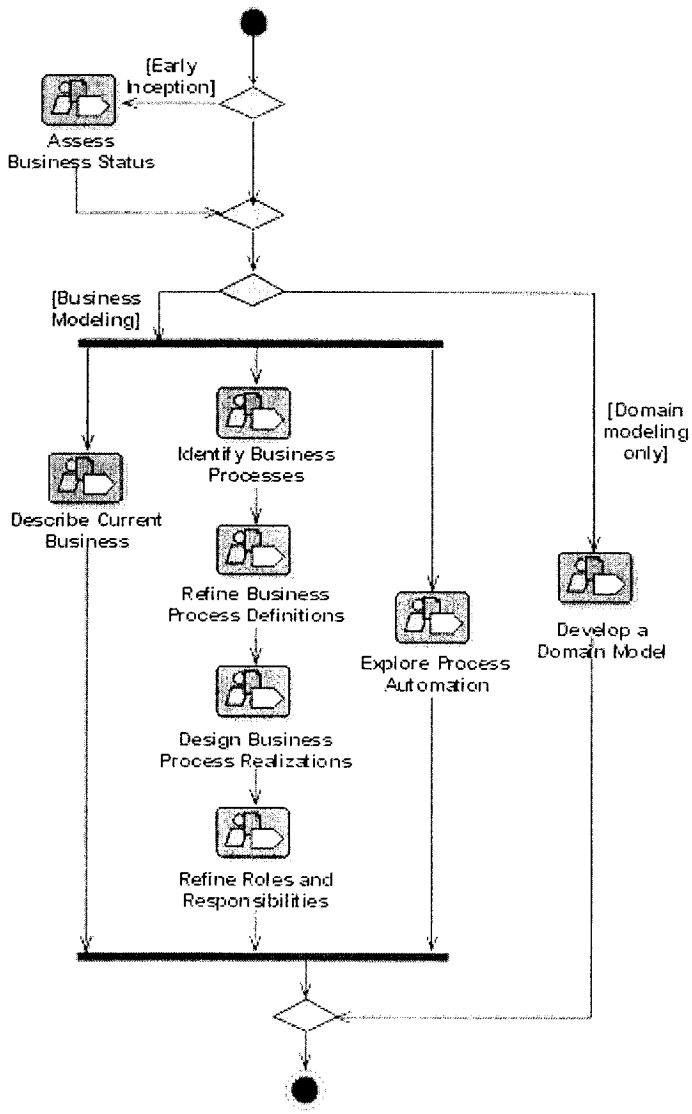


Figure 58: Activity diagram for the *Business Modeling* discipline

A.1.2 Business Modeling Workers

Business-Process Analyst. He leads and coordinates business use-case modeling by outlining and delimiting the organization being modeled. He establishes the vision of the new business, captures business goals and determines which business actors and use cases exist and how they interact.

Business Designer. He details the specification of a part of the organization by describing one or several business use cases. He or she determines the business workers and business entities needed to realize a business use case, how they work together to achieve the use case realization. He defines the responsibilities, operations, attributes, and relationships of one or several business workers and business entities.

Business-Model Reviewer. He participates in formal reviews of the business use-case model and business object model.

Stakeholders. They represent various parts of the organization and provide input and review.

A.1.3 Business Modeling Activities

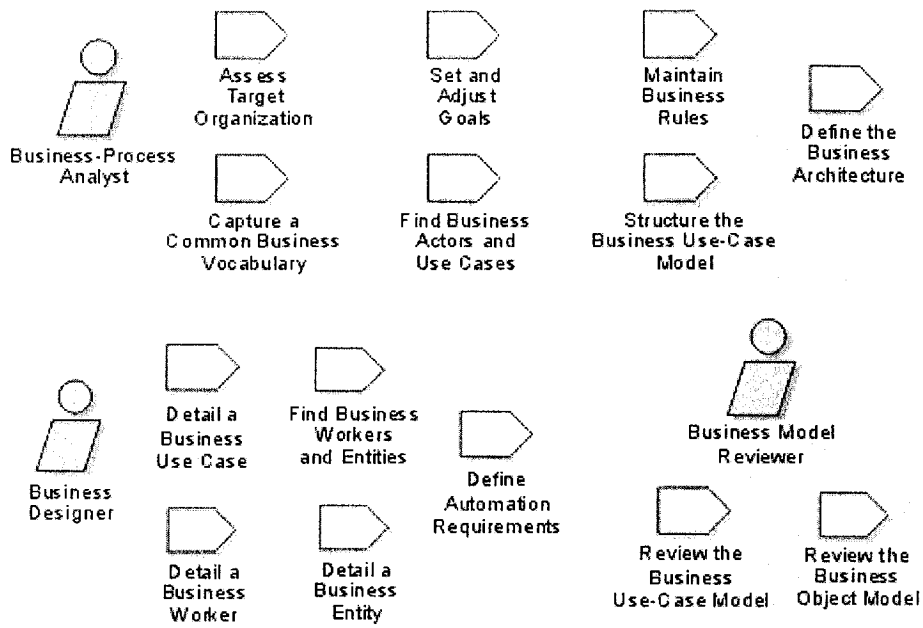


Figure 59: Activities by worker in the *Business Modeling* discipline

A.1.3.1 Business Process Analyst Activities

Assess Target Organization. Its purpose is to describe the current status of the organization in which the application is to be deployed, in terms of its current process, tools, people's competencies, people's attitudes, customers, competitors, technical trends, problems, and improvement areas, to motivate why the target organization must be engineered and to identify stakeholders to the business modeling effort.

Capture a Common Business Vocabulary. Its purpose is to define a common vocabulary that can be used in all textual descriptions of the business, especially in descriptions of business use cases.

Set and Adjust Goals. Its purpose is to define the boundaries of the business modeling effort, to develop a vision of the future target organization, to gain agreement on potential improvements and new goals of the target organization and to describe primary objectives of the target organization.

Maintain Business Rules. Its purpose is to determine what business rules to consider in the project and to give the business rules detailed definitions.

Find Business Actors and Use cases. Its purpose is to outline the processes in the business, to define the boundaries of the business to be modeled, to define who and what will interact with the business, to create diagrams of the business use-case model, to develop a survey of the business use-case model.

Structure the Business Use case Model. Its purpose is to extract behavior in business use cases that need to be considered as abstract business use cases. Examples of such behavior are common behavior, optional behavior, and behavior that is to be developed in later iterations, to find new abstract business actors that define roles that are shared by several business actors.

Define the Business Architecture. Its purpose is to define an architecture for the business, to define the business patterns, key mechanisms and modeling conventions for the business.

A.1.3.2 Business Designer Activities

Detail a Business Use Case. Its purpose is to describe the business use case's workflow in detail and to describe the business use case's workflow so that the customer, users, and stakeholders can understand it.

Find Business Workers and Entities. Its purpose is to identify all "roles" and "things" in the business and to describe how the business use-case realizations are performed by business workers and business entities.

Detail a Business Worker. Its purpose is to detail the responsibilities of a business worker.

Detail a Business Entity. Its purpose is to detail the definition of a business entity.

Define Automation Requirements. Its purpose is to understand how new technologies can be used to make the target organization more effective, to determine level of automation in the target organization and to derive system requirements from the business modeling artifacts.

A.1.3.3 Business-Model Reviewer Activities

Review the Business Use-case Model. Its purpose is to formally verify that the results of business use-case modeling conform to the stakeholders' views of the business.

Review the Business Object Model. Its purpose is to formally verify that the results of business object modeling conform to the stakeholders' views of the business.

A.1.4 Business Modeling Artifacts

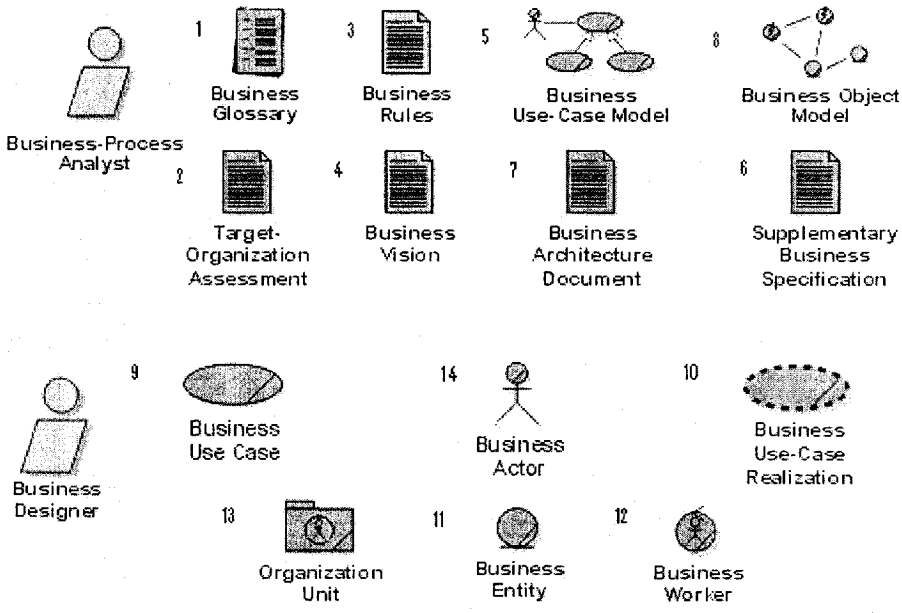


Figure 60: Artifacts produced in the *Business Modeling* discipline

A.1.4.1 Business Process Analyst Artifacts

Business Glossary. It defines important terms used in the business modeling portion of the project.

Business Rules. They are declarations of policy or conditions that must be satisfied.

Business Vision. It defines the set of goals and objectives at which the business modeling effort is aimed.

Target-Organization Assessment. It describes the current status of the organization in which the system is to be deployed. The description is in terms of

current processes, tools, peoples' competencies, peoples' attitudes, customers, competitors, technical trends, problems, and improvement areas.

Business Architecture Document. It provides a comprehensive overview of the business, using a number of different architectural views to depict different aspects of the business.

Business Use case Model. The Business Use-Case Model is a model of the business intended functions. It is used as an essential input to identify roles and deliverables in the organization.

Business Object Model. The business object model is an object model describing the realization of business use cases.

Supplementary Business Specification. This document presents any necessary definitions of the business not included in the Business Use-Case Model or the Business Object Model.

A.1.4.2 Business Designer Artifacts

Business Use case. It defines a set of business use-case instances, where each instance is a sequence of actions a business performs that yields an observable result of value to a particular business actor.

Business Actor. It represents a role played in relation to the business by someone or something in the business environment.

Business Use case Realization. Describes how a particular business use case is realized within the business object model, in terms of collaborating objects (instances of business workers and business entities).

Organization Unit. It is a collection of business workers, business entities, relationships, business use-case realizations, diagrams, and other organization units. It is used to structure the business object model by dividing it into smaller parts.

Business Entity. It's a class that is passive; that is, it does not initiate interactions on its own. A business entity object may participate in many different business use-case realizations and usually outlives any single interaction. In business modeling, business entities represent objects that business workers access, inspect, manipulate, produce, and so on. Business entity objects provide the basis for sharing among business workers participating in different business use-case realizations.

Business Worker. It's a class that represents an abstraction of a human that acts within the system. A business worker interacts with other business workers and manipulates business entities while participating in business use-case realizations.

A.1.4.3 Business Model Reviewer Artifacts

Review Record. It is a form document that is filled out for each review. It is created as a control document to manage the execution of the review of project artifacts. It is issued to the participants in the review to initiate the review process, and is used to capture the results and any action items arising from the review meeting. It forms an auditable record of the review and its conclusions.

A.1.5 Business Modeling Details

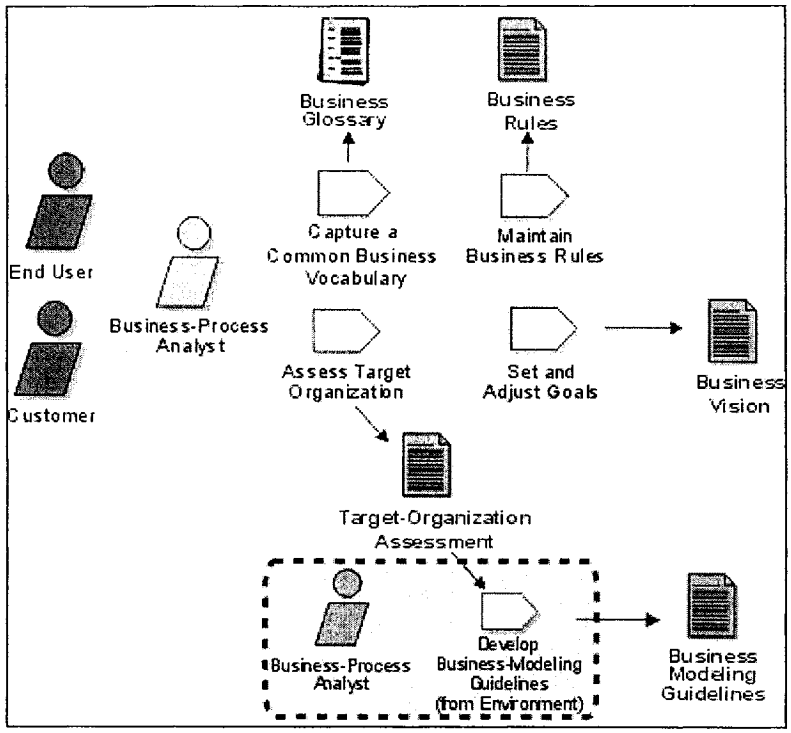


Figure 61: Business Modeling detail: Assess Business Status

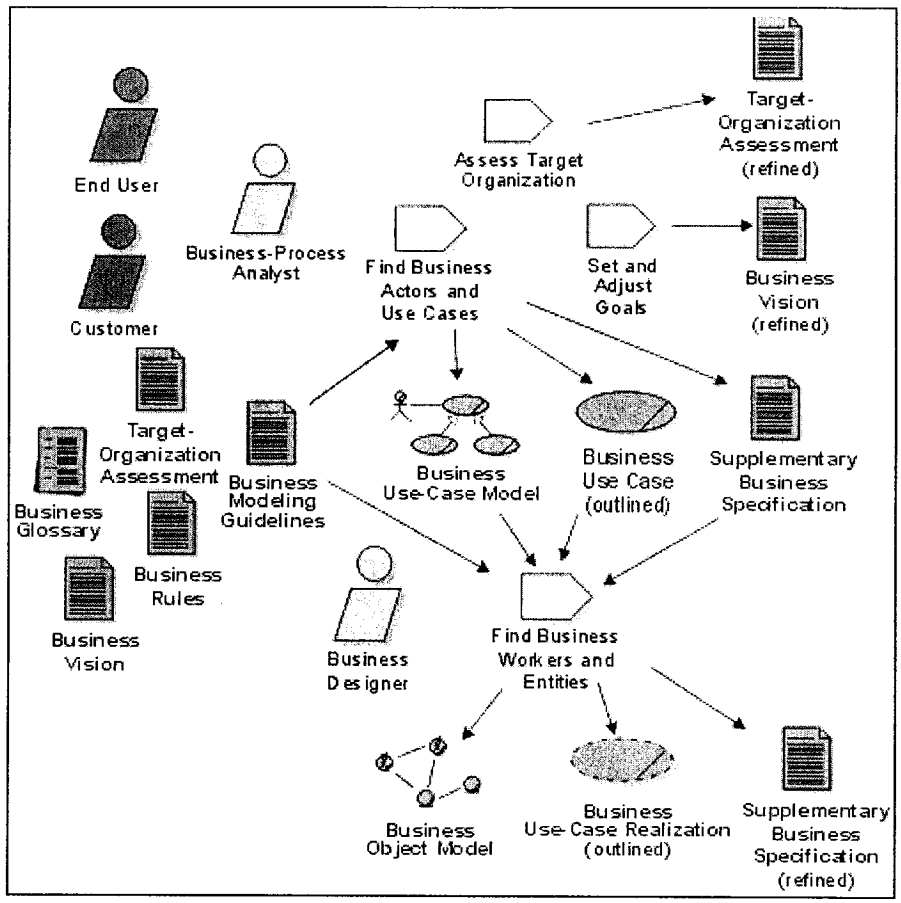


Figure 62: Business Modeling detail: Describe Current Business

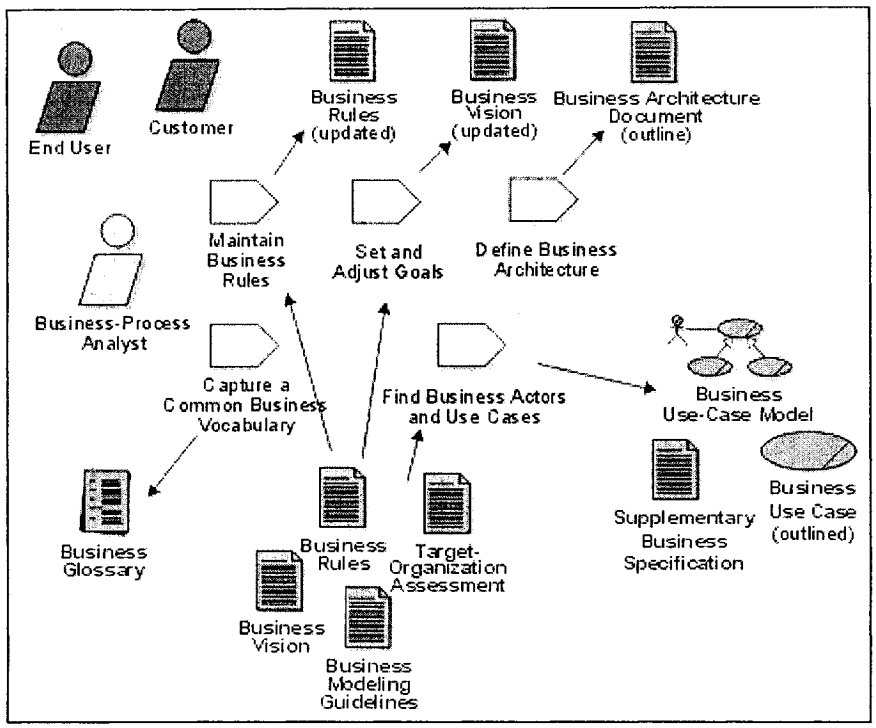


Figure 63: Business Modeling detail: Identify Business Processes

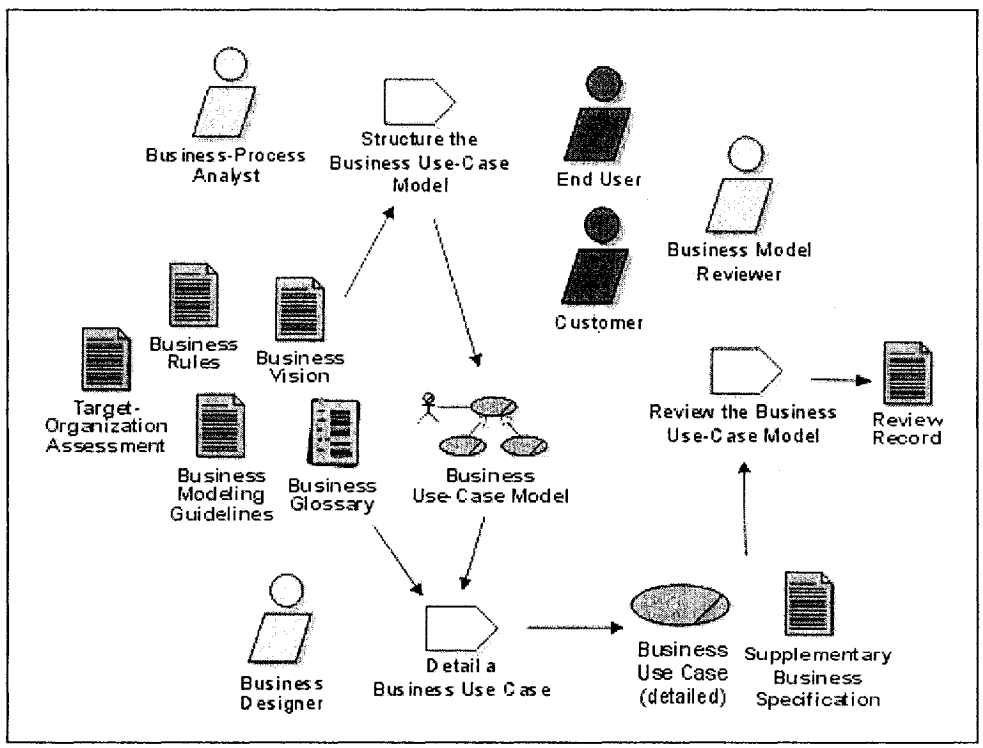


Figure 64: Business Modeling detail: Refine Business Process Definitions

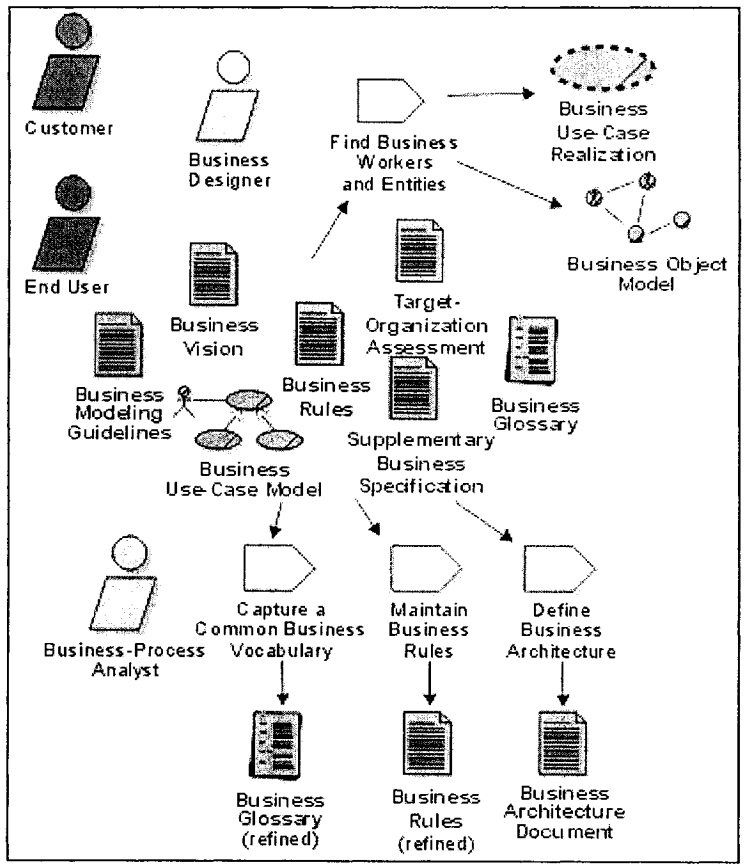


Figure 65: Business Modeling detail: Design Business Process Realizations

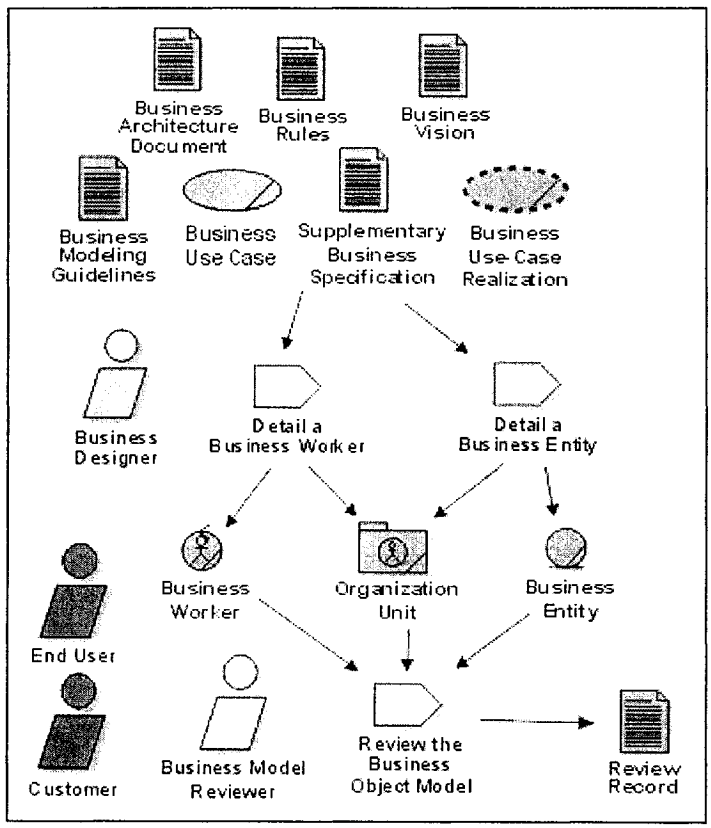


Figure 66: Business Modeling detail: Refine Roles and Responsibilities

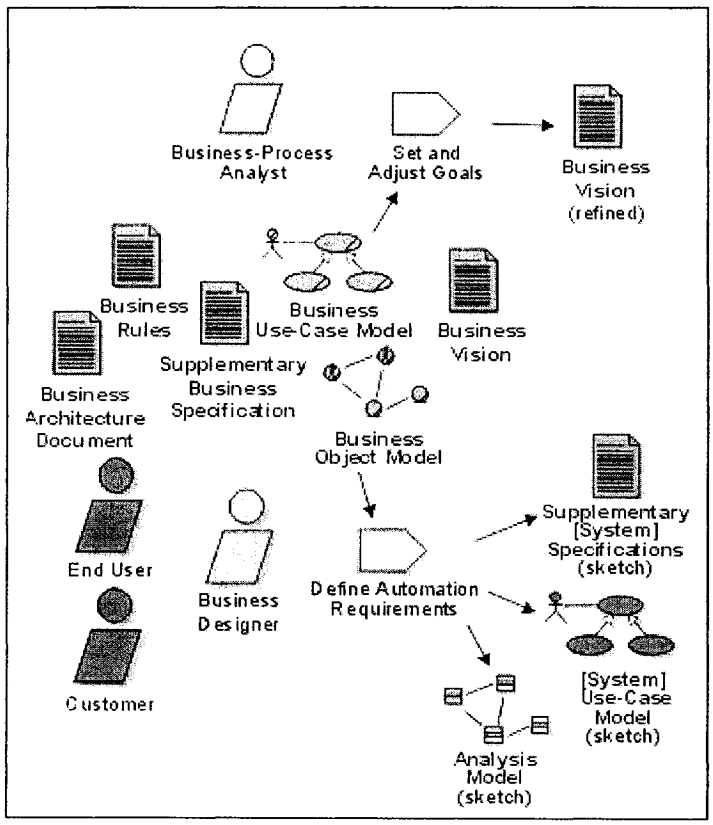


Figure 67: Business Modeling detail: Explore Process Automation

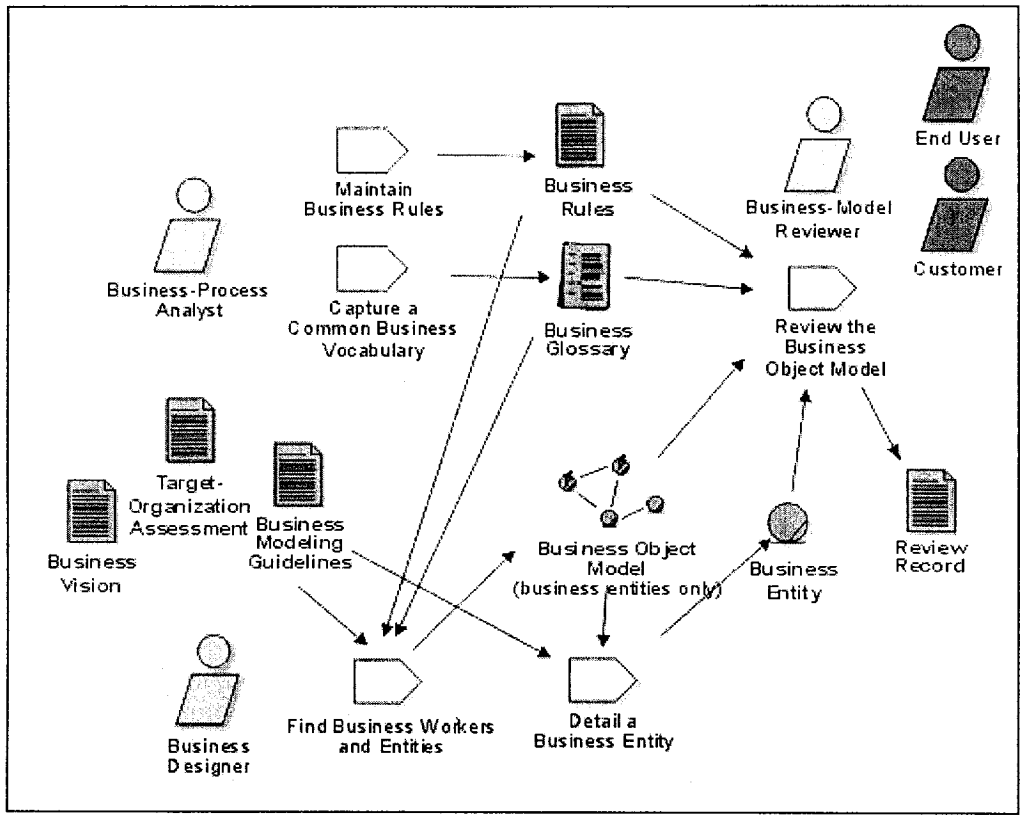


Figure 68: Business Modeling detail: Develop a Domain Model

A.2 Requirements

A.2.1 Requirements workflow

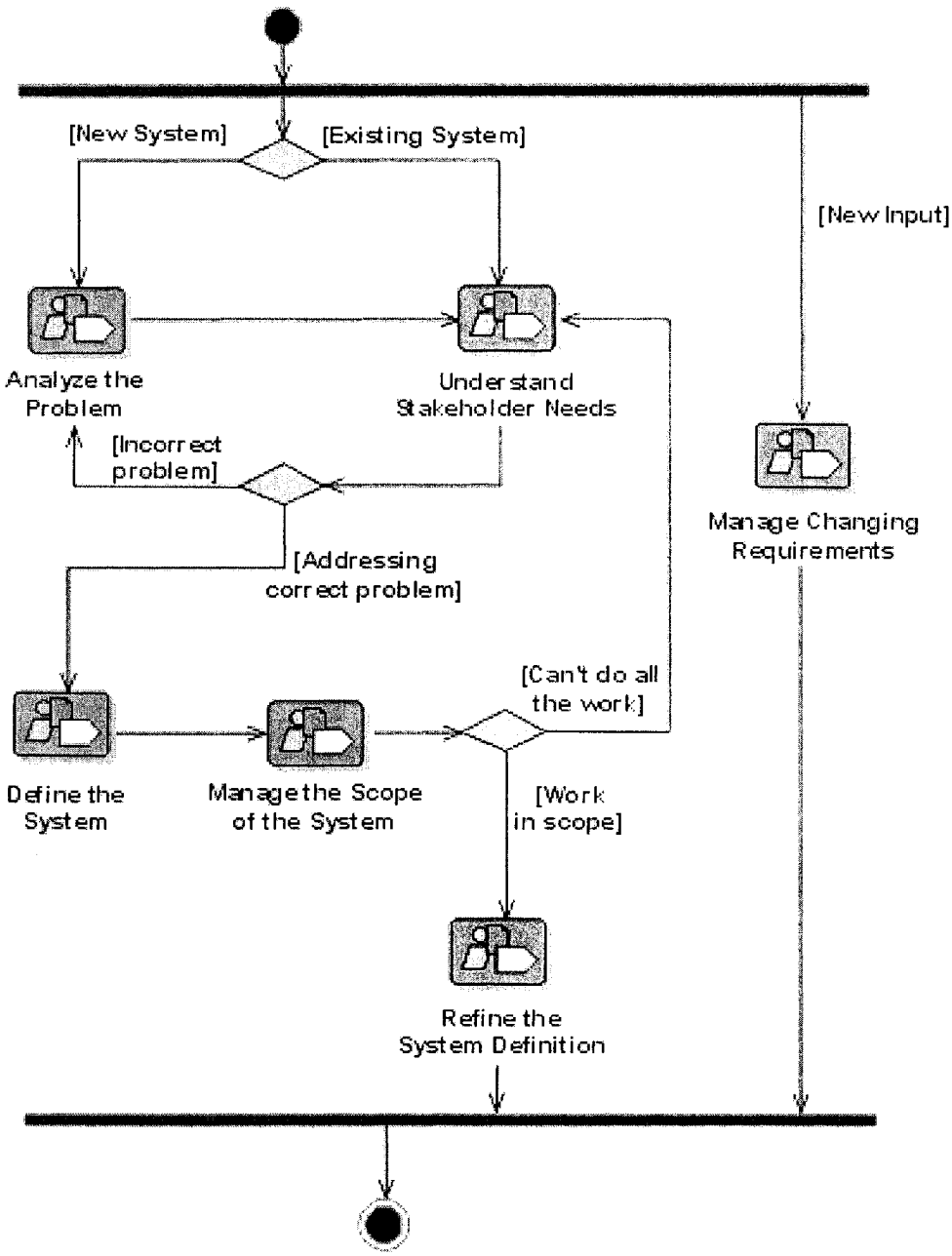


Figure 69: Activity diagram for the *Requirements* discipline

A.2.2 Requirements workers

System Analyst. He leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality and delimiting the system; for example, establishing what actors and use cases exist, and how they interact.

Requirements Specifier. The requirements specifier role details the specification of a part of the system's functionality by describing the Requirements aspect of one or several use cases and other supporting software requirements. The requirements specifier may also be responsible for a use-case package, and maintains the integrity of that package. It is recommended that the requirements specifier responsible for a use-case package is also responsible for its contained use cases and actors.

User Interface Designer. He leads and coordinates the prototyping and design of the user interface, by: capturing requirements on the user interface, including usability requirements and building user-interface prototypes. He involves other stakeholders of the user interface, such as end-users, in usability reviews and use testing sessions. He reviews and provides the appropriate feedback on the final implementation of the user interface, as created by other developers; that is, designers and implementers. User interface design can mean one of the two things: (1) Visual shaping of the user interface so that it handles various usability requirements; (2) the design of the user interface in terms of design classes (and components such as ActiveX classes and Java Beans), that is related to other design classes dealing with business logic, persistence, and that leads to the final implementation of the user interface. That's why user interface design appears in [2] as part of the Analysis & Design Discipline.

Requirements Reviewer. He plans and conducts the formal review of the use-case model.

Software Architect. He leads and coordinates technical activities and artifacts throughout the project. The software architect establishes the overall structure for

each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Therefore, in contrast to the other roles, the software architect's view is one of breadth as opposed to one of depth.

A.2.3 Requirements activities

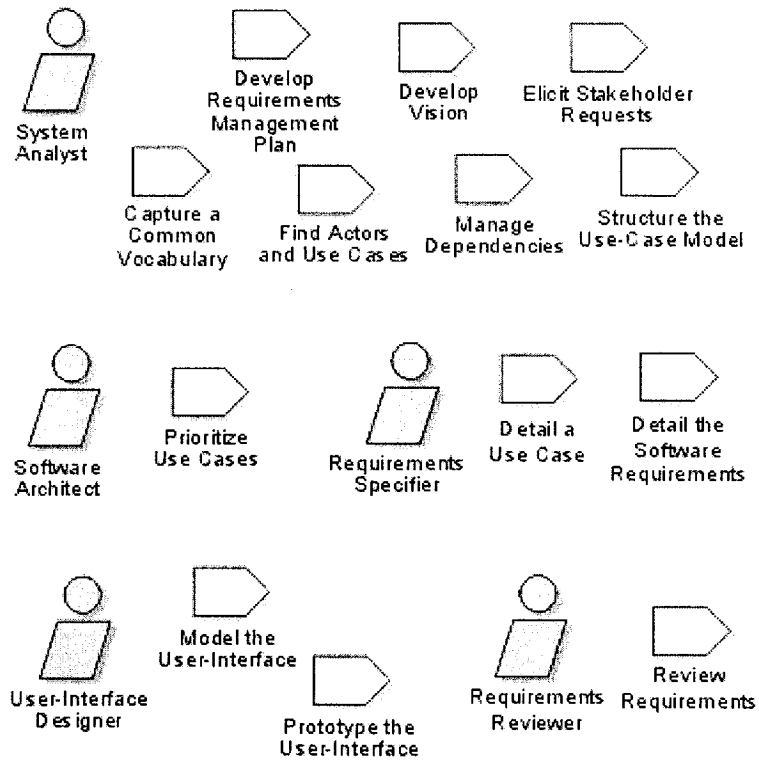


Figure 70: Activities by worker in the *Requirements* discipline

A.2.3.1 System Analyst Activities

Capture a Common Vocabulary. The purpose is to define a common vocabulary that can be used in all textual descriptions of the system, especially in use-case descriptions.

Develop Vision. The purpose is to gain agreement on what problems need to be solved, to identify stakeholders to the system, to define the boundaries of the system, to describe primary features of the system.

Develop Requirements Management Plan. The purpose is to develop a plan for documenting requirements, their attributes and guidelines for traceability and management of product requirements.

Elicit Stakeholder Requests. The purpose is to understand who are the stakeholders of the project, to collect requests on what needs the system should fulfill, to prioritize stakeholder requests.

Find Actors and Use Cases. The purpose is to outline the functionality of the system, to define what will be handled by the system and what will be handled outside the system, to define who and what will interact with the system, divide the model into packages with actors and use cases, create diagrams of the use-case model and develop a survey of the use-case model.

Manage Dependencies. We use attributes and traceability of project requirements to assist in managing the scope of the project and manage changing requirements.

Structure the Use Case Model. We extract behavior in use cases that need to be considered as abstract use cases. Examples of such behavior are common behavior, optional behavior, exceptional behavior, and behavior that will be developed in later iterations. We also find new abstract actors that define roles that are shared by several actors.

A.2.3.2 Requirements Specifier Activities

Detail a use case. We describe the use case's flow of events in detail, so that the customer and the users can understand it.

Detail the Software Requirements. We collect, detail and organize the set (package) of artifacts that completely describe the software requirements of the system or subsystem.

A.2.3.3 User Interface Designer Activities

Model the user interface. We build a model of the user interface that supports the reasoning about, and the enhancement of, its usability.

Prototype the user interface. We create a user-interface prototype.

A.2.3.4 Requirements Reviewer Activities

Review the Requirements. We formally verify that the results of Requirements conform to the customer's view of the system.

A.2.3.5 Software Architect Activities

Prioritize use cases. We define input to the selection of the set of scenarios and use cases that are to be analyzed in the current iteration. We also define the set of scenarios and use cases that represent some significant, central functionality or those that have a substantial architectural coverage (that exercise many architectural elements) or that stress or illustrate a specific, delicate point of the architecture.

A.2.4 Requirements artifacts

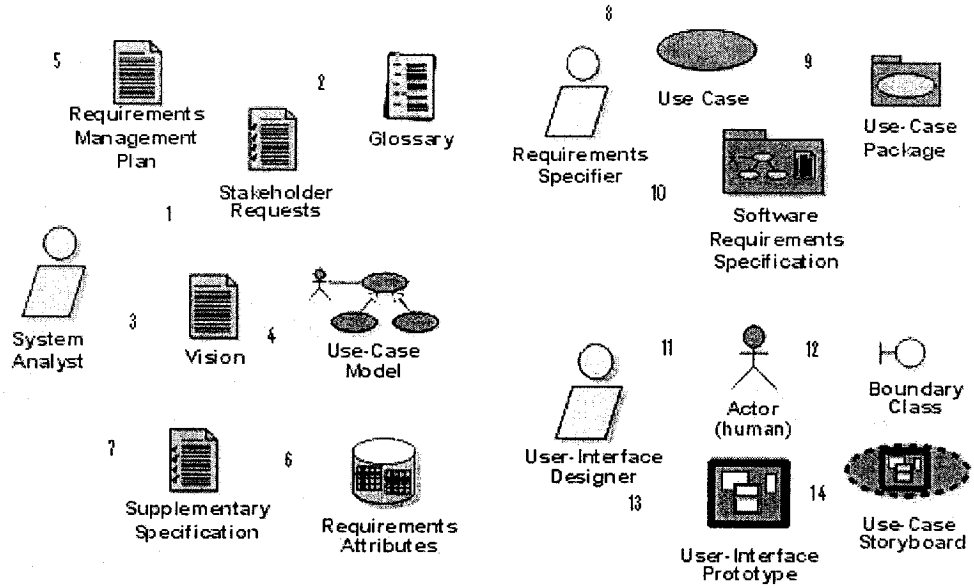


Figure 71: Artifacts produced in the *Requirements* discipline

A.2.4.1 System Analyst Artifacts

Requirements Management Plan. It describes the requirements documentation, requirement types, and their respective requirements attributes, specifying the information and control mechanisms to be collected and used for measuring, reporting, and controlling changes to the product requirements.

Glossary. It defines important terms used by the project.

Stakeholder Requests. It contains any type of requests a stakeholder (customer, end user, marketing person, and so on) might have on the system to be developed. It may also contain references to any type of external sources to which the system must comply.

Vision. It defines the stakeholders' view of the product to be developed, specified in terms of the stakeholders' key needs and features. Containing an outline of the envisioned core requirements, it provides the contractual basis for the more detailed technical requirements.

Supplementary Specification. It captures the system requirements that are not readily captured in the use cases of the use-case model. Such requirements include: legal and regulatory requirements, and application standards, quality attributes of the system to be built, including usability, reliability, performance, and supportability requirements and other requirements such as operating systems and environments, compatibility requirements, and design constraints.

Requirements Attributes. It contains a repository of project requirements, attributes and dependencies to track from a requirements management perspective.

Use case model. It is a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The use-case model is used as an essential input to activities in analysis, design, and test.

A.2.4.2 Requirements Specifier Artifacts

Use Case. It defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

Use Case Package. It is a collection of use cases, actors, relationships, diagrams, and other packages; it is used to structure the use-case model by dividing it into smaller parts.

Software Requirements Specification. It captures the complete software requirements for the system, or a portion of the system. When using use-case modeling, this artifact consists of a package containing use cases of the use-case model and applicable Supplementary Specifications.

A.2.4.3 User Interface Designer Artifacts

Actor. It defines a coherent set of roles that users of the system can play when interacting with it. An **actor instance** can be played by either an individual or an external system.

Boundary Class. It models the interaction between one or more actors and the system.

User-Interface Prototype. It is a prototype of the user interface. For example, the prototype can manifest itself as: paper sketches or pictures, bitmaps from a drawing tool or an interactive executable prototype; for example, in Microsoft Visual Basic®

Use-case Storyboard. It is a logical and conceptual description of how a use case is provided by the user interface, including the interaction required between the actor(s) and the system.

A.2.4.4 Requirements Reviewer Artifacts

Review Record. It is a form document that is filled out for each review. It is created as a control document to manage the execution of the review of project artifacts. It is issued to the participants in the review to initiate the review process, and is used to capture the results and any action items arising from the review meeting. It forms an auditable record of the review and its conclusions.

A.2.4.5 Software Architect Artifacts

Software Architecture Document. It provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. In this case, it will be a view of the use case model with its most significant use cases, those that describe important or critical

functionality or that concern a requirement that needs to be developed early in the life cycle [1].

A.2.5 Requirements details

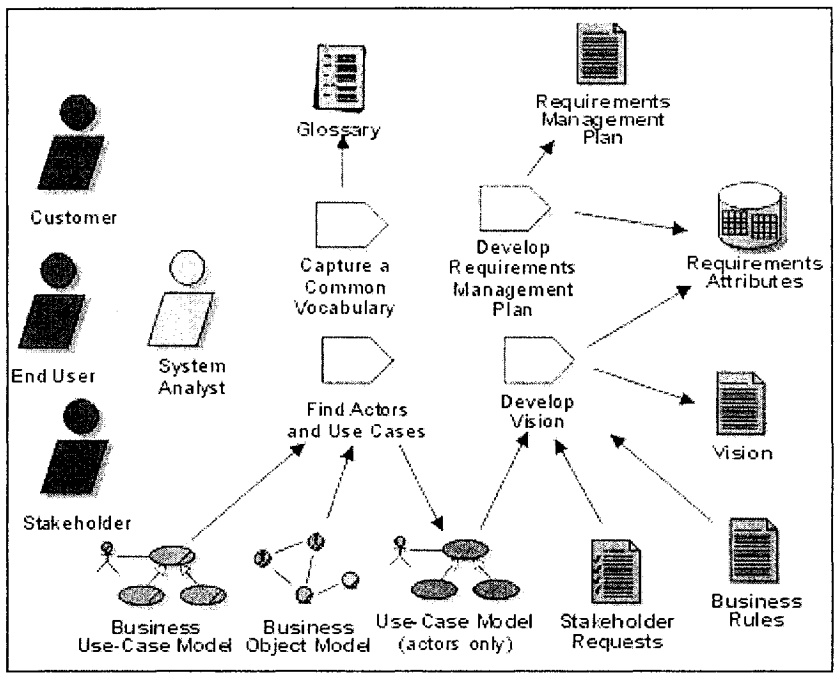


Figure 72: Requirements detail: Analyze the problem

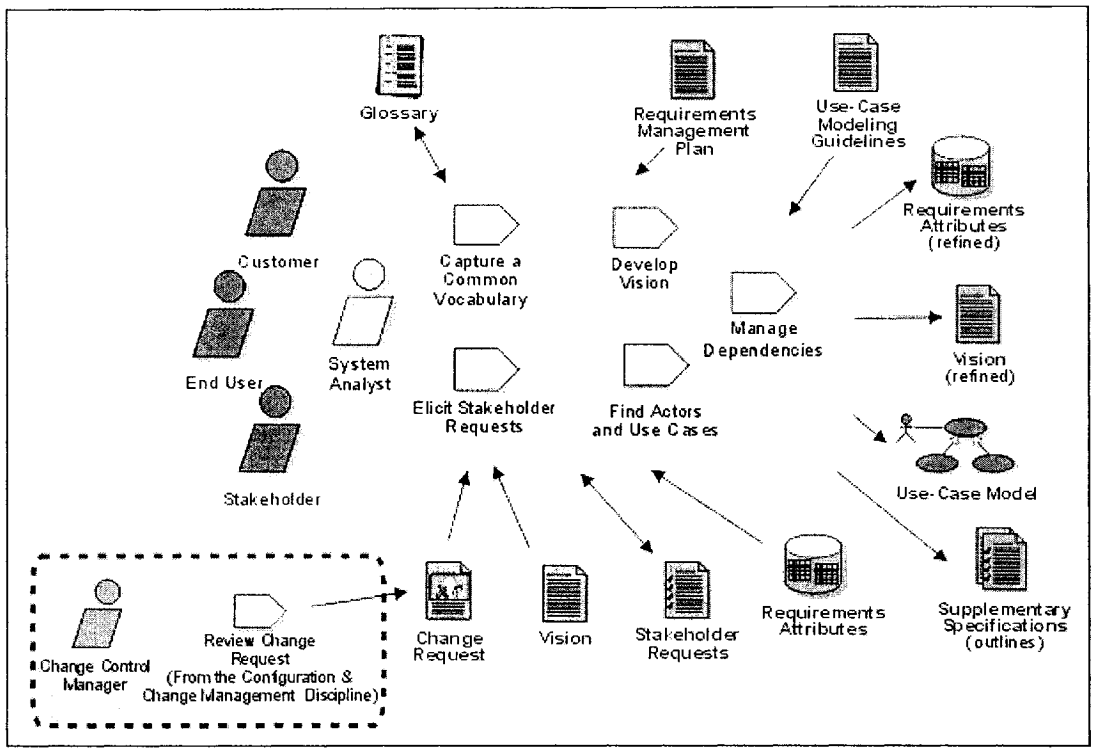


Figure 73: Requirements detail: Understand Stakeholder Needs

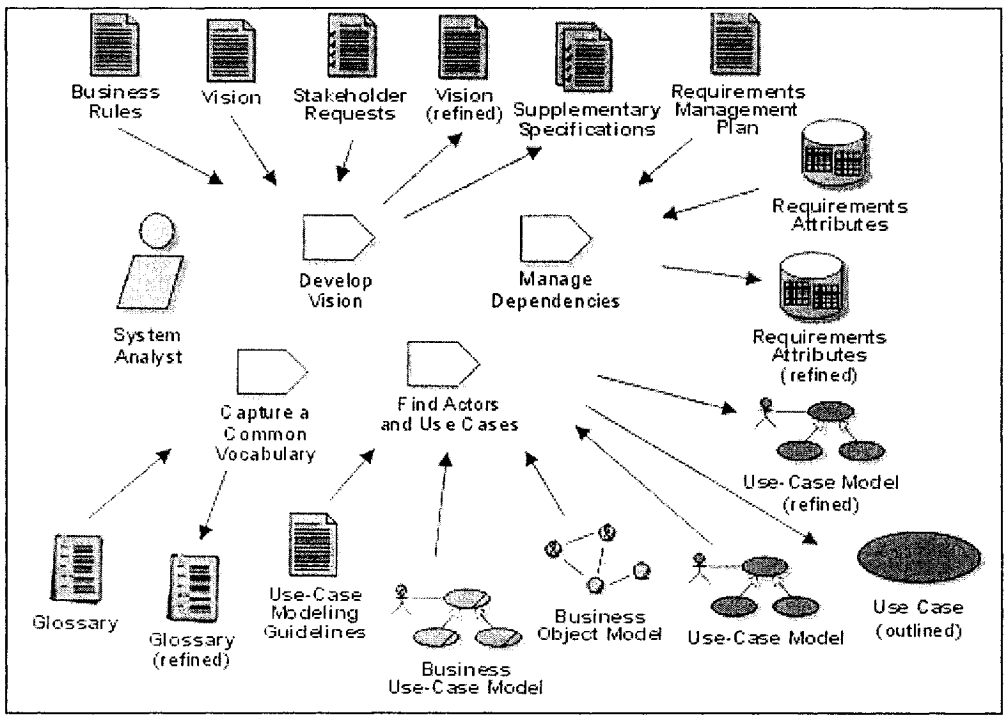


Figure 74: Requirements detail: Define the system

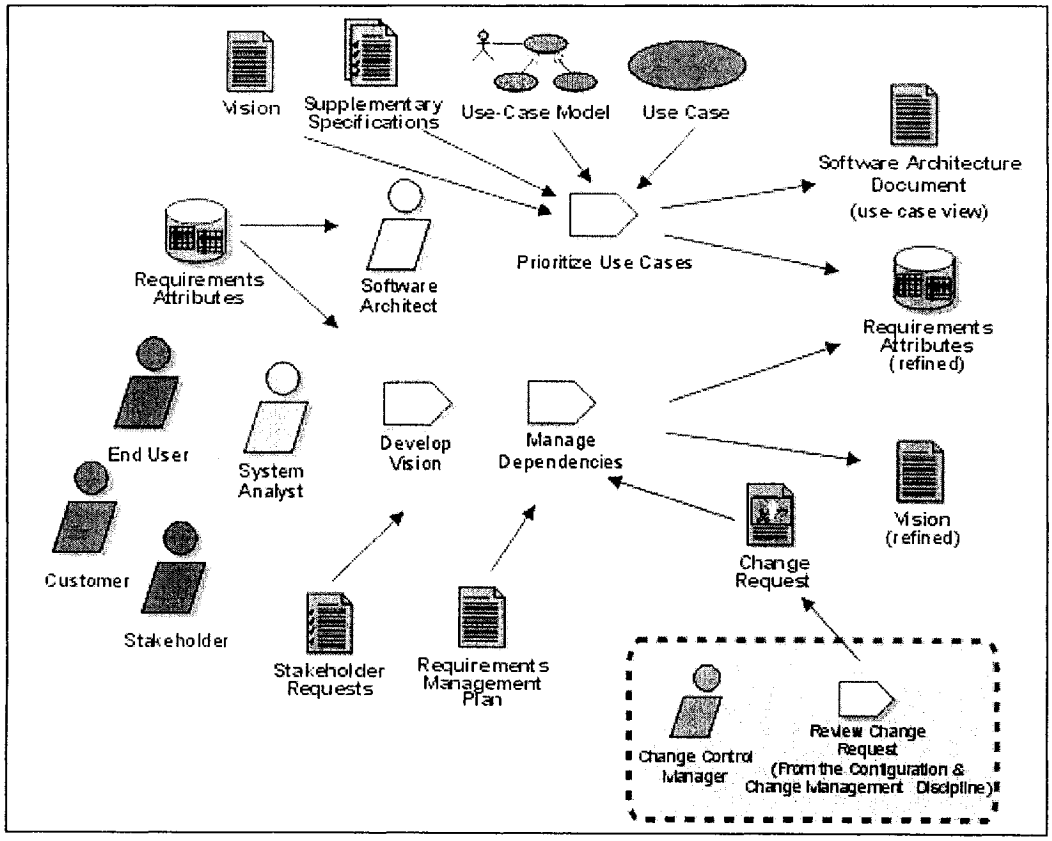


Figure 75: Requirements detail: Manage the scope of the system

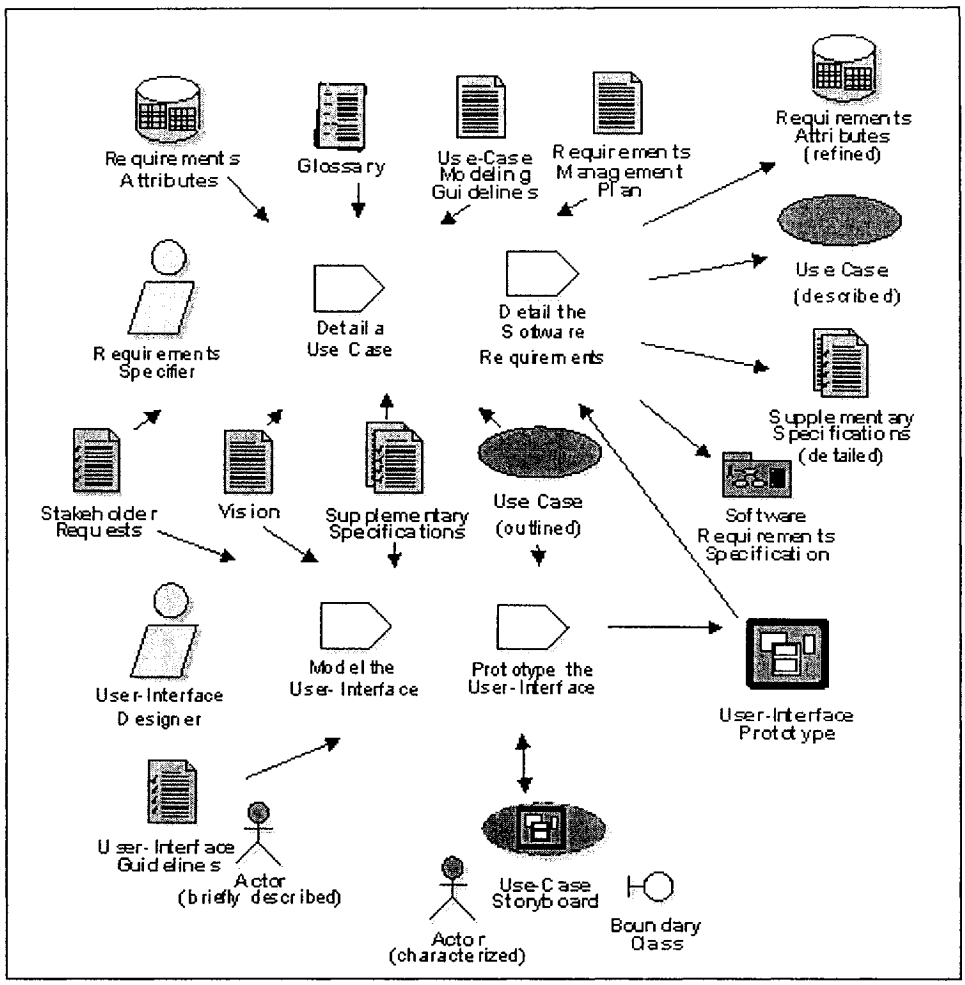


Figure 76: Requirements detail: Refine the system definition

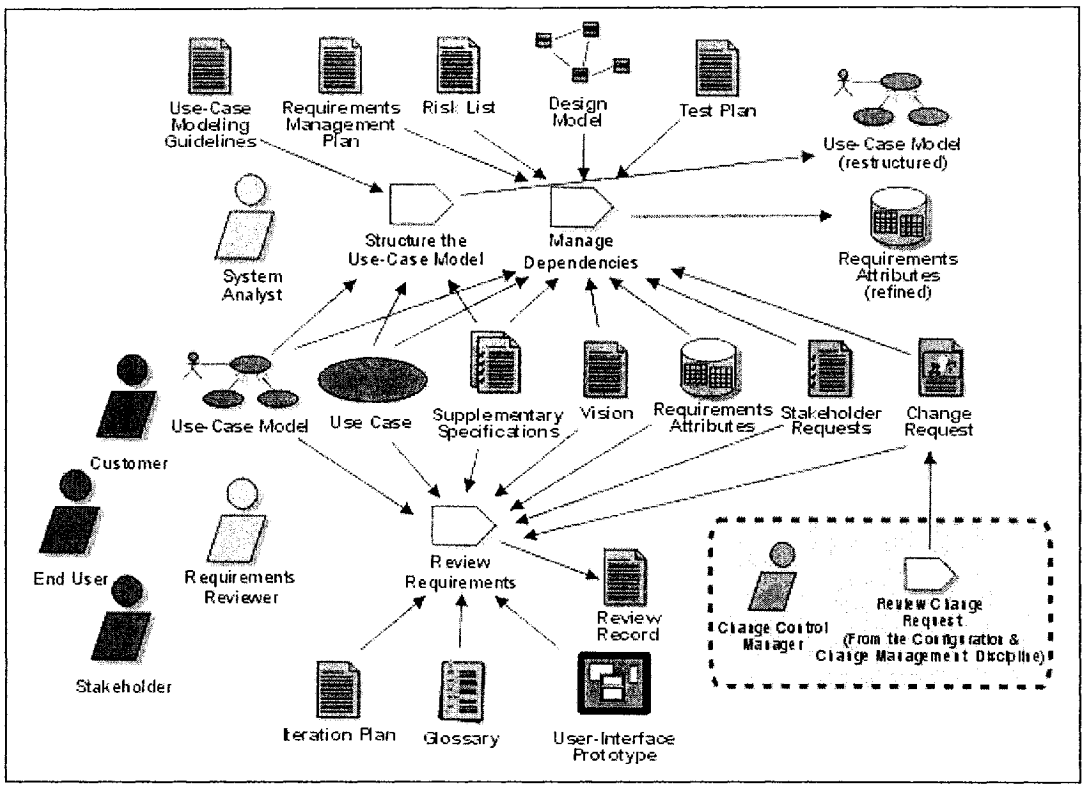


Figure 77: Requirements detail: Manage changing requirements

A.3 Analysis and Design

A.3.1 Analysis and Design Workflow

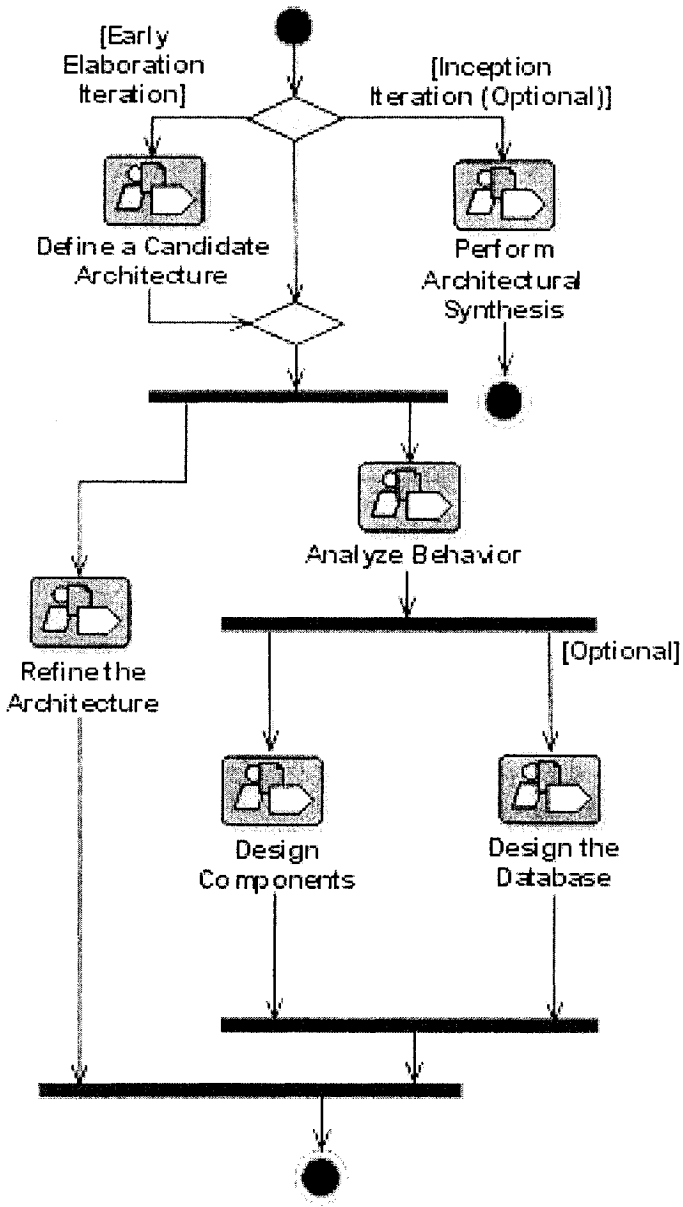


Figure 78: Activity Diagram for the *Analysis and Design* discipline

A.3.2 Analysis and Design Workers

Software Architect. He leads and coordinates technical activities and artifacts throughout the project. The software architect establishes the overall structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Therefore, in contrast to the other roles, the software architect's view is one of breadth as opposed to one of depth.

Designer. He defines the responsibilities, operations, attributes, and relationships of one or several classes, and determines how they will be adjusted to the implementation environment. In addition, the designer role may have responsibility for one or more design packages, or design subsystems, including any classes owned by the packages or subsystems.

Capsule Designer. He ensures the system can respond to events in a timely manner, in accordance with concurrency requirements. The primary vehicle for solving these problems is the Artifact: Capsule.

Database Designer. He defines the tables, indexes, views, constraints, triggers, stored procedures, tablespaces or storage parameters, and other database-specific constructs needed to store, retrieve, and delete persistent objects. This information is maintained in the Artifact: Data Model.

Architecture reviewer. He plans and conducts the formal reviews of the software architecture in general.

Design Reviewer. He plans and conducts the formal reviews of the Artifact: Design Model.

A.3.3 Analysis and Design Activities

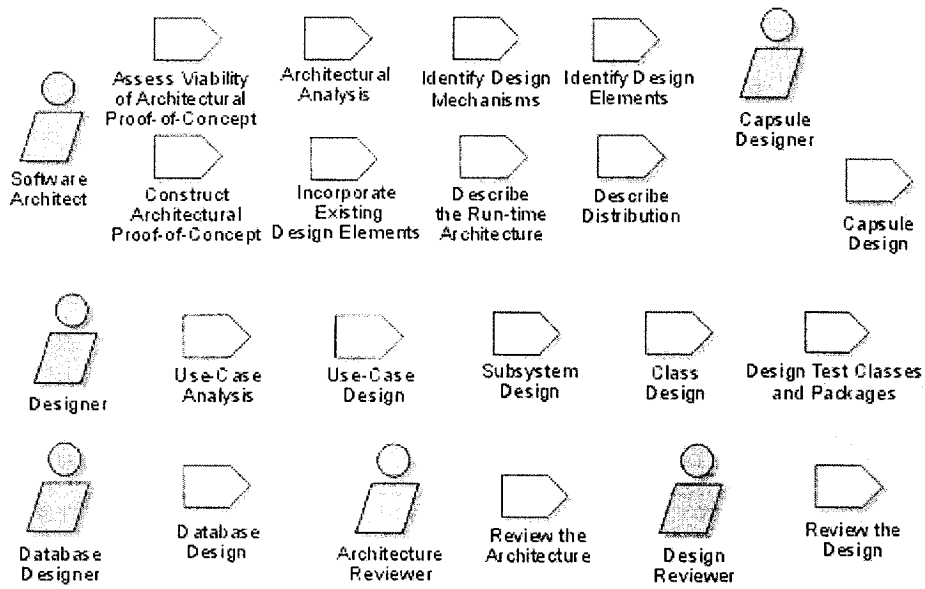


Figure 79: Activities by worker in the *Analysis and Design* discipline

A.3.3.1 Software Architect Activities

Architectural Analysis. The purpose is to define a candidate architecture for the system, based on experience gained from similar systems or in similar problem domains. The software architect also defines the architectural patterns, key mechanisms and modeling conventions for the system, the reuse strategy and provides input to the planning process.

Identify Design Mechanisms. The purpose is to refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.

Identify Design Elements. The purpose is to analyze interactions of analysis classes in order to identify design model elements.

Incorporate Existing Design Elements. The purpose is to analyze interactions of analysis classes in order to find interfaces, design classes and design subsystems, to refine the architecture, by incorporating reuse where possible, to identify common solutions to commonly encountered design problems and to include architecturally significant design model elements in the Logical View section of the Software Architecture Document.

Describe the Run-time Architecture. The purpose is to analyze concurrency requirements, to identify processes, identify inter-process communication mechanisms, allocate inter-process coordination resources, identify process lifecycles, and distribute model elements among processes.

Describe Distribution. The purpose is to describe how the functionality of the system is distributed across physical nodes (this activity is required only for distributed systems).

Construct Architectural Proof-of-Concept. The purpose is to synthesize at least one solution (which may simply be conceptual) that meets the critical architectural requirements.

Assess Viability of Architectural Proof-of-Concept. The purpose is to evaluate the synthesized Architectural Proof-of-Concept to determine whether the critical architectural requirements are feasible and can be met (by this or any other solution).

A.3.3.2 Designer Activities

Use case Analysis. He identifies the classes which perform a use case's flow of events, distributes the use case behavior to those classes, using use-case realizations, identifies the responsibilities, attributes and associations of the classes and notes the usage of architectural mechanisms.

Use case Design. This activity is concerned with the refining of several design elements: use-case realizations in terms of interactions, the requirements on the operations of design classes, the requirements on the operations of subsystems and/or their interfaces and the requirements on the operations of capsules.

Subsystem Design. This activity is concerned with defining the behaviors specified in the subsystem's interfaces in terms of collaborations of contained classes, documenting the internal structure of the subsystem, defining realizations between the subsystem's interfaces and contained classes and determining the dependencies upon other subsystems. We design subsystems to be independent from other subsystems. However, if dependencies exist, we maintain and minimize them. In a similar way, we maintain and refine the interfaces between the subsystems [1].

Class Design. We design a class to fulfill its role in use case realization and the non-functional requirements that apply to it. We outline the design class in terms of the analysis classes and interfaces. We need special tools and technology to design different types of classes. We than identify the operation on the class, either from the corresponding analysis classes or from the use case realization – design in which the class participates. We also identify its attributes, which take into account the programming language, the associations and aggregations between the design classes and the generalizations (if supported by the programming language) [1]. We also incorporate the design mechanisms used by the class.

Design Test Classes & Packages. The purpose is to design test-specific functionality. It includes: Identifying Test-Specific Classes and Packages, Designing Interface to Automated Test Tool and Designing Test Procedure Behavior.

A.3.3.3 Capsule Designer Activities

Capsule Design. This activity is concerned with elaborating and refining the descriptions of a capsule. It includes creating an initial set of port classes that represent the interfaces to the capsule, then finding an appropriate protocol to bind to these ports. The Capsule is defined as a state machine, which has states and transitions between them. The internal and external behavior of the Capsule is tested and validated by simulating the events that will exercise the Capsule behavior and by considering the interaction with other Capsules.

A.3.3.4 Database Designer Activities

Database Design. The purpose is to ensure that persistent data is stored consistently and efficiently and to define behavior that must be implemented in the database.

A.3.3.5 Architecture Reviewer Activities

Review the Architecture. During this activity, the Reviewer uncovers any unknown or perceived risks in the schedule or budget. He detects any architectural design flaws (which are known to be the hardest to fix, the most damaging in the long run), or any possible mismatch between the requirements and the architecture: over-design, unrealistic requirements, or missing requirements. In particular the assessment may examine some aspects often neglected in the areas of operation, administration and maintenance. How is the system installed? Updated? How do we transition the current databases? He identifies reuse opportunities and evaluates one or more specific architectural qualities: performance, reliability, modifiability, security, safety.

A.3.3.6 Design Reviewer Activities

Review the Design. The purpose is to verify that the design model fulfills the requirements on the system, and that it serves as a good basis for its

implementation, to ensure that the design model is consistent with respect to the general design guidelines and that the design guidelines fulfill their objectives.

A.3.4 Analysis and Design Artifacts

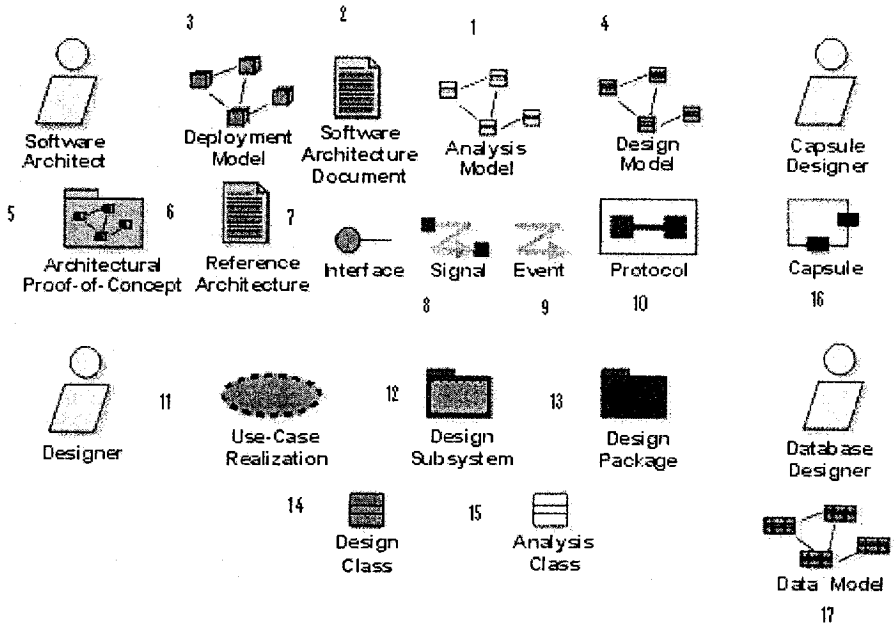


Figure 80: Artifacts produced in the *Analysis and Design* discipline

A.3.4.1 Software Architect Artifacts

Analysis Model. It is an object model describing the realization of use cases, and which serves as an abstraction of the Design Model. The Analysis Model contains the results of use case analysis, instances of the Analysis Class.

NB: The analysis model may be an optional model in the development of the product. For very small projects, where the requirements are clear and understandable, the analysis may be part of either the requirements capture or of the design. For important projects, the analysis model is necessary, because it provides the conceptual basis for the more "physical" design model [1].

Design Model. It is an object model describing the realization of use cases, which serves as an abstraction of the implementation model and its source code. The design model is used as essential input to activities in implementation and test.

Deployment Model. It is an object model that describes the physical distribution of the system onto nodes. Each node represents a computational resource. Nodes have relationships that represent means of communication between them – Internet, intranet, bus.

Software Architecture Document. The Software Architecture Document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. In the first edition of RUP, Analysis and Design are two separate disciplines. In each one, a different artifact is produced, i.e. Architecture description (view of the Analysis Model) and an Architecture description (view of the Design Model), whose definitions are provided here. Moreover, in the Design discipline an Architecture description is produced that contains the view of the Deployment Model.

Architectural description (view of the Analysis Model). The architecture description contains an architectural view of the analysis model, depicting its architecturally significant artifacts. Architecturally significant artifacts are the decomposition of the analysis model into analysis packages and their dependencies, which impacts the subsystems in design and implementation (so impacts the architecture). The key analysis classes are: entity classes that encapsulate an important phenomenon of the problem domain, boundary classes that encapsulate important communication interfaces, control classes that encapsulate important sequences with large coverage, analysis classes that are general, central and have many relations with other analysis classes, use case

realizations that realize important and critical functionality (involve many analysis classes) [1].

Architecture description (view of the Design Model). It contains an architecture view of the design model, depicting its architecturally significant artifacts (subsystems, their interfaces dependencies; key design classes, active classes, general and central classes, use case realization – design that realize implementation and critical functionality that needs to be developed early in the software's life cycle, involve many design classes, have large coverage (across several subsystems) [1].

Architecture description (view of the Deployment Model). It contains an architectural view of the deployment model, which depicts its architecturally significant artifacts – mapping of component onto nodes [1].

Reference Architecture. It is, in essence, a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.

Architectural Proof-of-Concept. It is a solution, which may simply be conceptual, to the architecturally-significant requirements that are identified early in Inception.

Interface. It is a model element which defines a set of behaviors (a set of operations) offered by a **classifier** model element (specifically, a class, subsystem or component). A classifier may **realize** one or more interfaces. An interface may be realized by one or more classifiers. Any classifiers which realize the same interfaces may be substituted for one another in the system. Each interface should provide an unique and well-defined set of operations.

Signal. It is an asynchronous communication entity which may cause a state transition in the state machine of an object that receives it.

Event. It represents the specification of an occurrence in space and time; less formally, an occurrence of something to which the system must respond.

Protocol. It is a common specification for a set of Capsule ports.

A.3.4.2 Designer Artifacts

Use case realization. A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects. In the first edition of RUP, an additional artifact is produced during the Analysis discipline, i.e. Use case realization – Analysis, whose definition is provided here.

Use case realization – Analysis. It is a collaboration within the analysis model, which describes how a specific use case is realized and performed in terms of analysis classes and their interacting analysis objects. It has textual flow of events description, class diagrams, and interaction diagrams [1].

Analysis Class. It is an abstraction of one or more classes or subsystems in design. It handles only functional requirements. It defines interfaces in terms of operations or conceptual signatures and it defines attributes. There are several stereotypes on analysis classes, which we present here:

Boundary class. It is a model interaction between system and actors. They collect the requirements on the system boundary (i.e. windows, forms, panes, communication interfaces; terminals).

Entity class. It is used to model information that is long-lived and persistent, like an individual, real life object, real life event. It can be derived from business entity class.

Control class. It represents coordination, sequencing, transaction and control of other objects. It is used to represent complex derivation and calculation (business logic).

Design Class. A class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics. It is an abstraction of a class or other constraint in the system's implementation. It takes into account the language used in implementation; its relationships and methods are preserved in implementation. It can realize and provide interfaces [1].

Design Subsystem. Design Subsystems are a means of organizing the artifacts of the design model in more manageable pieces. They can consist of design classes, use case realizations, interfaces and other subsystems (recursively). A subsystem should be cohesive (contents strongly related). They should be loosely coupled (their dependencies on all other's interfaces should be minimal). They can represent a separation of concerns. They can often have top application layers, traces to analysis package or analysis classes. It can represent large grained components in implementation.
They can represent reused software products (in middleware and systems software layers) or legacy systems [1].

Design Package. A design package is a collection of classes, relationships, use-case realizations, diagrams, and other packages. It is used to structure the design model by dividing it into smaller parts.

A.3.4.3 Capsule Designer Artifacts

Capsule. A capsule is a specific design pattern which represents an encapsulated thread of control in the system.

A.3.4.4 Database Designer Artifacts

Data Model. The data model is a subset of the implementation model which describes the logical and physical representation of persistent data in the system. It also includes any behavior defined in the database, such as stored procedures, triggers, constraints, and so forth.

A.3.4.5 Architecture Reviewer Artifacts

Review record. It is a form document that is filled out for each review. It is created as a control document to manage the execution of the review of project artifacts. It is issued to the participants in the review to initiate the review process, and is used to capture the results and any action items arising from the review meeting. It forms an auditable record of the review and its conclusions.

Change Request. Changes to development artifacts are proposed through Change Requests (CRs). Change Requests are used to document and track defects, enhancement requests and any other type of request for a change to the product. The benefit of CRs is that they provide a record of decisions and, due to their assessment process, ensure that change impacts are understood across the project.

A.3.4.6 Design Reviewer Artifacts

Review record. (see A.3.4.5)

Change Request. (see A.3.4.5)

A.3.5 Analysis and Design Details

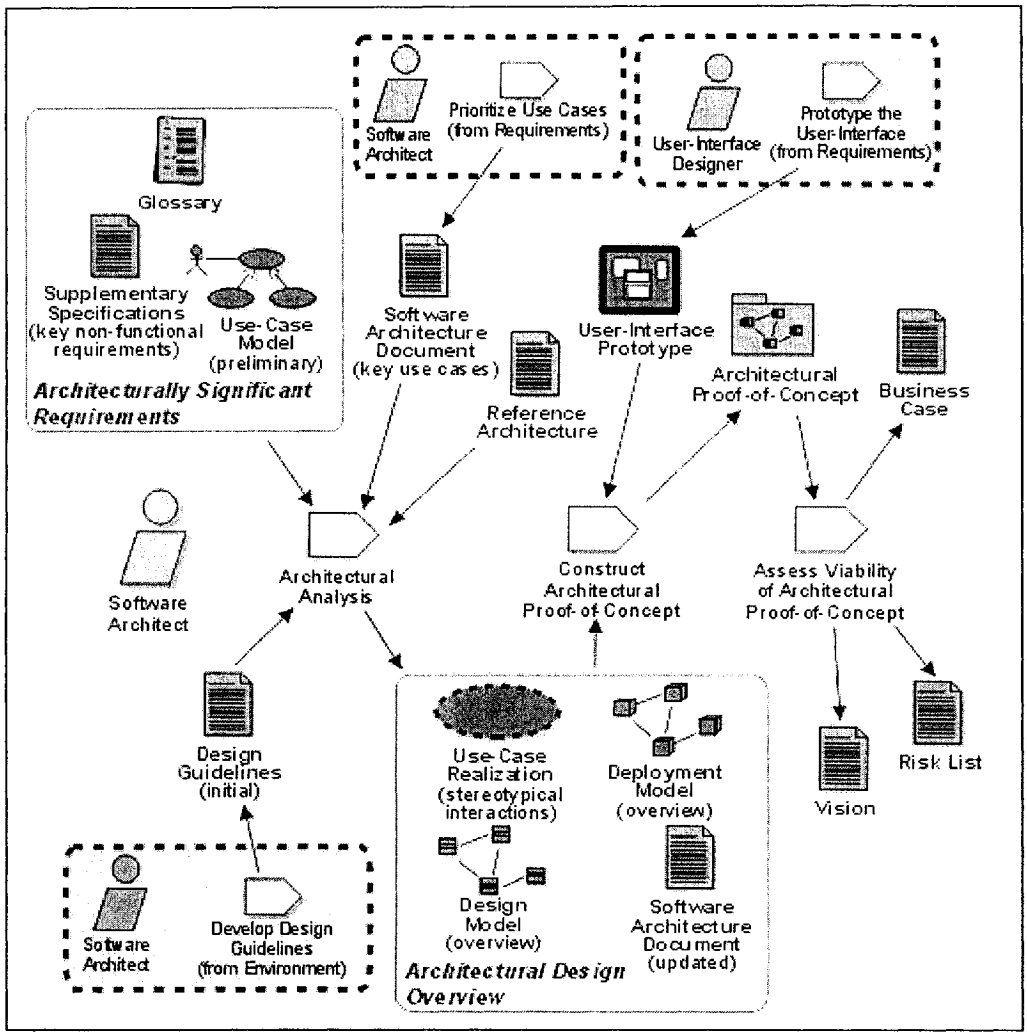


Figure 81: Analysis and Design detail: Perform Architectural Synthesis

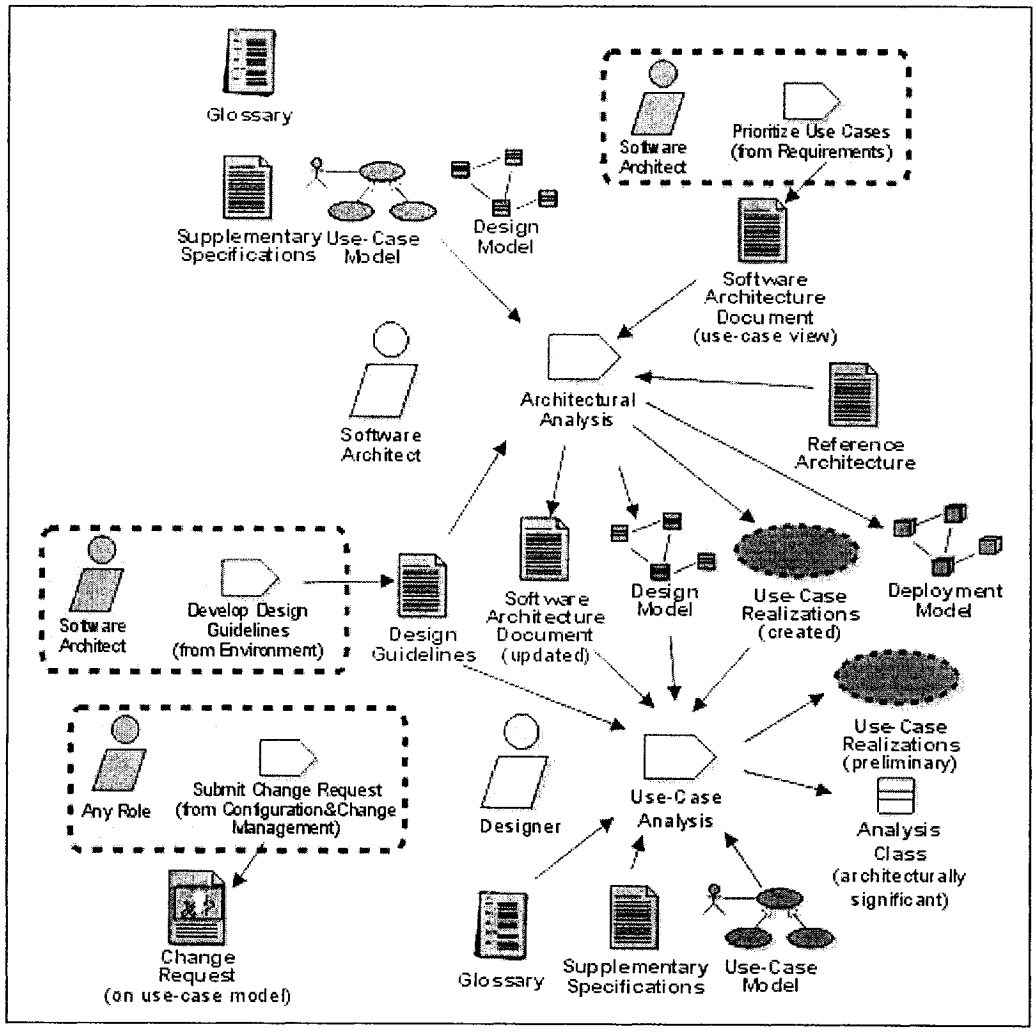


Figure 82: Analysis and Design detail: Define a candidate architecture

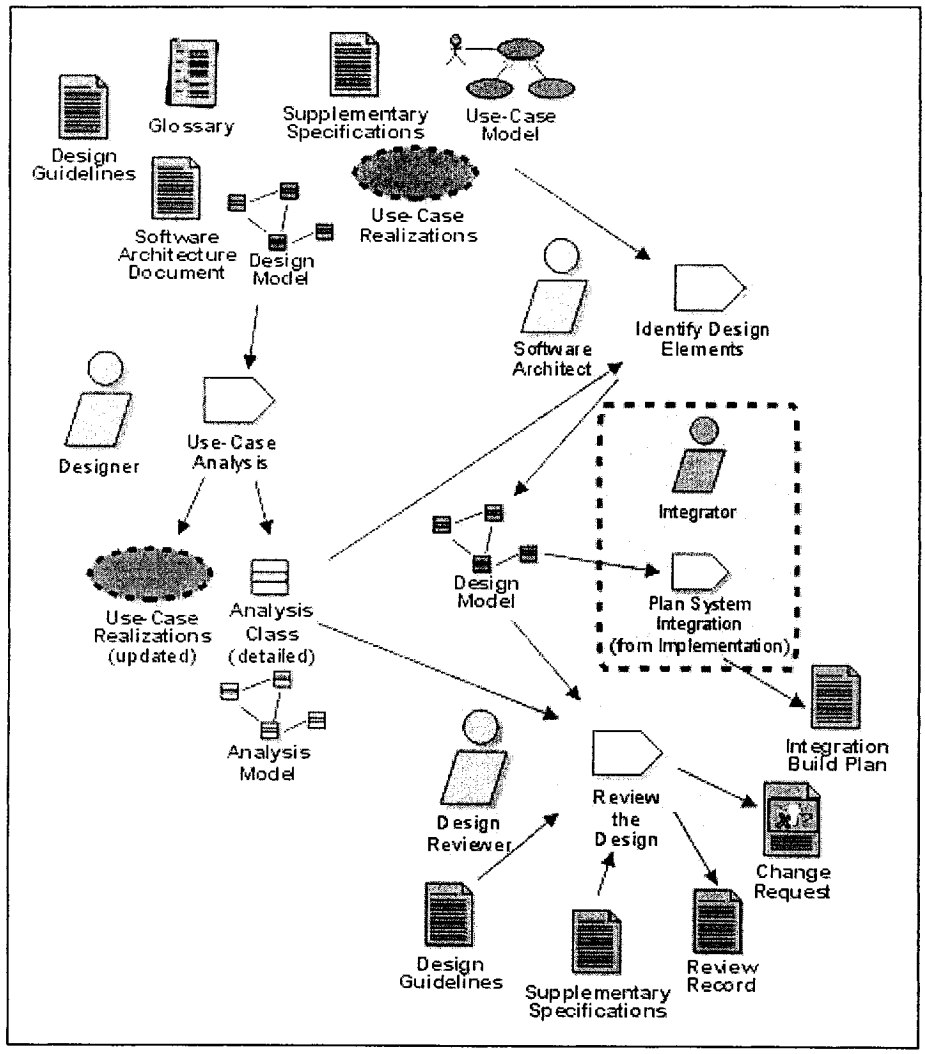


Figure 83: Analysis and Design detail: Analyze Behavior

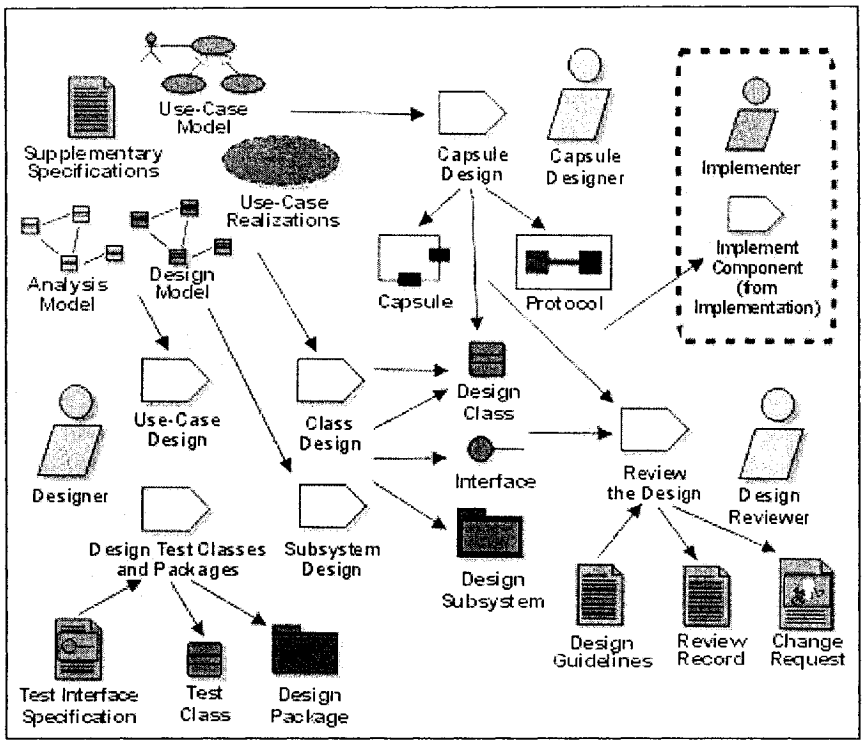


Figure 84: Analysis and Design detail: Design Components

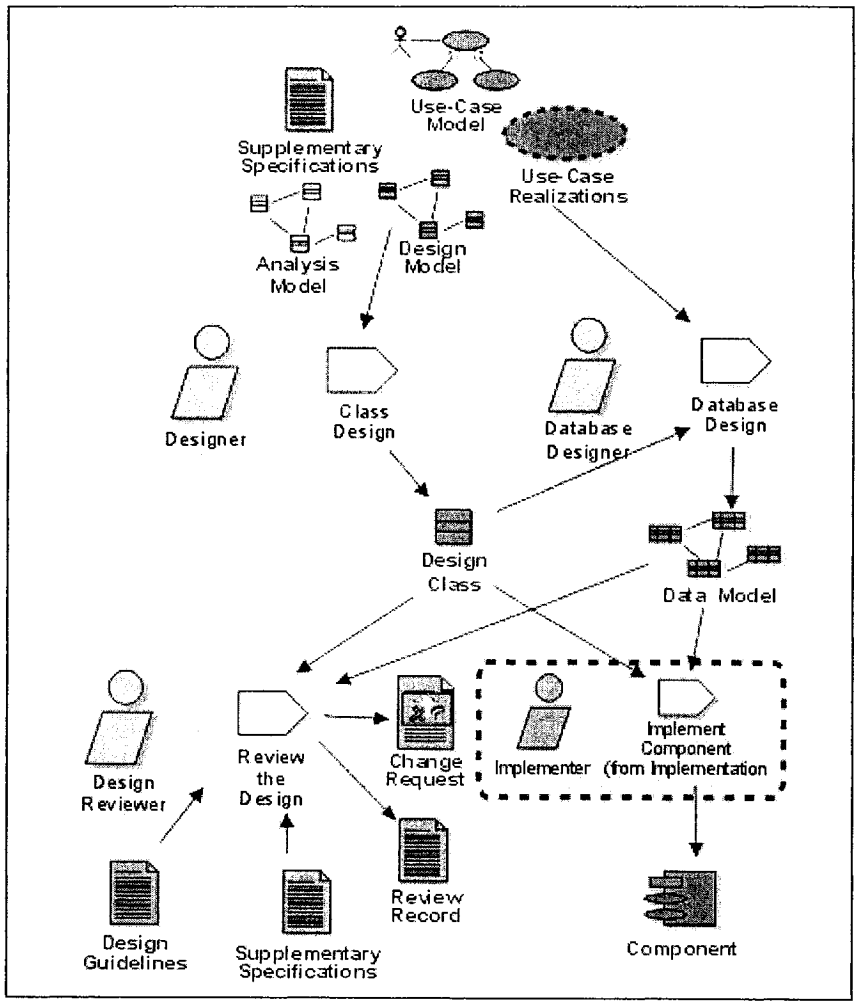


Figure 85: Analysis and Design detail: Design the Database

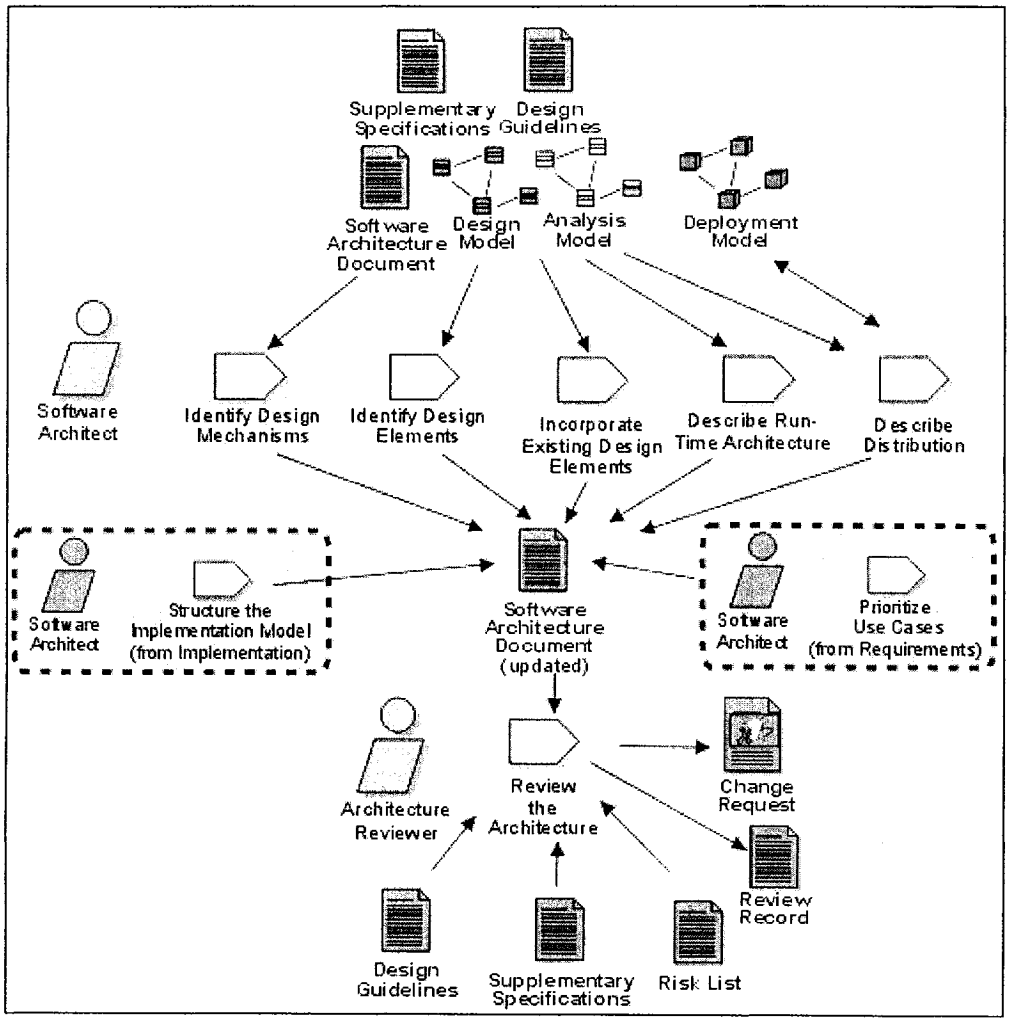


Figure 86: Analysis and Design detail: Refine the architecture

A.4 Implementation

A.4.1 Implementation workflow

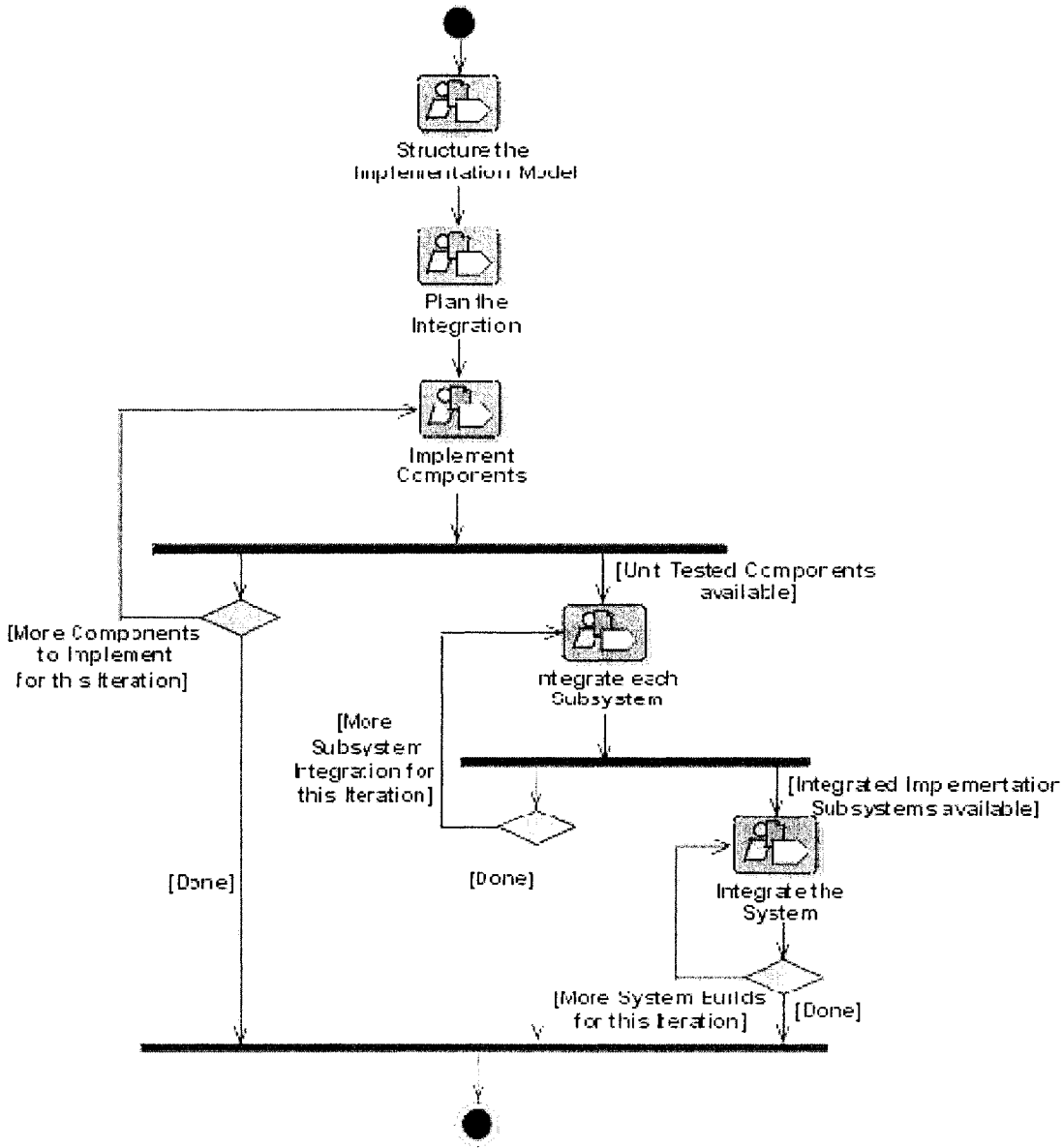


Figure 87: Activity diagram for the *Implementation* discipline

A.4.2 Implementation workers

Software Architect. He leads and coordinates technical activities and artifacts throughout the project. The software architect establishes the overall structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Therefore, in contrast to the other roles, the software architect's view is one of breadth as opposed to one of depth.

Implementer. He is responsible for developing and testing components, in accordance with the project adopted standards, for integration into larger subsystems. When test components, such as drivers or stubs, must be created to support testing, the implementer is also responsible for developing and testing the test components and corresponding subsystems.

Integrator. Implementers deliver their tested components into an integration workspace, whereas integrators combine them to produce a build. An integrator is also responsible for planning the integration, which takes place at the subsystem and system levels, with each having a separate integration workspace. Tested components are delivered from an implementer's private development workspace into a subsystem integration workspace, whereas integrated implementation subsystems are delivered from the subsystem integration workspace into the system integration workspace.

Code Reviewer. The code reviewer role ensures the quality of the source code, and plans and conducts source code reviews. The code reviewer is responsible for any review feedback that recommends necessary rework.

A.4.3 Implementation activities

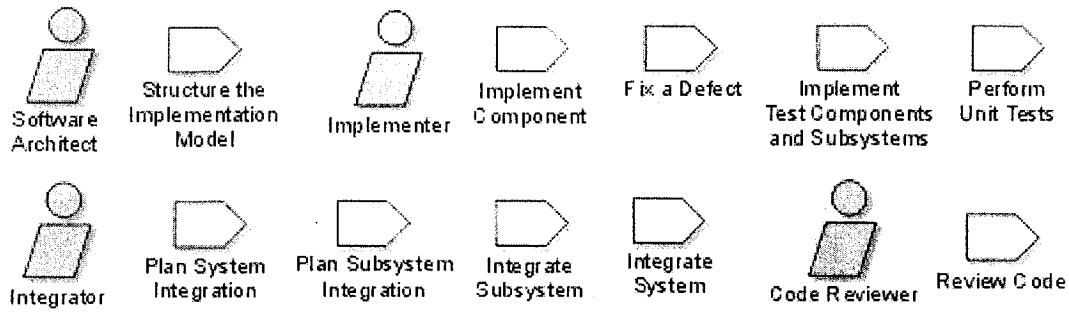


Figure 88: Activities by worker in the *Implementation* discipline

A.4.3.1 Software Architect Activities

Structure the Implementation Model. The purpose is to establish the structure in which the implementation will reside and to assign responsibilities for Implementation Subsystems and their contents.

A.4.3.2 Implementer Activities

Implement Components. The purpose is to implement a design class in a file component. For that, we need to outline the file components that will contain the code that implements the design class. Then we need to generate the source code for the design class and its relationship, operations and attributes. This can be straightforward, because the design class uses the syntax of the programming language. However, it is more delicate to generate code for associations and aggregations. The operations are known as methods. They involve choosing an algorithm and data structures and then coding the algorithm [1].

Fix a Defect. The purpose is to fix a defect.

Implement Test Components and Subsystems. The purpose is to implement test-specific functionality.

Perform Unit Tests. We test the implemented components as individual units. Two types of unit testing are possible: specification testing ("black box testing"), which verifies the unit's externally observable behavior and structure testing ("white box testing"), which verifies the unit's internal implementation.

Specification testing. It looks at what output the component would return when given certain input and when starting in a particular state. We cannot test all possible combinations, but we divide the possible sets of inputs / outputs / states into equivalence classes. It should be enough to test a component for each combination of equivalence classes.

Performing Structure Tests. It is used to verify that the component works internally as intended. All code must be tested. The component engineer should test the most interesting paths through the code (i.e. most commonly followed, critical, least known paths).

A.4.3.3 Integrator Activities

Plan System Integration. The purpose is to plan the integration of the system.

Plan Subsystem Integration. The purpose is to plan the order in which the components contained in an implementation subsystem should be integrated.

Integrate Subsystem. The purpose is to integrate the components in an implementation subsystem, then deliver the implementation subsystem for system integration.

Integrate System. The purpose is to integrate the implementation subsystems piecewise into a build.

A.4.3.4 Code Reviewer Activities

Review Code. The purpose is to verify the source code.

A.4.4 Implementation artifacts

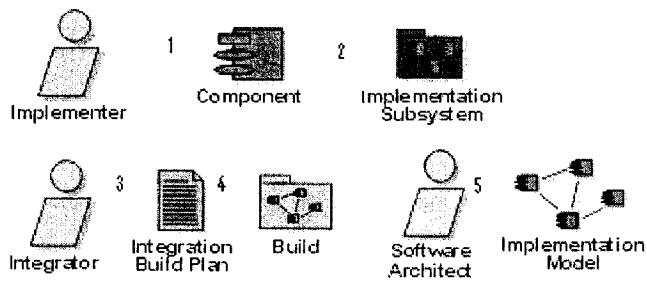


Figure 89: Artifacts produced in the *Implementation* discipline

A.4.4.1 Software Architect Artifacts

Implementation Model. It is a collection of components, and the implementation subsystems that contain them. Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code files.

A.4.4.2 Implementer Artifacts

Component. It represents a piece of software code (source, binary or executable), or a file containing information (for example, a startup file or a ReadMe file). A component can also be an aggregate of other components; for example, an application consisting of several executables. A special type of component is a stub, which is used to develop or test another component. It can be used when integrating the system [1].

Implementation Subsystem. It is a collection of components, interfaces and other implementation subsystems (recursively), and is used to structure the implementation model by dividing it into smaller parts. Implementation subsystems are traced one-to-one to a corresponding design subsystem. The subsystem in implementation provides the same interface as the subsystem in design. Implementation subsystems that trace to service subsystems in design model encapsulate components that provide the various services of the system [1].

A.4.4.3 Integrator Artifacts

Integration Build Plan. It provides a detailed plan for integration within an iteration. It describes the functionality that is expected to be implemented in the build in terms of use cases or scenarios and which parts of the implementation model are affected by the build in terms of subsystems and components [1].

Build. A build is an operational version of a system or part of a system that demonstrates a subset of the capabilities to be provided in the final product. A build comprises one or more components (often executable), each constructed from other components, usually by a process of compilation and linking of source code.

A.4.4.4 Code Reviewer Artifacts

Review Record. It is a form document that is filled out for each review. It is created as a control document to manage the execution of the review of project artifacts. It is issued to the participants in the review to initiate the review process, and is used to capture the results and any action items arising from the review meeting. It forms an auditable record of the review and its conclusions.

A.4.5 Implementation details

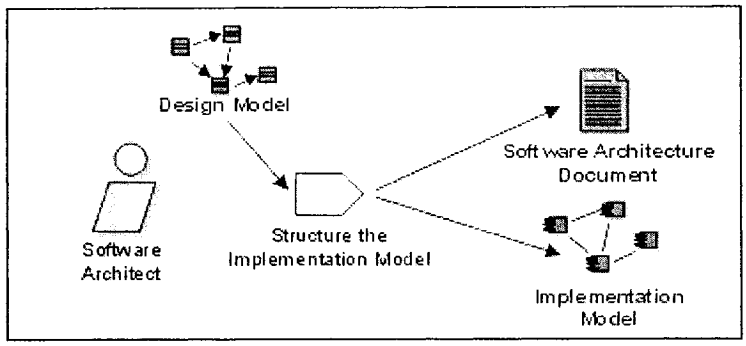


Figure 90: Implementation detail: Structure the implementation model

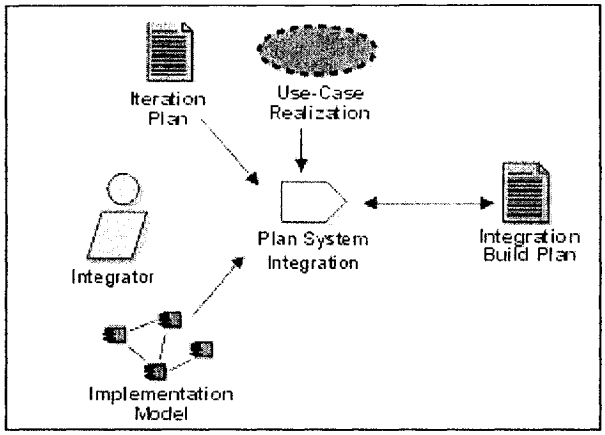


Figure 91: Implementation detail: Plan the integration

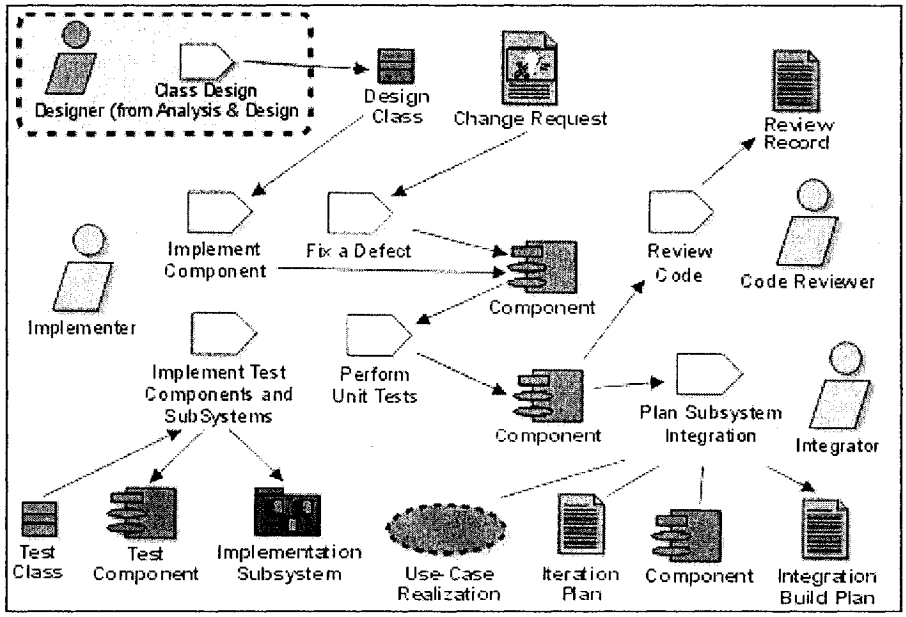


Figure 92: Implementation detail: Implement components

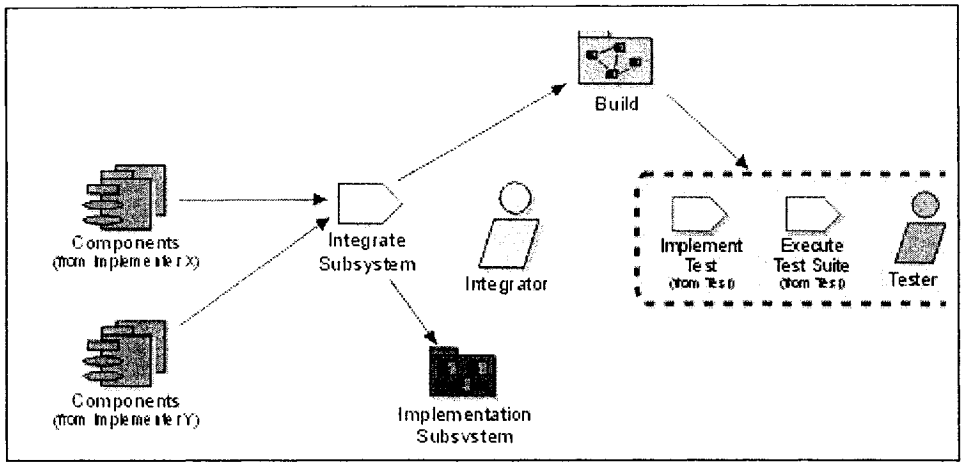


Figure 93: Implementation detail: Integrate each subsystem

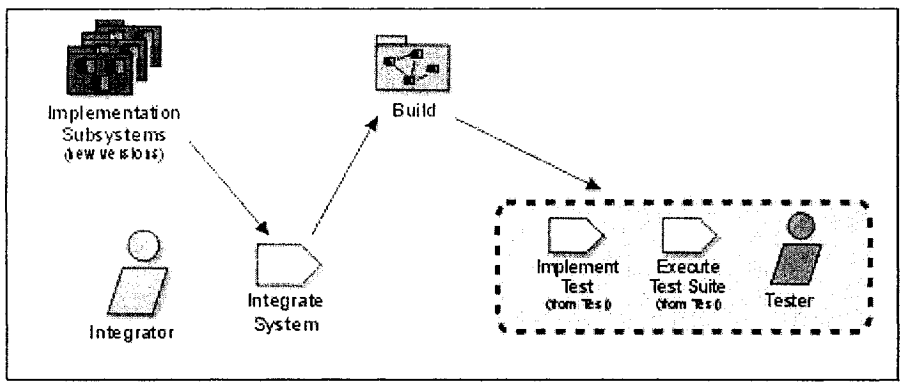


Figure 94: Implementation detail: Integrate the system

A.5 Testing

A.5.1 Testing workflow

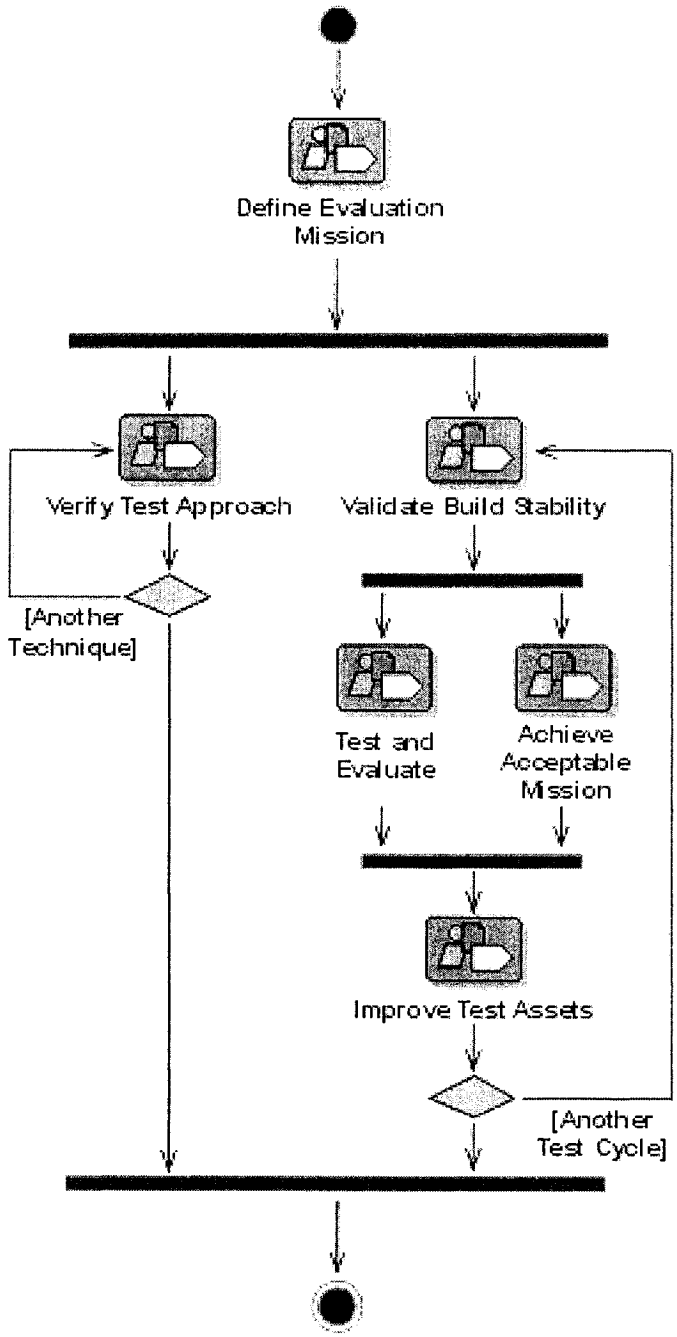


Figure 95: Activity diagram for the *Testing* discipline

A.5.2 Testing workers

Test Manager. He has the overall responsibility for the test effort's success. The role involves quality and test advocacy, resource planning and management, and resolution of issues that impede the test effort. This covers: negotiating the ongoing purpose and deliverables of the test effort, ensuring the appropriate planning and management of the test resources, assessing the progress and effectiveness of the test effort, advocating the appropriate level of quality by the resolution of important defects, advocating an appropriate level of testability focus in the software development process.

Test Analyst. He is responsible for initially identifying and subsequently defining the required tests, monitoring the test coverage and evaluating the overall quality experienced when testing the Target Test Items. This role also involves specifying the required Test Data and evaluating the outcome of the testing conducted in each test cycle. This role is responsible for: identifying the Target Test Items to be evaluated by the test effort, defining the appropriate tests required and any associated Test Data, gathering and managing the Test Data, evaluating the outcome of each test cycle.

Test Designer. He is responsible for defining the test approach and ensuring its successful implementation. The role involves identifying the appropriate techniques, tools and guidelines to implement the required tests, and to give guidance on the corresponding resources requirements for the test effort. This role is responsible for identifying and describing appropriate test techniques, identifying the appropriate supporting tools, defining and maintaining a Test Automation Architecture, specifying and verifying the required Test Environment Configurations, verifying and assessing the Test Approach.

Tester. He is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing. This

covers: identifying the most appropriate implementation approach for a given test, implementing individual tests, setting up and executing the tests, logging outcomes and verifying test execution, analyzing and recovering from execution errors.

A.5.3 Testing activities

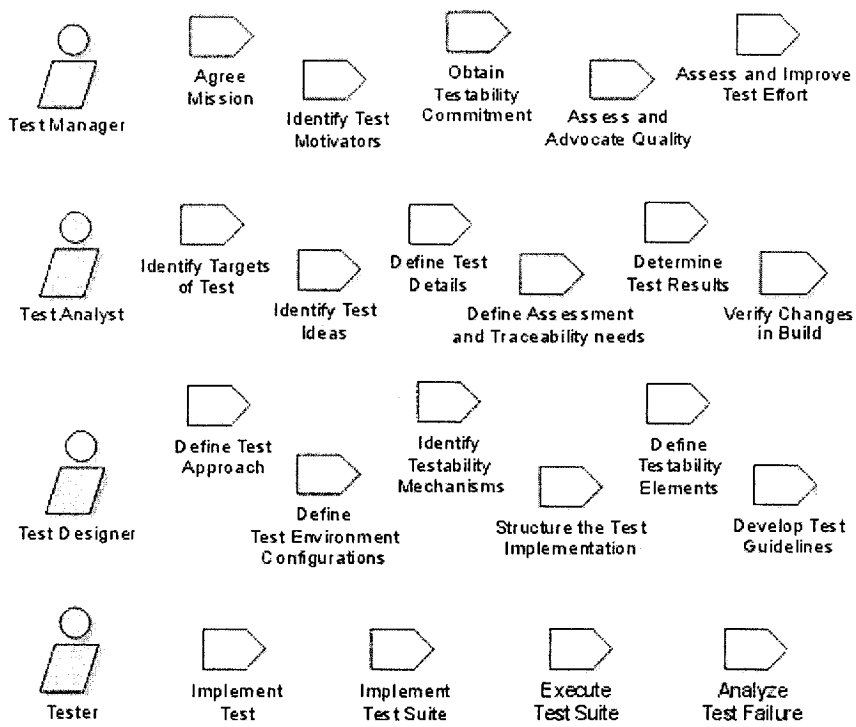


Figure 96: Activities by worker in the *Testing* discipline

A.5.3.1 Test Manager Activities

Agree Mission. The purpose is to negotiate the most effective use of testing resources for each iteration and to agree on an appropriate and achievable set of objectives and deliverables for the iteration.

Identify Test Motivators. The purpose is to identify the specific list of things, including both events and artifacts, that will serve to motivate testing in this iteration.

Obtain Testability Commitment. The purpose is to promote the creation of testable software that supports the needs of the test effort, promote and support the use of appropriate automation techniques and tools.

Assess and Advocate Quality. The purpose is to identify and advocate the resolution of defects that have a serious detrimental impact on software quality or which prevent or impair the testing effort, by monitoring the progress of and support the appropriate completion of changes that improve software quality to the required level.

Assess and Improve Test Effort. The purpose is to make an assessment of the productivity, effectiveness and completeness of the test effort and to make adjustments to the test effort (both tactical and strategic) to improve effectiveness.

A.5.3.2 Test Analyst Activities

Identify Targets of Test. The purpose is to identify the individual system elements, both hardware and software, that need to be tested.

Identify Test Ideas. The purpose is to identify the test ideas that should be explored to assess acceptable quality of the Target Test Items and to identify a sufficient number of ideas to adequately validate Target Test Items against Test Motivators.

Define Test Details. The purpose is to define the individual conditions necessary to realize a test idea in a specific context, identify potential points of observation

and control for the related test item(s) and potential oracles to facilitate observation points and provide consumable resources to support the test.

Define Assessment and Traceability needs. The purpose is to define the assessment strategy for the test effort and the traceability and coverage requirements

Determine Test Results. The purpose is to make ongoing summary evaluations of the perceived quality of the product, determine the detailed Test Results and propose appropriate corrective actions to resolve failures in quality.

Verify Changes in Build. This activity confirms that a Change Request has been completed, typically by conducting subset of tests on one or more builds.

A.5.3.3 Test Designer Activities

Define Test Approach. The purpose is to identify each specific technique that will be employed to enable the desired testing, outline the workings of each technique including the types of testing it supports and define a candidate architecture for the test automation system.

Define Test Environment Configurations. The purpose is to define the requirements for the evaluation environment(s) needed to support the test effort.

Identify Testability Mechanisms. The purpose is to identify the general mechanisms of the technical solution needed to facilitate the test approach and outline the general scope and key characteristics of those mechanisms.

Structure the Test Implementation. The purpose is to establish the structure in which the test suite implementation will reside, assign responsibilities for test suite implementation areas and their contents and outline the required Test Suites.

Define Testability Elements. The purpose is to identify the physical elements of the test implementation infrastructure required to enable testing under each Test Environment Configuration and to define the software design requirements that will need to be met to enable the software to be physically testable.

Develop Test Guidelines. The purpose is to make tactical adjustments to the way the process is enacted and record those decisions, capture project-specific practices discovered during the dynamic enactment of the process and develop an understanding of the strengths and weaknesses of the "micro-process" as it emerges.

A.5.3.4 Tester Activities

Implement Test. The purpose is to implement meaningful tests that provide the required product validation and develop tests that operate as part of a larger test infrastructure.

Implement Test Suite. The purpose is to arrange or assemble collections of tests to be executed together in a valuable way and facilitate both breadth and depth of the test effort by exercising as many required test combinations as possible.

Execute Test Suite. The purpose is to execute the appropriate collections of tests required to validate the product quality and capture test results that facilitate the assessment of the product

Analyze Test Failure. The purpose is to investigate the Test Log details and analyze the failures that occurred during test implementation and execution, correct them and record important findings appropriately.

A.5.4 Testing artifacts

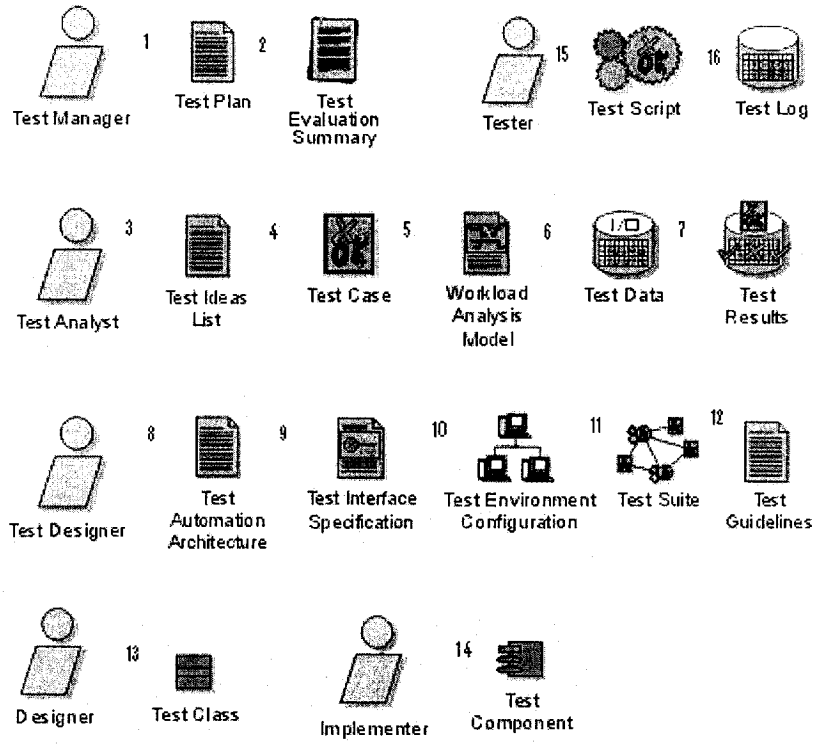


Figure 97: Artifacts produced in the *Testing* discipline

A.5.4.1 Test Manager Artifacts

Test Plan. It contains the definition of the goals and objectives of testing within the scope of the iteration (or project), the items being targeted, the approach to be taken, the resources required and the deliverables to be produced.

Test Evaluation Summary. It organizes and presents a summary analysis of the Test Results and key measures of test for review and assessment, typically by key quality stakeholders. In addition, the Test Evaluation Summary may contain a general statement of relative quality and provide recommendations for future test effort.

A.5.4.2 Test Analyst Artifacts

Test Ideas List. It is an enumerated list of ideas, often partially formed, that identify potentially useful tests to conduct.

Test Case. It contains the definition (usually formal) of a specific set of test inputs, execution conditions, and expected results, identified for the purpose of making an evaluation of some particular aspect of a Target Test Item.

Workload Analysis Model. It is a model that identifies one or more workload profiles that are deemed to accurately define a system state of interest in which evaluation of the software and/ or it's operating environment can be undertaken. The workload profiles represent candidate conditions to be simulated against the Target Test Items under one or more Test Environment Configurations.

Test Data. It contains the definition (usually formal) of a collection of test input values that are consumed during the execution of a test, and expected results referenced for comparative purposes during the execution of a test.

Test Results. It contains a collection of summary information determined from the analysis of one or more Test Logs and Change Requests, providing a relatively detailed assessment of the quality of the Target Test Items and the status of the test effort.

A.5.4.3 Test Designer Artifacts

Test Automation Architecture. It represents a composition of various test automation elements and their specifications that embody the fundamental characteristics of the test automation software system. The Test Automation Architecture provides a comprehensive architectural overview of the test

automation system, using a number of different architectural views to depict different aspects of the system.

Test Interface Specification. It is a specification for the provision of a set of behaviors (operations) by a classifier (specifically, a Class, Subsystem or Component) for the purposes of test access (testability). Each test Interface should provide a unique and well-defined group of services.

Test Environment Configuration. It is a specific arrangement of hardware, software, and the associated environment settings required to conduct accurate tests that enable the evaluation of the Target Test Items.

Test Suite. It is a package-like artifact used to group collections of Test Scripts, both to sequence the execution of the tests and to provide a useful and related set of Test Log information from which Test Results can be determined.

Test Guidelines. It is a documented record of any of the following: process control and enactment decisions, standards to be adhered to, or good-practice guidance generally to be followed by the practitioners on a given project.

A.5.4.4 Tester Artifacts

Test Script. It consists of the step-by-step instructions that realize a test, enabling its execution. Test Scripts may take the form of either documented textual instructions that are executed manually or computer readable instructions that enable automated test execution.

Test Log. It is a collection of raw output captured during a unique execution of one or more tests, usually representing the output resulting from the execution of a Test Suite for a single test cycle run.

A.5.5 Testing details

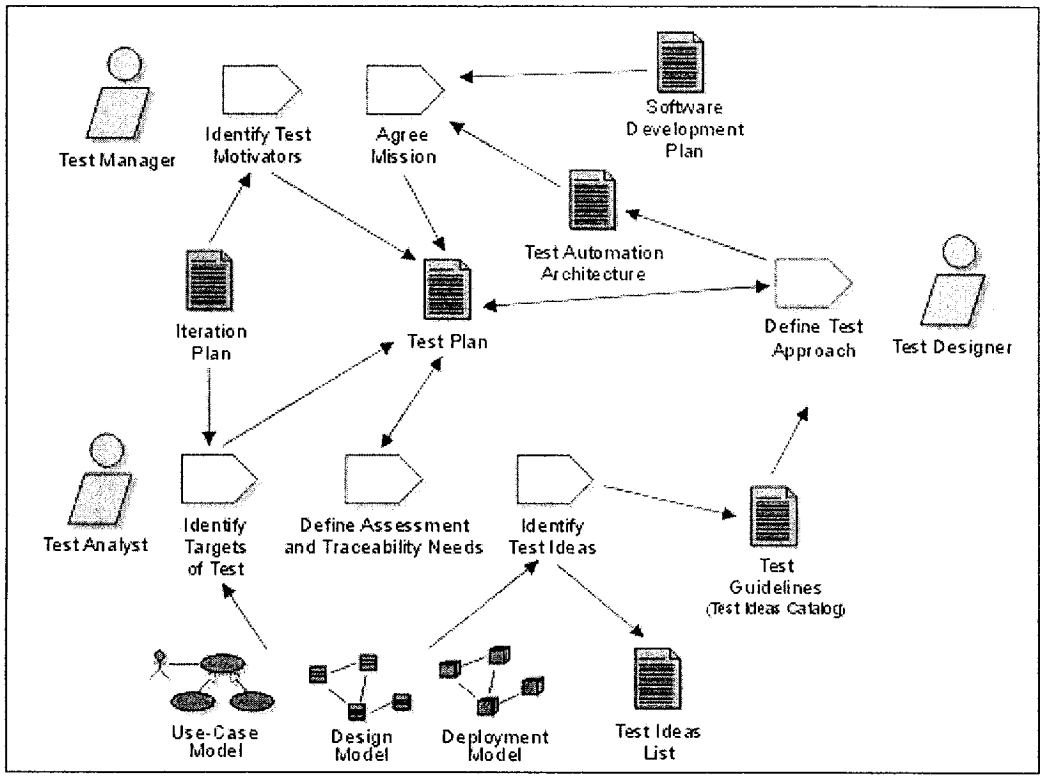


Figure 98: Testing detail: Define Evaluation Mission

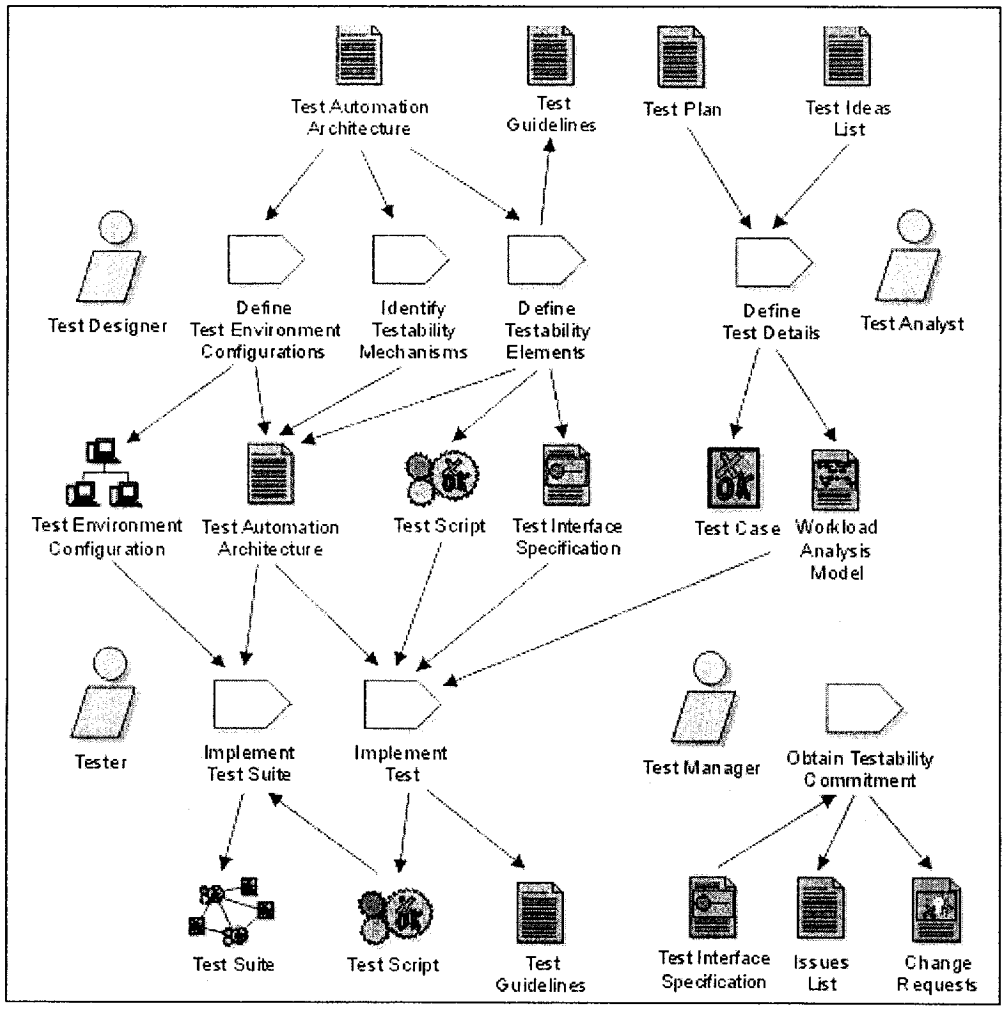


Figure 99: Testing detail: Verify Test approach

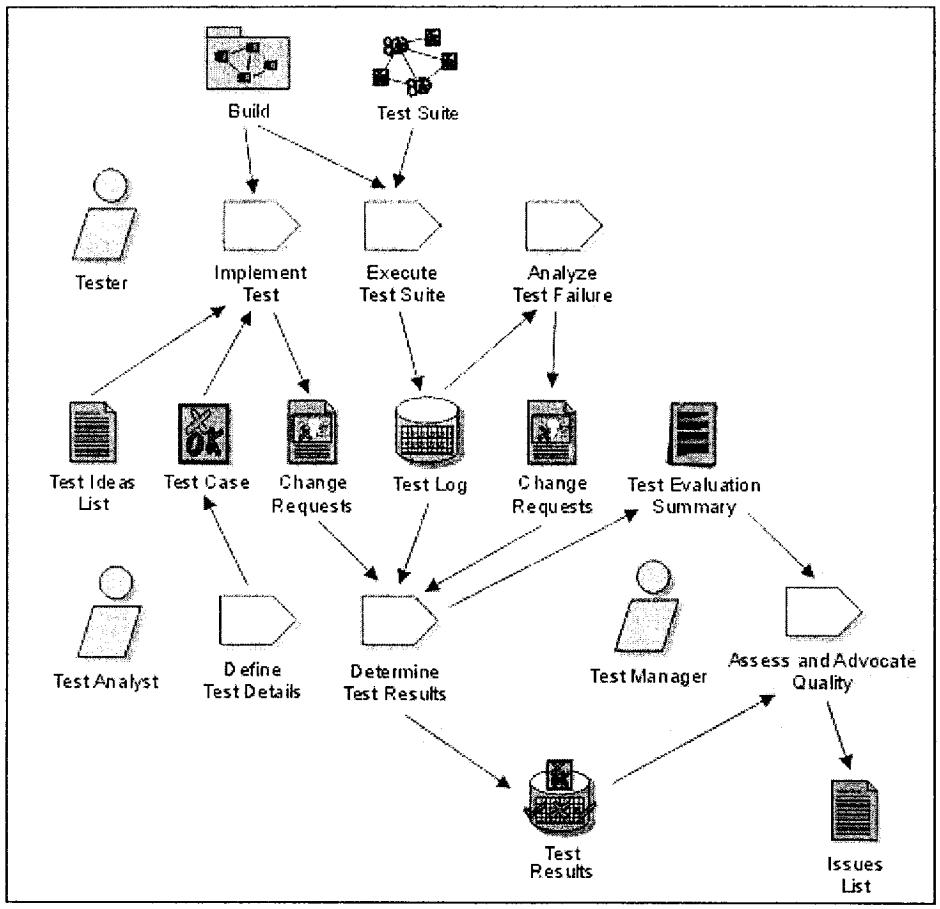


Figure 100: Testing detail: Validate Build Stability

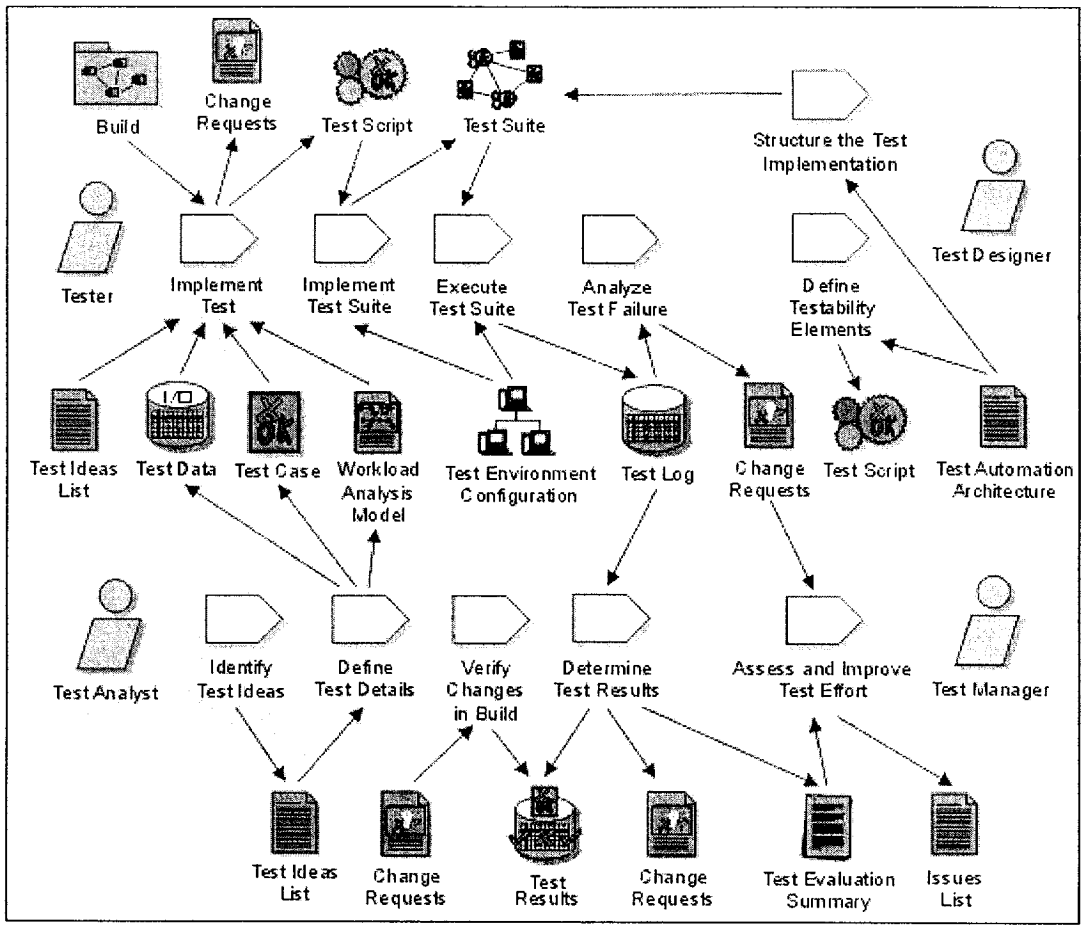


Figure 101: Testing detail: Test and Evaluate

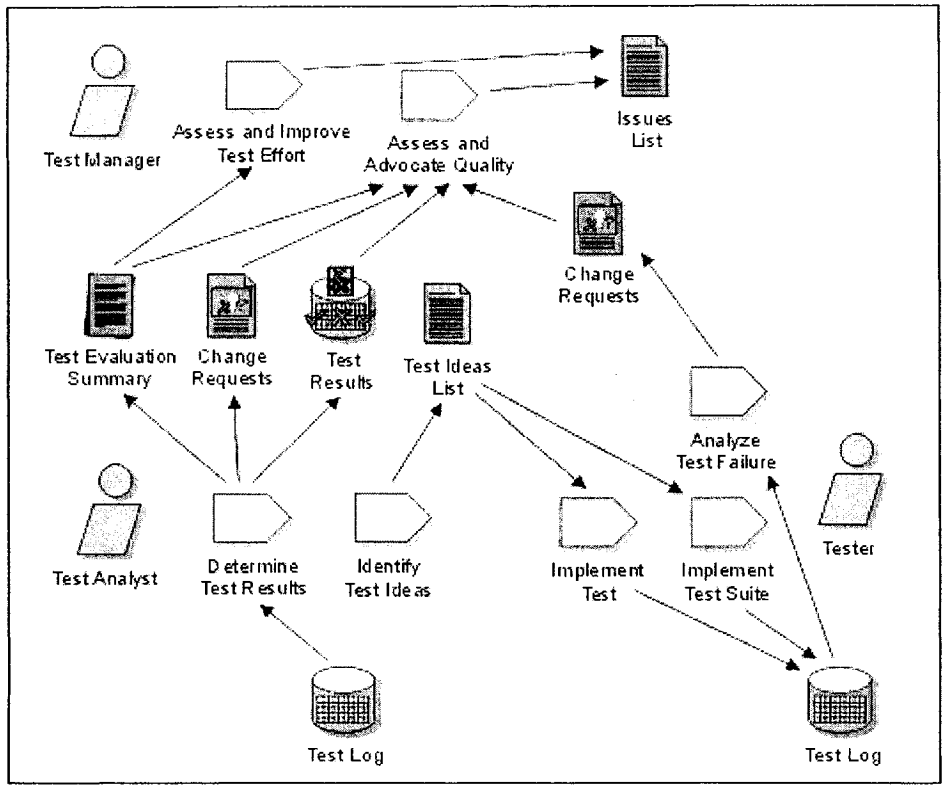


Figure 102: Testing detail: Achieve acceptable mission

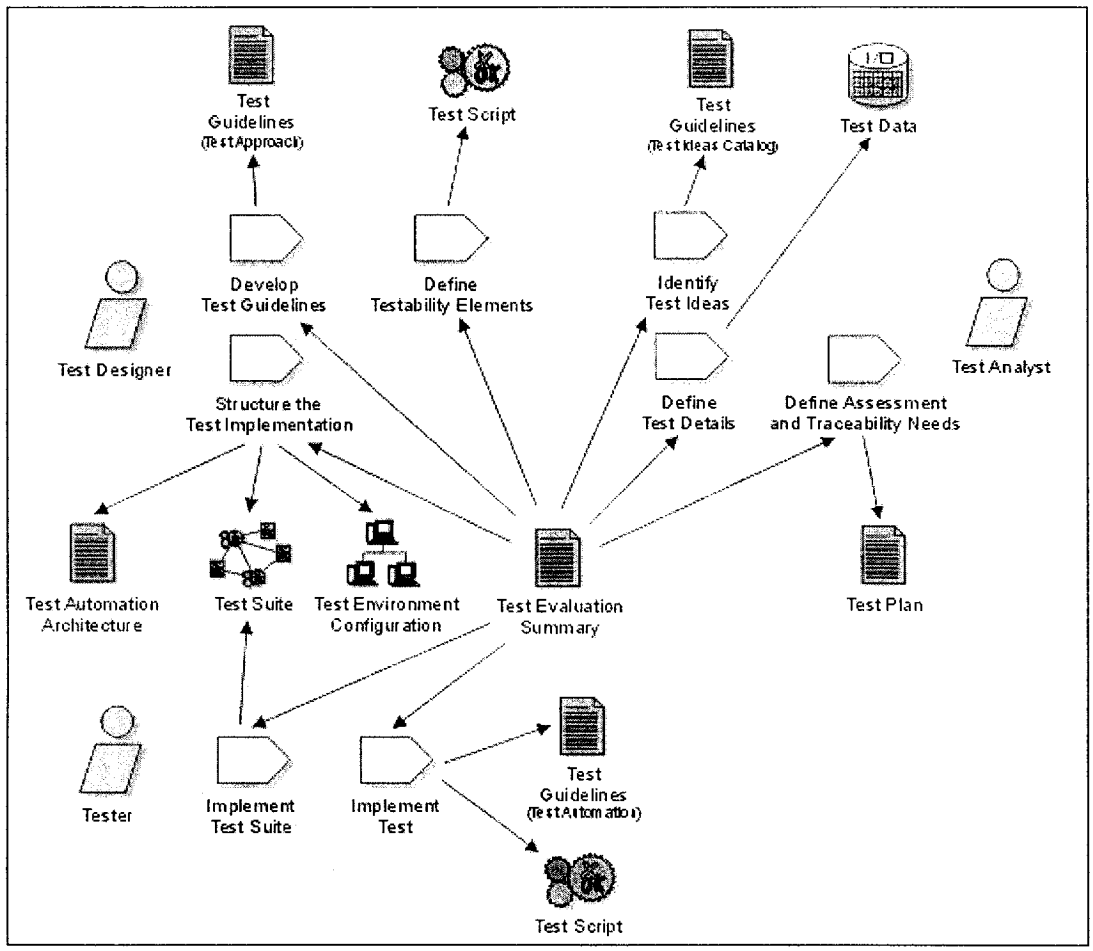


Figure 103: Testing detail: Improve test assets

A.6 Deployment

A.6.1 Deployment workflow

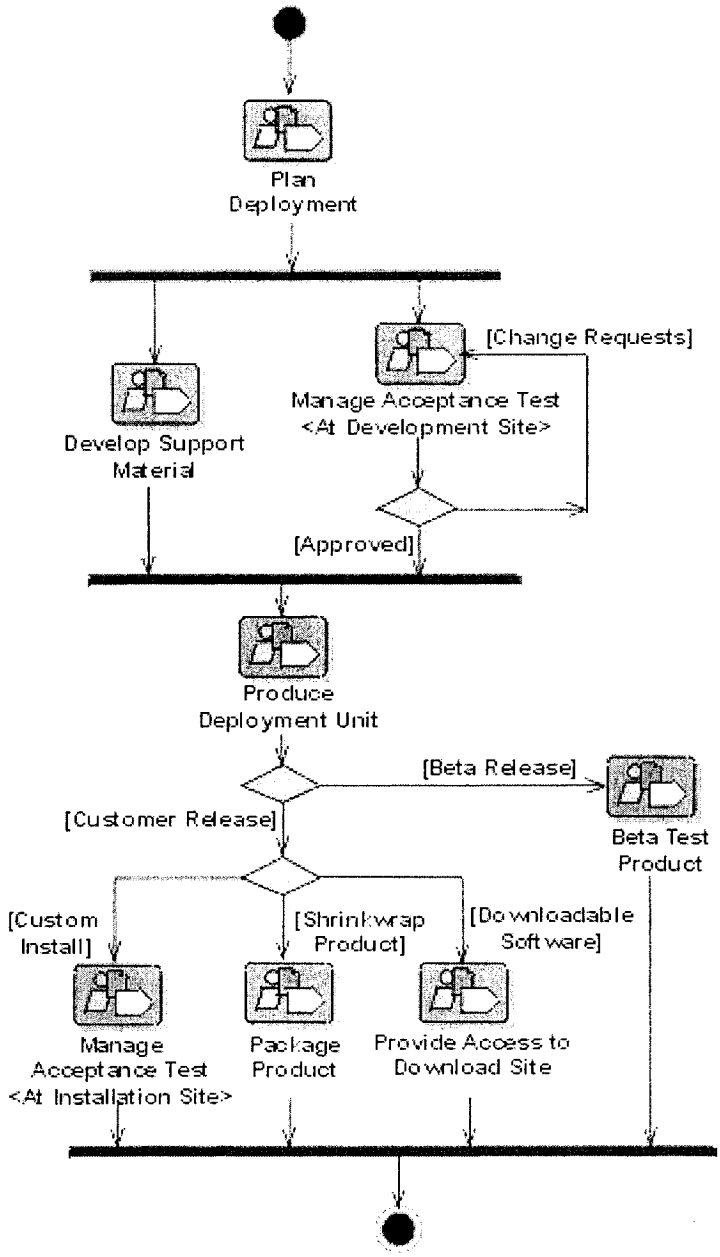


Figure 104: Activity diagram for the *Deployment* discipline

A.6.2 Deployment workers

Deployment Manager. He plans the product's transition to the user community and documents it in various associated documents.

Course Developer. He develops training material to teach users how to use the product. This includes creating slides, student notes, examples, tutorials, and so on, to enhance users' understanding of the product.

Implementer. He is responsible for developing and testing components, in accordance with the project's adopted standards, for integration into larger subsystems. When test components, such as drivers or stubs, must be created to support testing, the implementer is also responsible for developing and testing the test components and corresponding subsystems.

Graphic Artist. He creates product artwork that is included as part of the product packaging.

Technical Writer. He produces end-user support material such as user guides, help texts, release notes, and so on.

Configuration Manager. He provides the overall Configuration Management (CM) infrastructure and environment to the product development team. The CM function supports the product development activity so that developers and integrators have appropriate workspaces to build and test their work, and so that all artifacts are available for inclusion in the deployment unit as required. The configuration manager also has to ensure that the CM environment facilitates product review, and change and defect tracking activities. The configuration manager is also responsible for writing the CM Plan and reporting progress statistics based on change requests.

A.6.3 Deployment activities

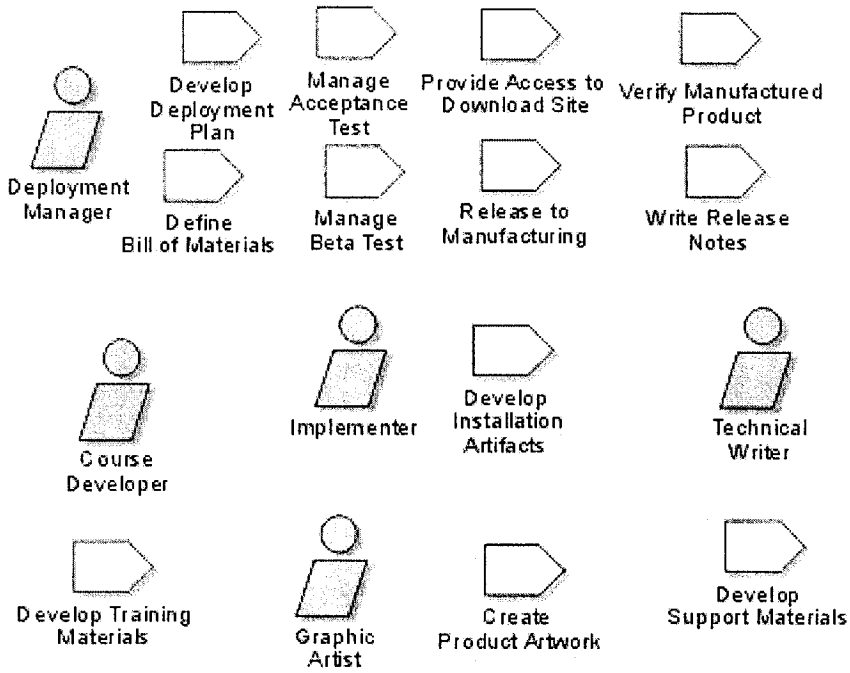


Figure 105: Activities by worker in the *Deployment* discipline

A.6.3.1 Deployment Manager Activities

Develop Deployment Plan. The end-user's willingness to use the product is the mark of its success. The Deployment plan documents show how and when the product is to be made available to the user community.

Manage Acceptance Test. The purpose is to ensure that the developed product fulfills its acceptance criteria at both the development, and target installation site.

Provide Access to Download Site. The purpose is to ensure that the product is available for purchase, and download over the internet.

Verify Manufactured Product. The purpose is to ensure that the manufactured product is complete, and useable. This activity is sometimes referred to as the

'first article inspection'. It serves as a quality control activity to ensure that the retailed product has all the required attributes and artifacts.

Define Bill of Materials. The purpose is to create a complete list of artifacts that go to make up the build/product. The list includes software configuration items, documents and installation scripts. In the case of packaged products, the Bill of Materials will need to identify the pieces of artwork and packaging items that make up the final product.

Manage Beta Test. Beta testing is "pre-release" testing in which a sampling of the intended audience tries out the product. Beta testing serves two purposes. Firstly it gives the product a controlled "real-world" test, and secondly it provides a preview of the next release. The Deployment Manager needs to manage the product's beta test program to ensure that both these purposes are served.

Release to Manufacturing. The purpose is to mass produce the 'shrink wrap' version of the product.

Write Release Notes. The purpose is to describe the major new features and changes in the release. The Release Notes should also describe any known bugs and limitations or workarounds to using the product.

A.6.3.2 Course Developer Activities

Develop Training Materials. He produces the material needed to train the users of the product.

A.6.3.3 Implementer Activities

Develop Installation Artifacts. He produces all the software required to install and uninstall the product quickly, easily and safely without affecting other applications or system characteristics.

A.6.3.4 Graphic Artist Activities

Create Product Artwork. The purpose of creating product artwork is to publish it either as hardcopy directly on the product packaging or in softcopy for the web. Artwork should reinforce product branding standards, established by the company's marketing group, to create the appropriate messaging for the consumer.

A.6.3.5 Technical Writer Activities

Develop Support Materials. The purpose is to develop the end-user support material.

A.6.3.6 Configuration Manager Activities

Create Deployment Unit. A deployment unit consists of a build (an executable collection of components), documents (end-user support material and release notes) and installation artifacts. The purpose of this activity is to create a deployment unit that is sufficiently complete to be downloadable, installable and run on a node as a group.

A.6.4 Deployment artifacts

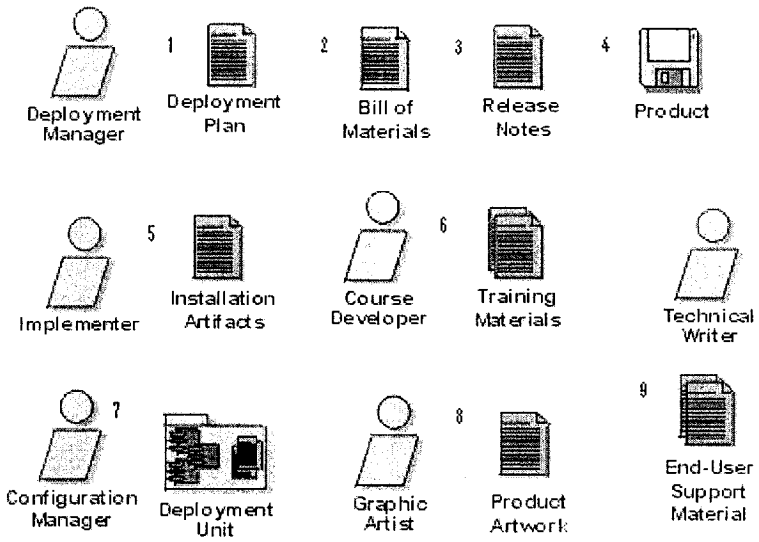


Figure 106: Artifacts produced in the *Deployment* discipline

A.6.4.1 Deployment Manager Artifacts

Deployment Plan. It describes the set of tasks necessary to install and test the developed product such that it can be effectively transitioned to the user community.

Bill of Materials. It lists the constituent parts of a given version of a product, and where the physical parts may be found. It describes the changes made in the version, and refers to how the product may be installed.

Release Notes. They identify changes and known bugs in a version of a build or deployment unit that has been made available for use.

Product. The packaging of a **product** for market appeal distinguishes it from a deployment unit. A product can contain multiple deployment units, and may be accessible as a downloadable commodity, in shrink wrap or on any digital storage media formats.

A.6.4.2 Course Developer Artifacts

Training Materials. They refer to the material that is used in training programs or courses to assist the end-users with product use, operation and/or maintenance.

A.6.4.3 Implementer Artifacts

Installation Artifacts. They refer to the software and documented instructions required to install the product.

A.6.4.4 Graphic Artist Artifacts

Product Artwork. It includes the text (print specifications) and artwork that will be used to 'brand' the product. The Product Artwork may appear on physical packaging or on a web site.

A.6.4.5 Technical Writer Artifacts

End-User Support Material. It contains the materials that assist the end-user in learning, using, operating and maintaining the product.

A.6.4.6 Configuration Manager Artifacts

Deployment Unit. It consists of a build (an executable collection of components), documents (end-user support material and release notes) and installation artifacts. A deployment unit is typically associated with a single node in the overall network of computer systems or peripherals.

A.6.5 Deployment details

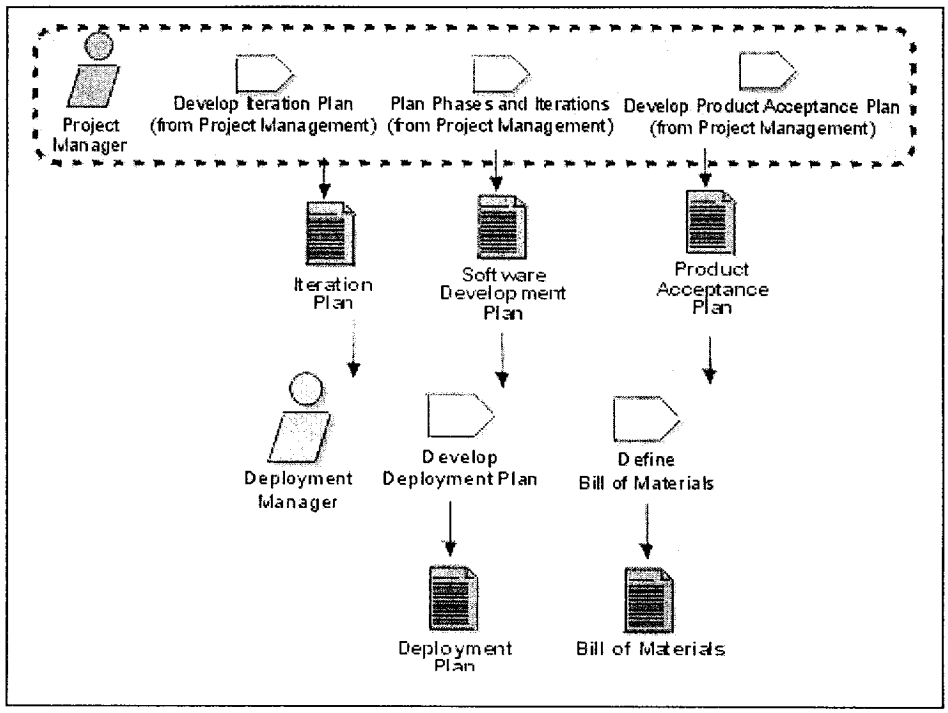


Figure 107: Deployment detail: Plan deployment

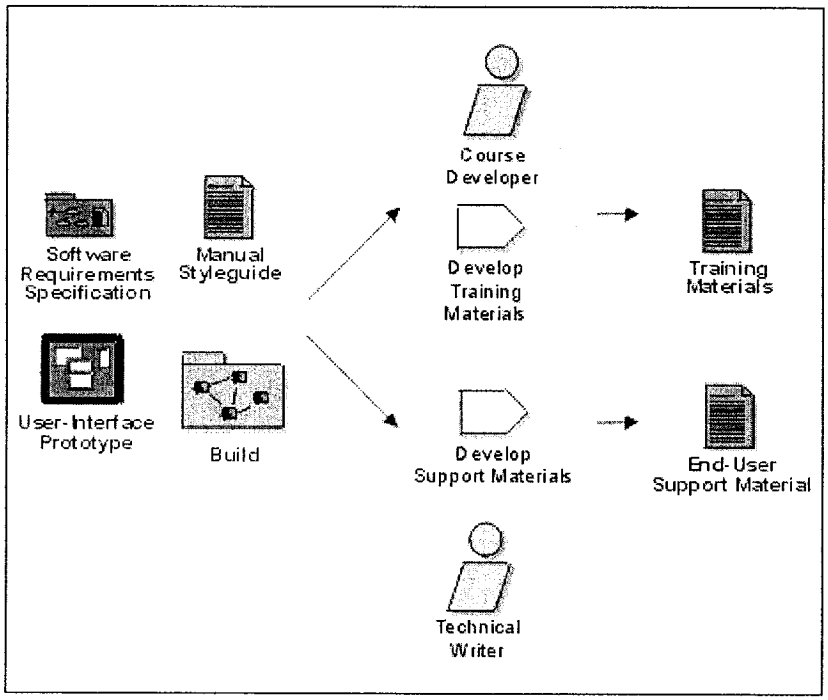


Figure 108: Deployment detail: Develop support material

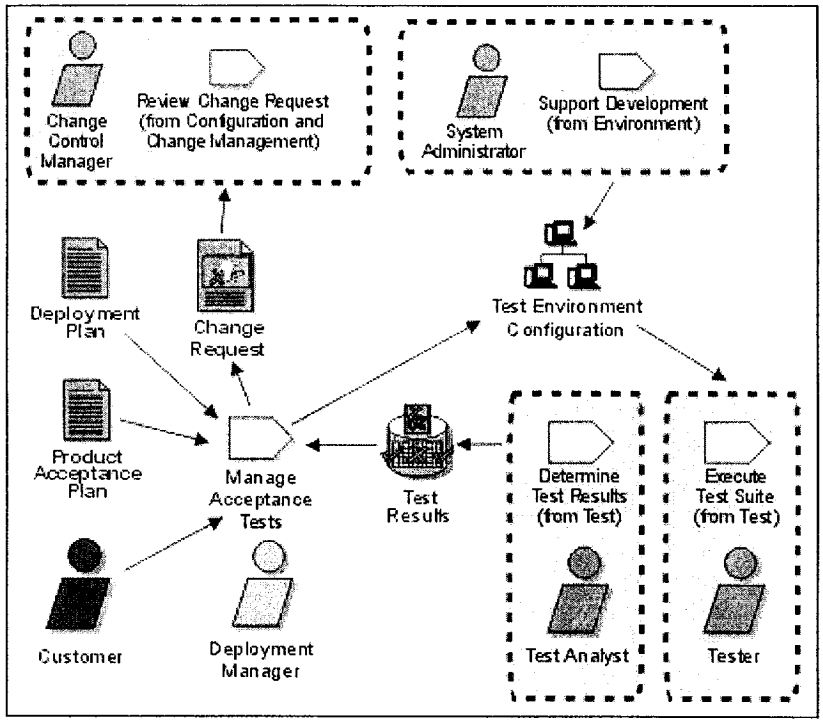


Figure 109: Deployment detail: Manage Acceptance Test

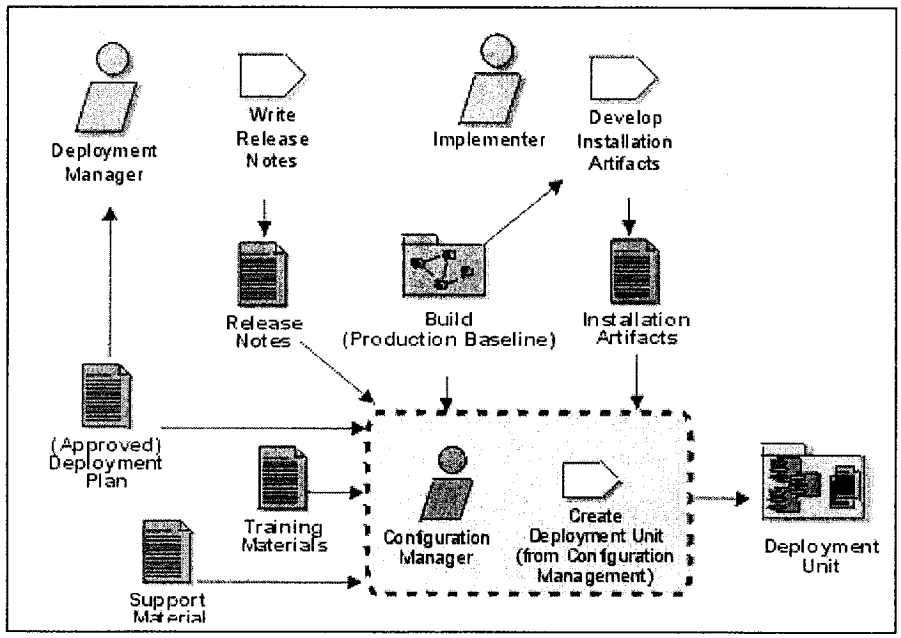


Figure 110: Deployment detail: Produce Deployment Unit

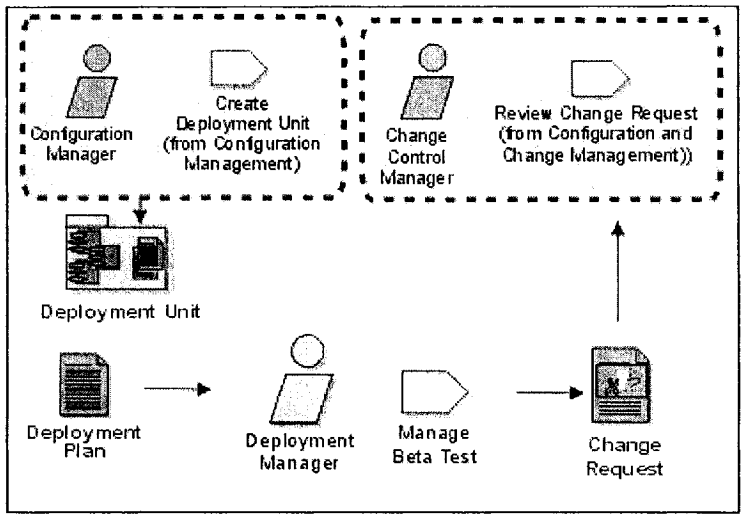


Figure 111: Deployment detail: Beta Test Product

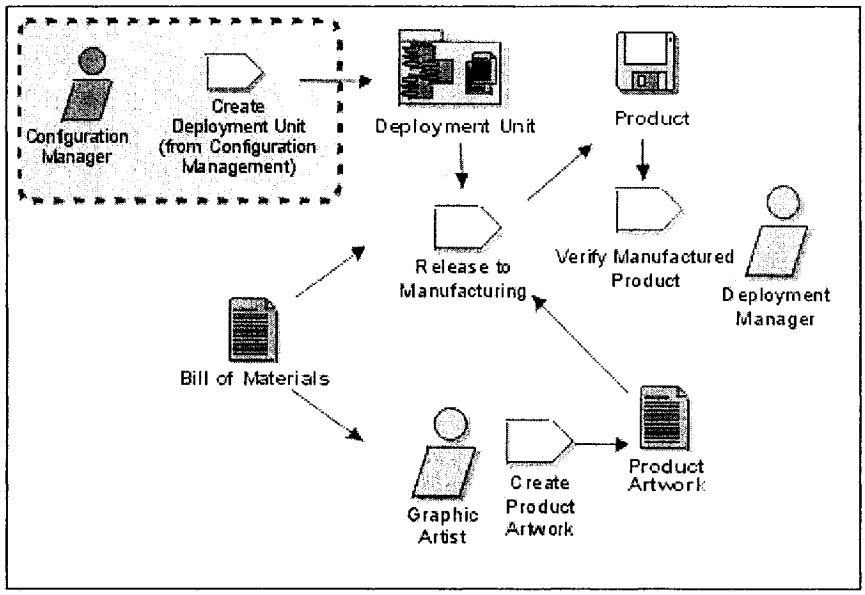


Figure 112: Deployment detail: Package product

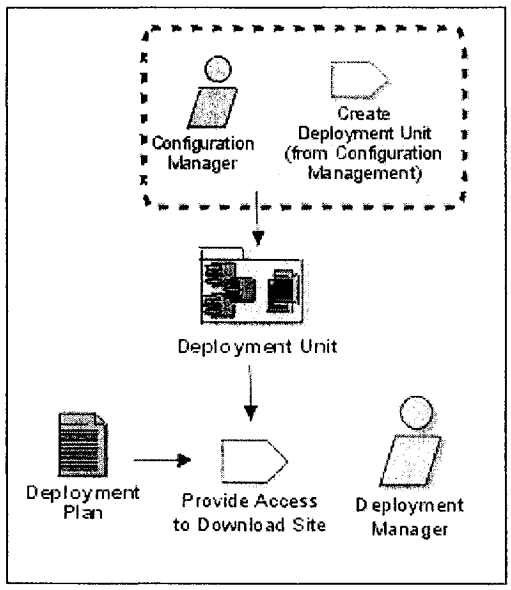


Figure 113: Deployment detail: Provide access to download site

A.7 Activity Diagrams

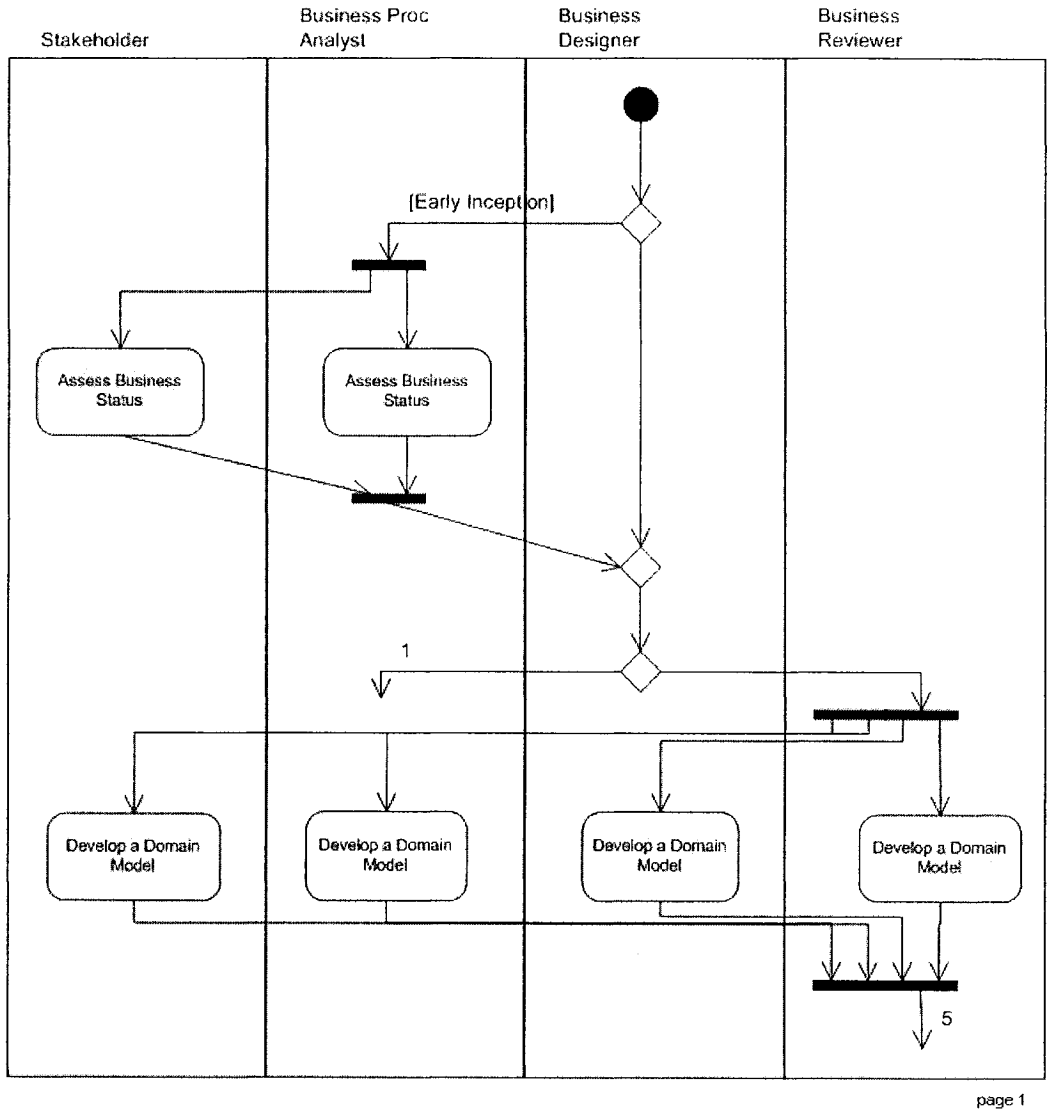


Figure 114: Activity diagram for the *Business Modeling* discipline including roles

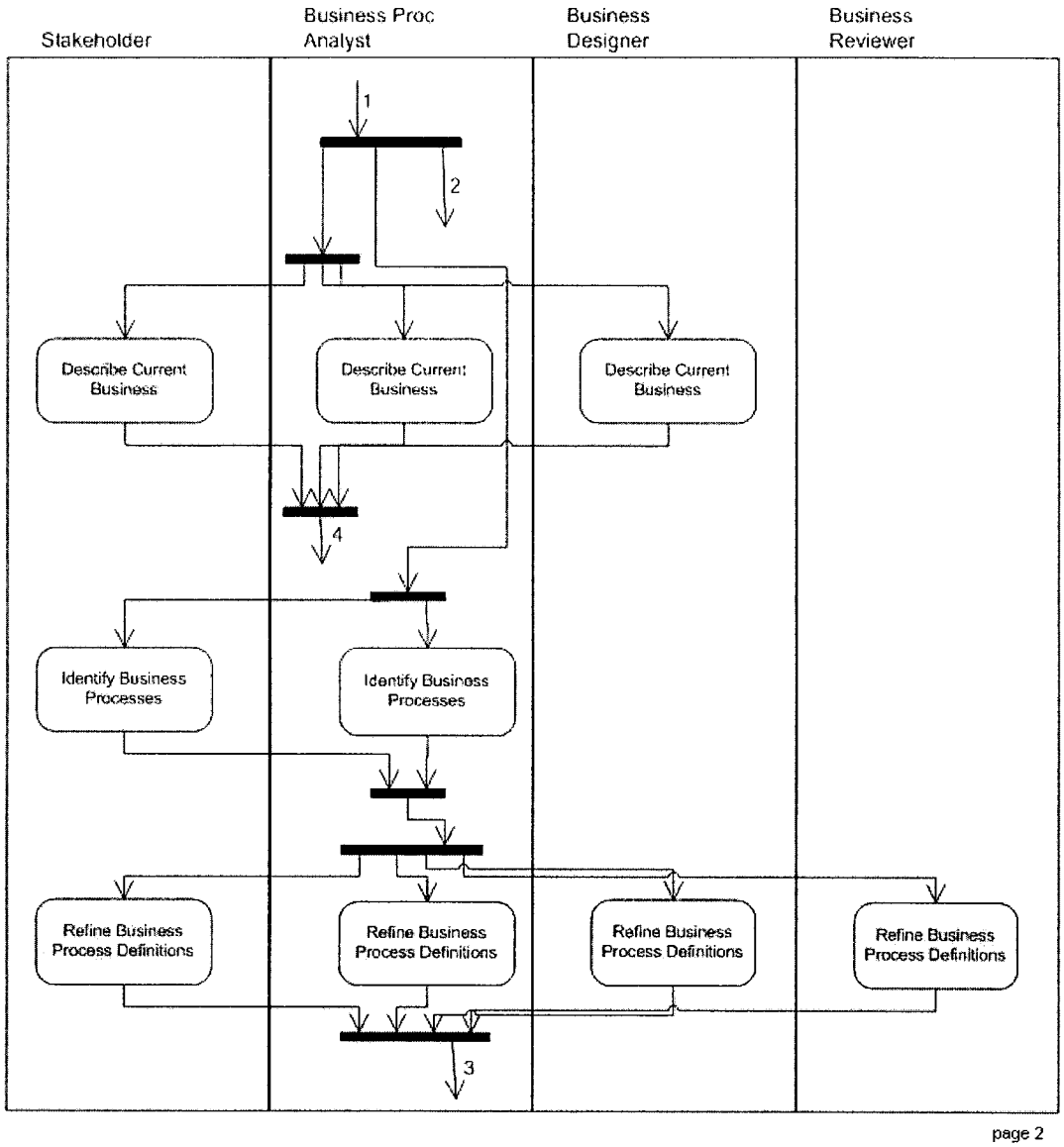


Figure 115: Activity diagram for the *Business Modeling* discipline including roles (contd.)

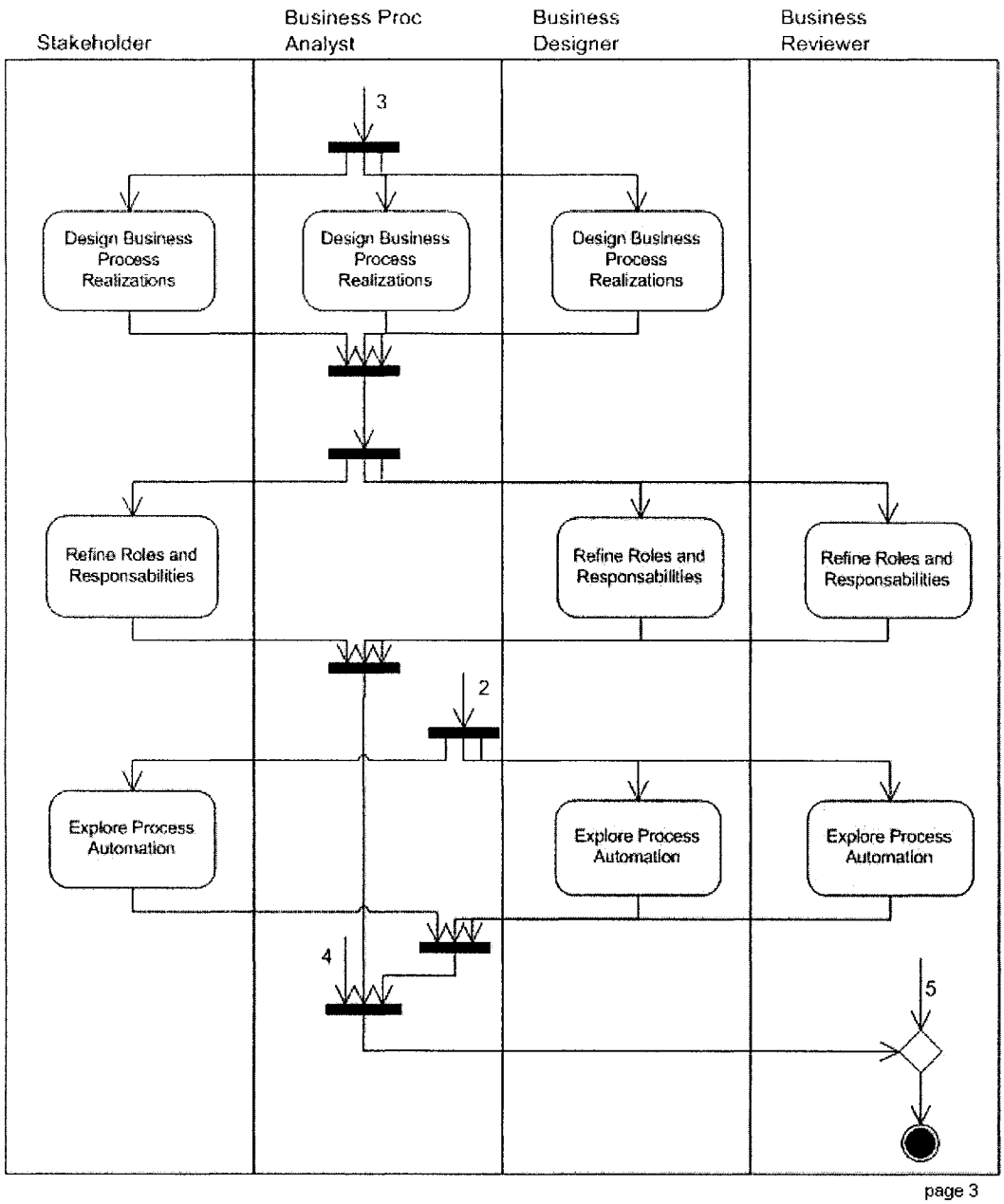


Figure 116: Activity diagram for the *Business Modeling* discipline including roles (contd.)

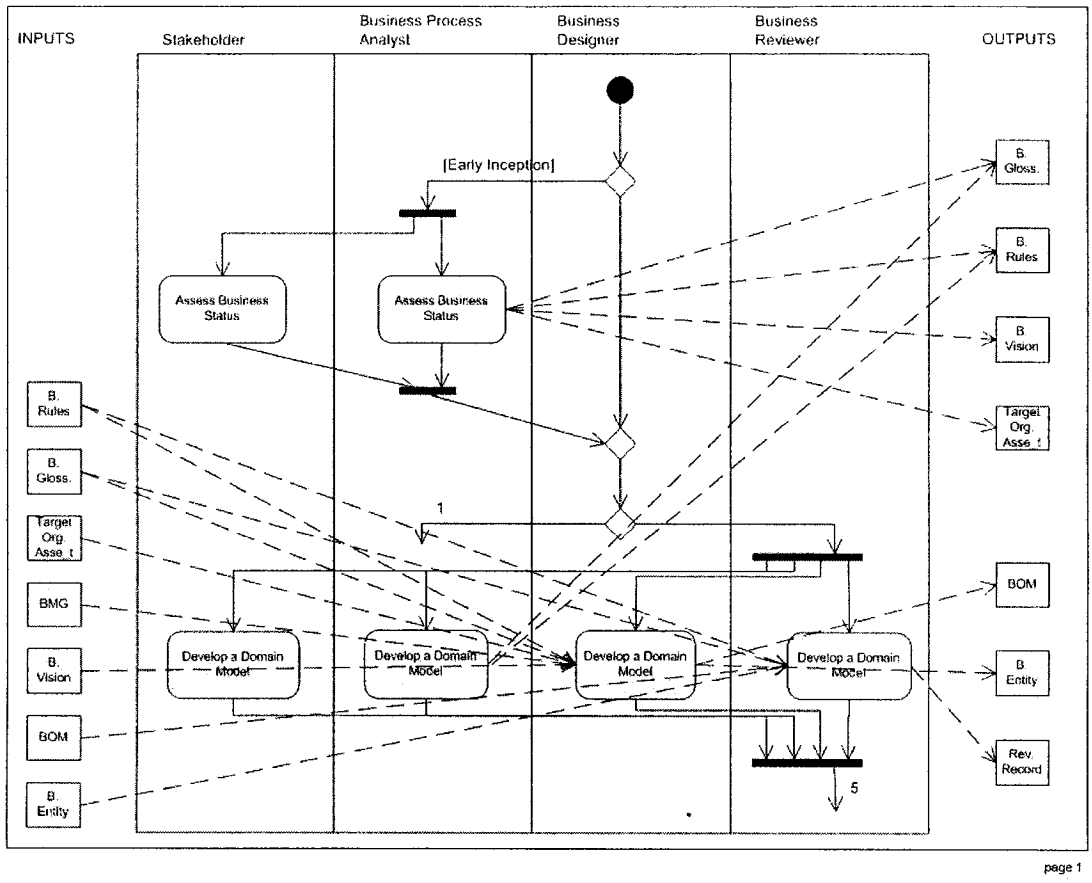


Figure 117: Activity diagram for the *Business Modeling* discipline including roles and artifacts

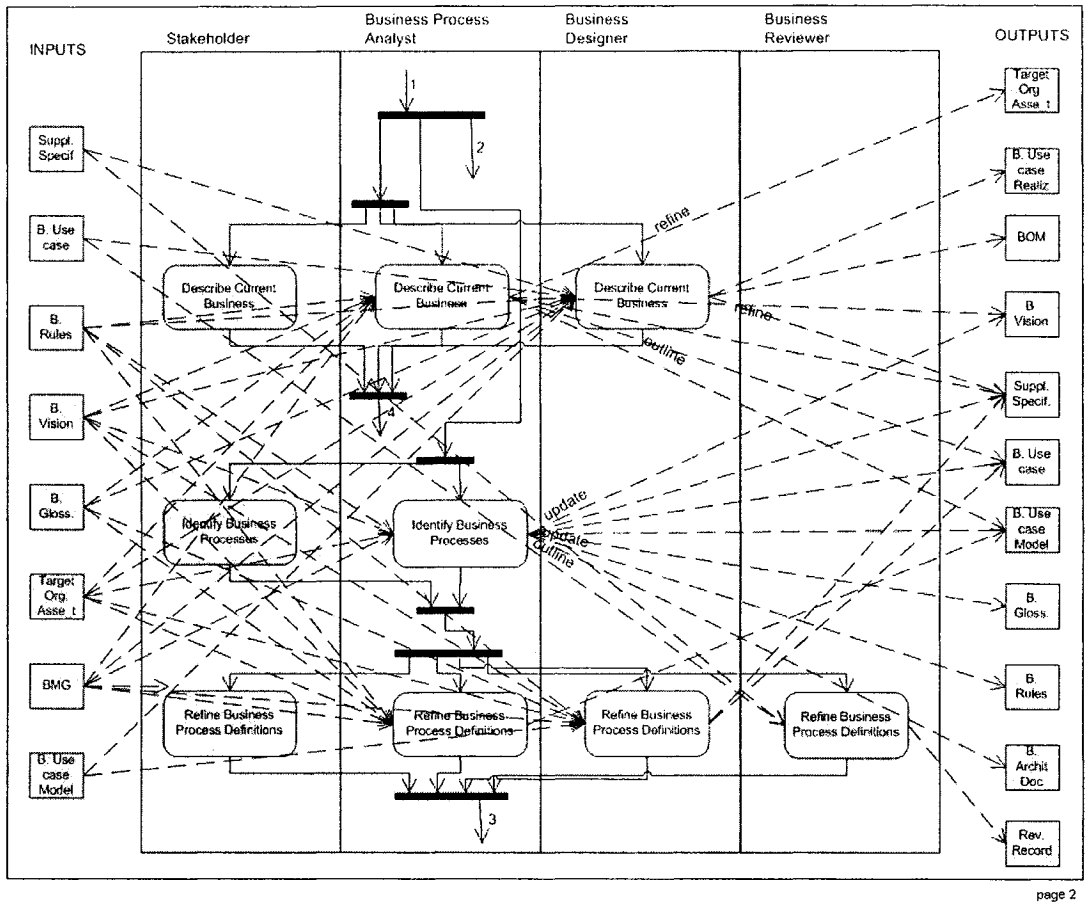


Figure 118: Activity diagram for the *Business Modeling* discipline including roles and artifacts (contd.)

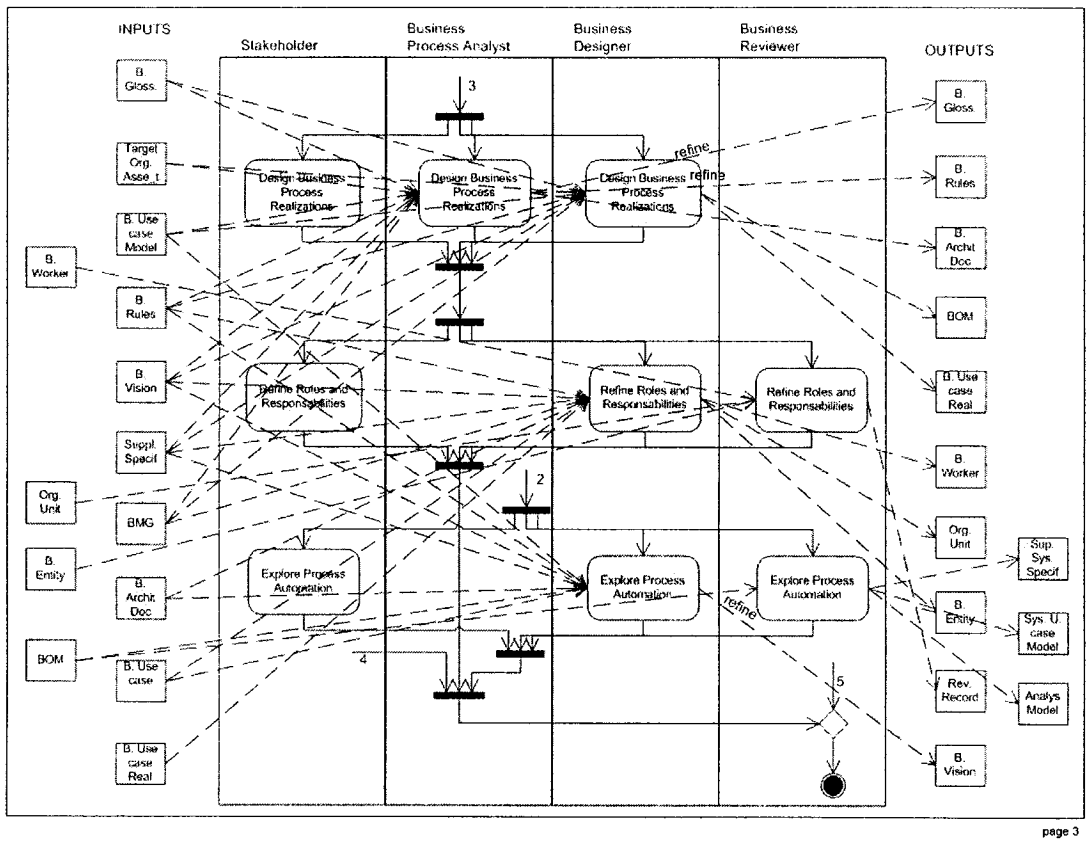


Figure 119: Activity diagram for the *Business Modeling* discipline including roles and artifacts (contd.)

A.8 RADs and XRADs

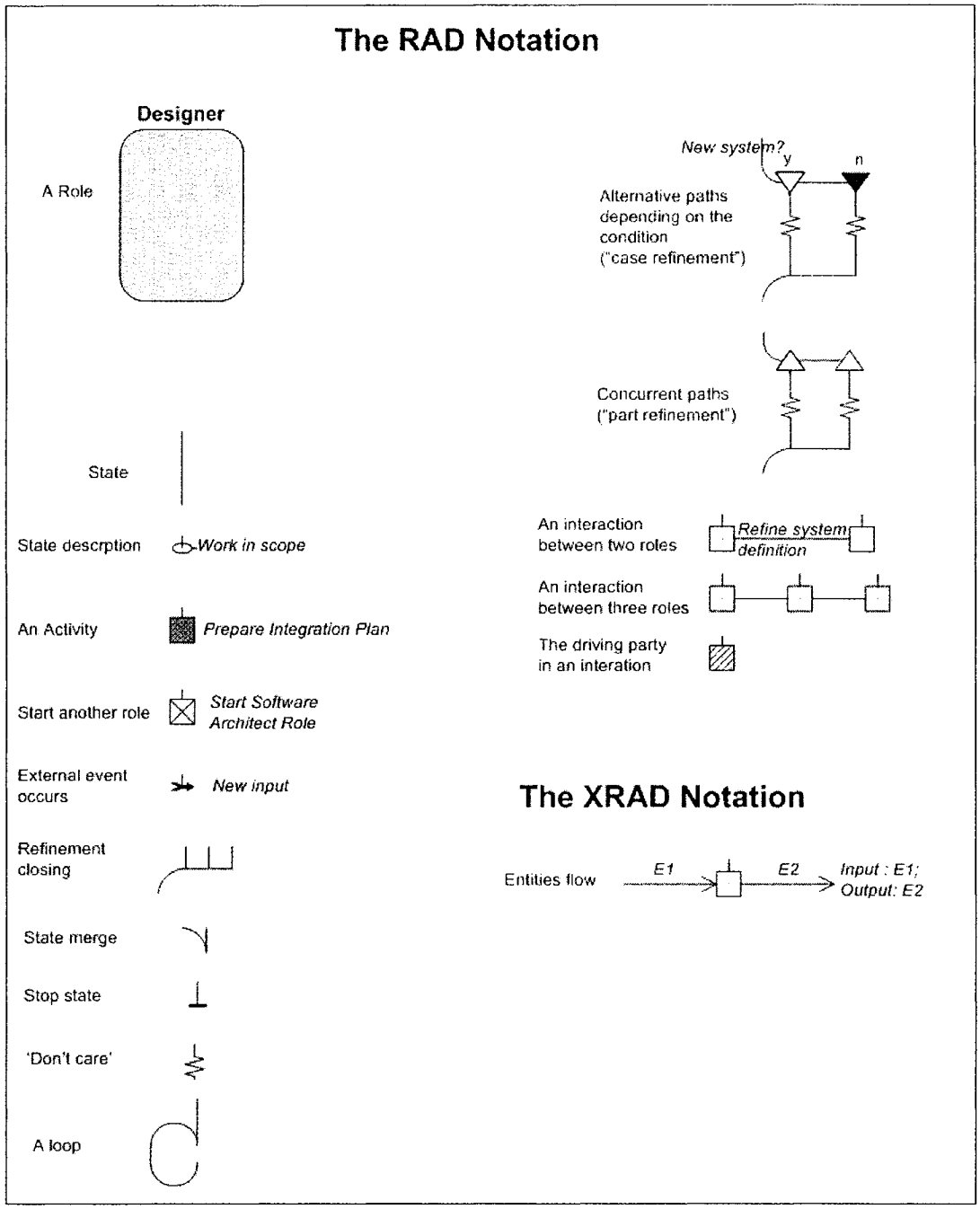


Figure 120: RAD and XRAD *stencil* containing the symbols and notations used

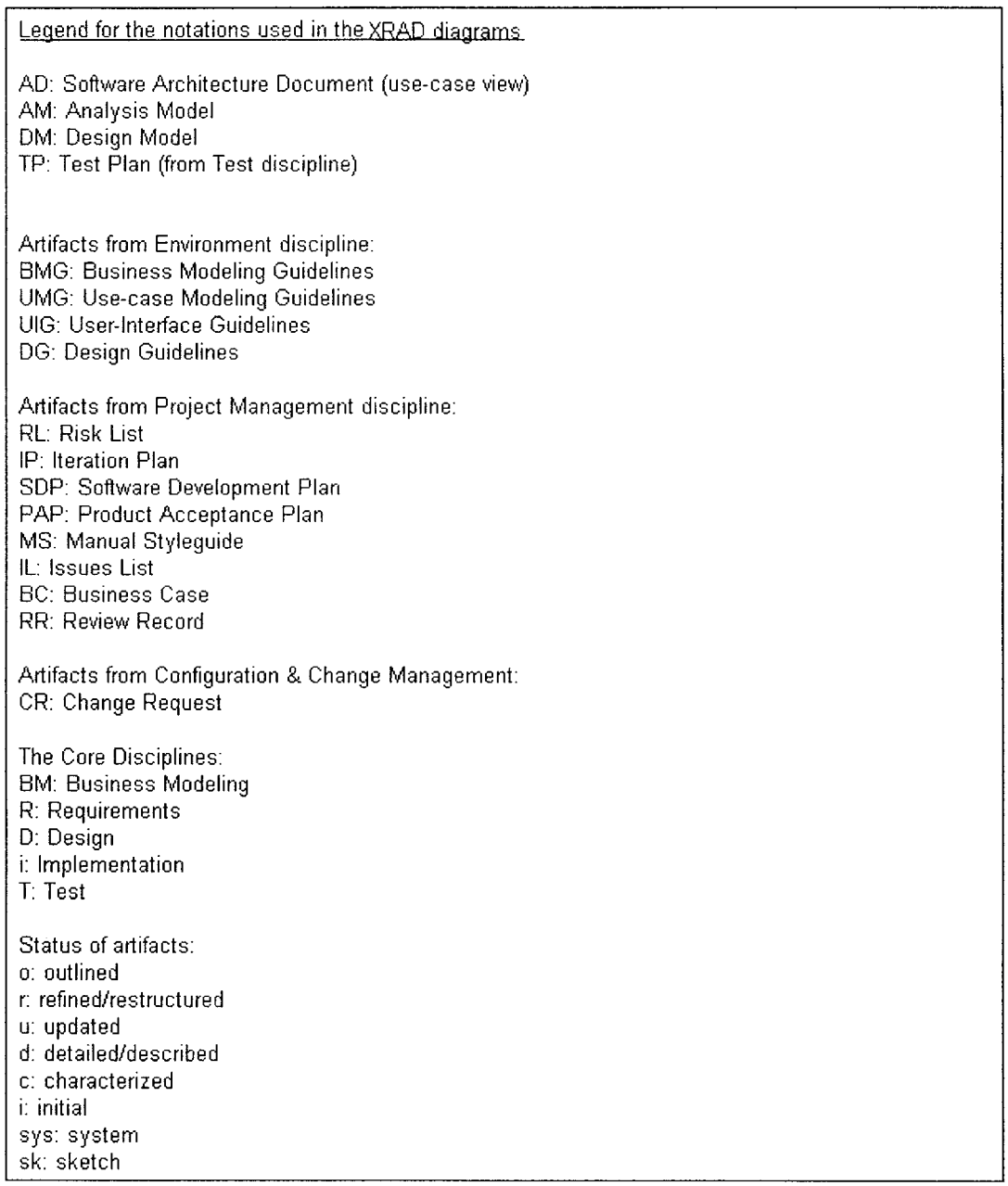


Figure 121: Legend with the abbreviations used for the notations on the artifacts, in the XRAD diagrams

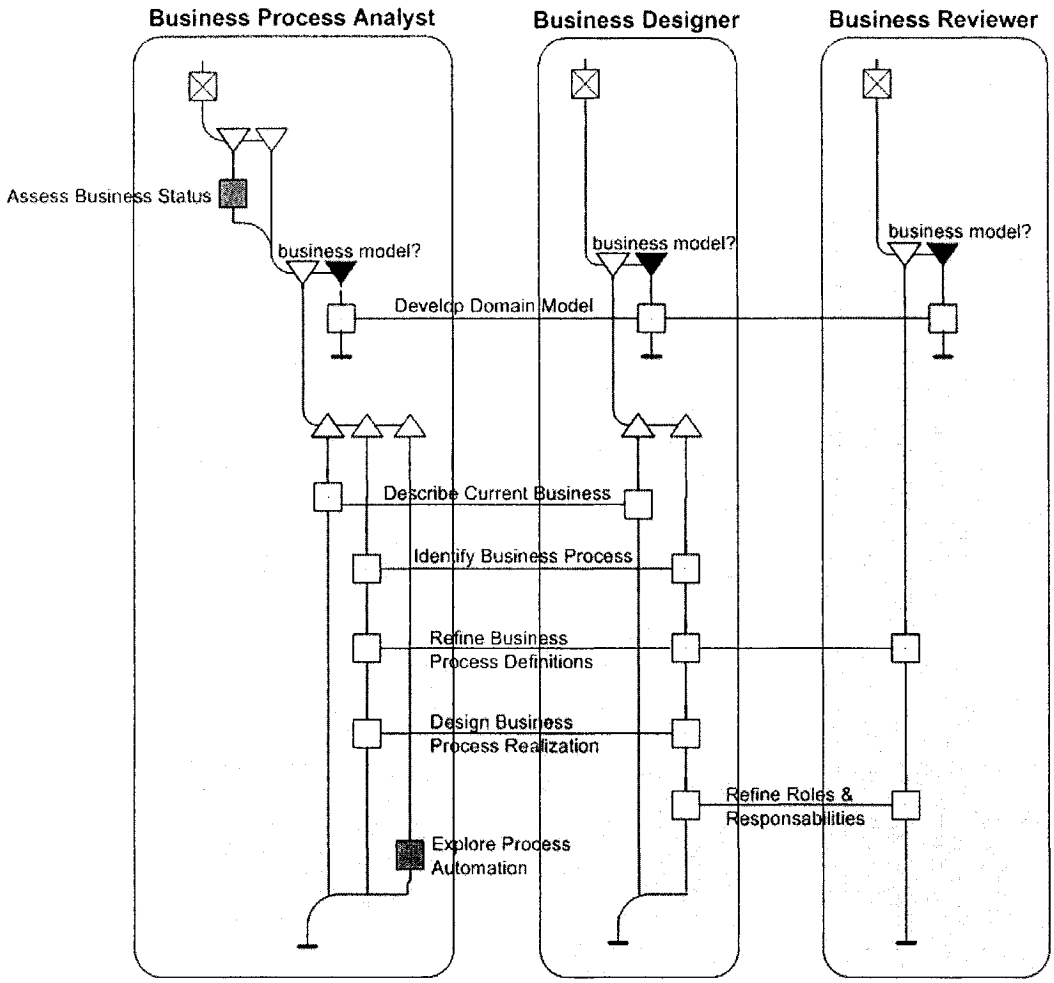


Figure 122: Role Activity Diagram for the *Business Modeling* discipline (v. 1)

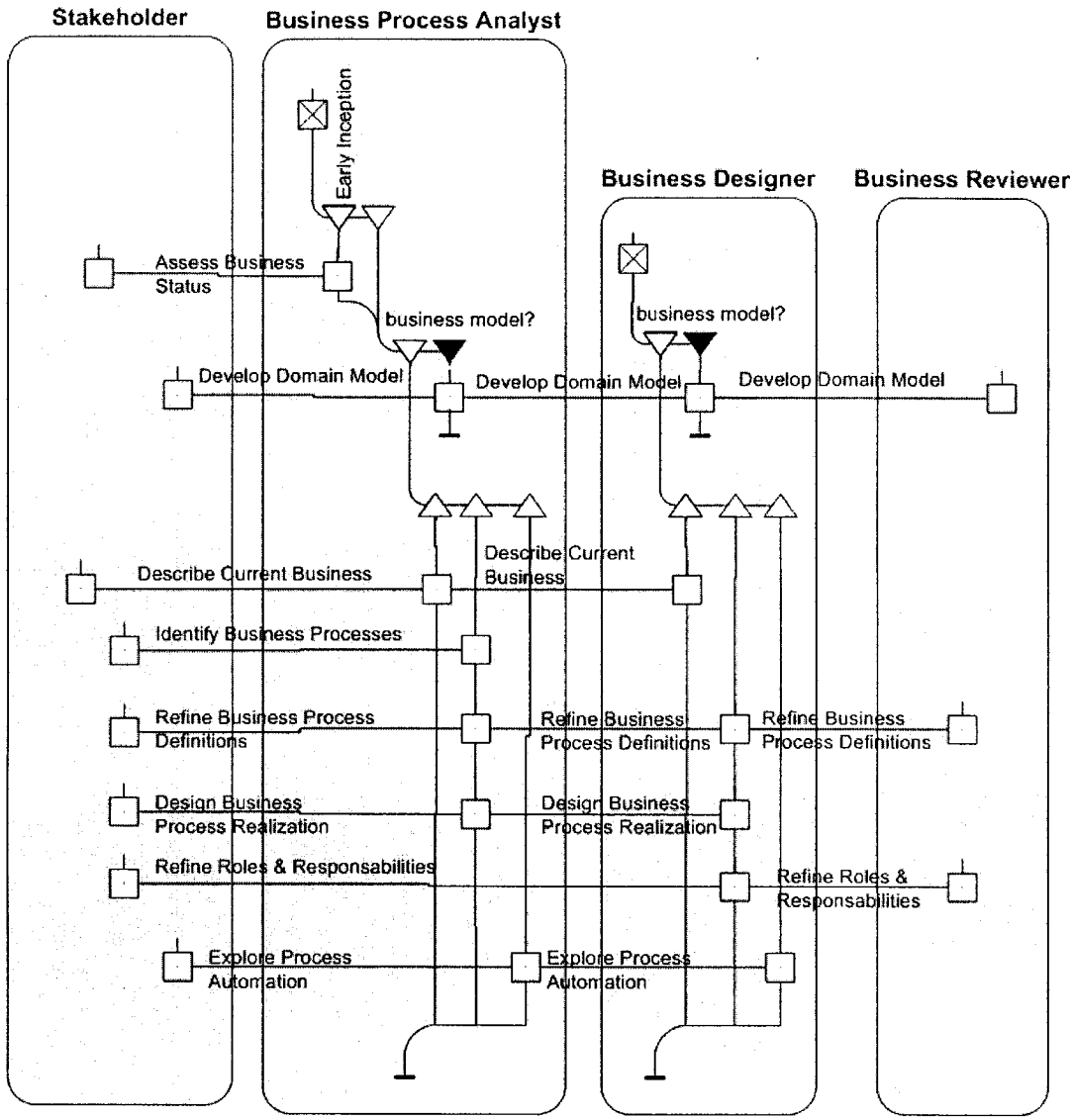


Figure 123: Role Activity Diagram for the *Business Modeling* discipline (v. 2)

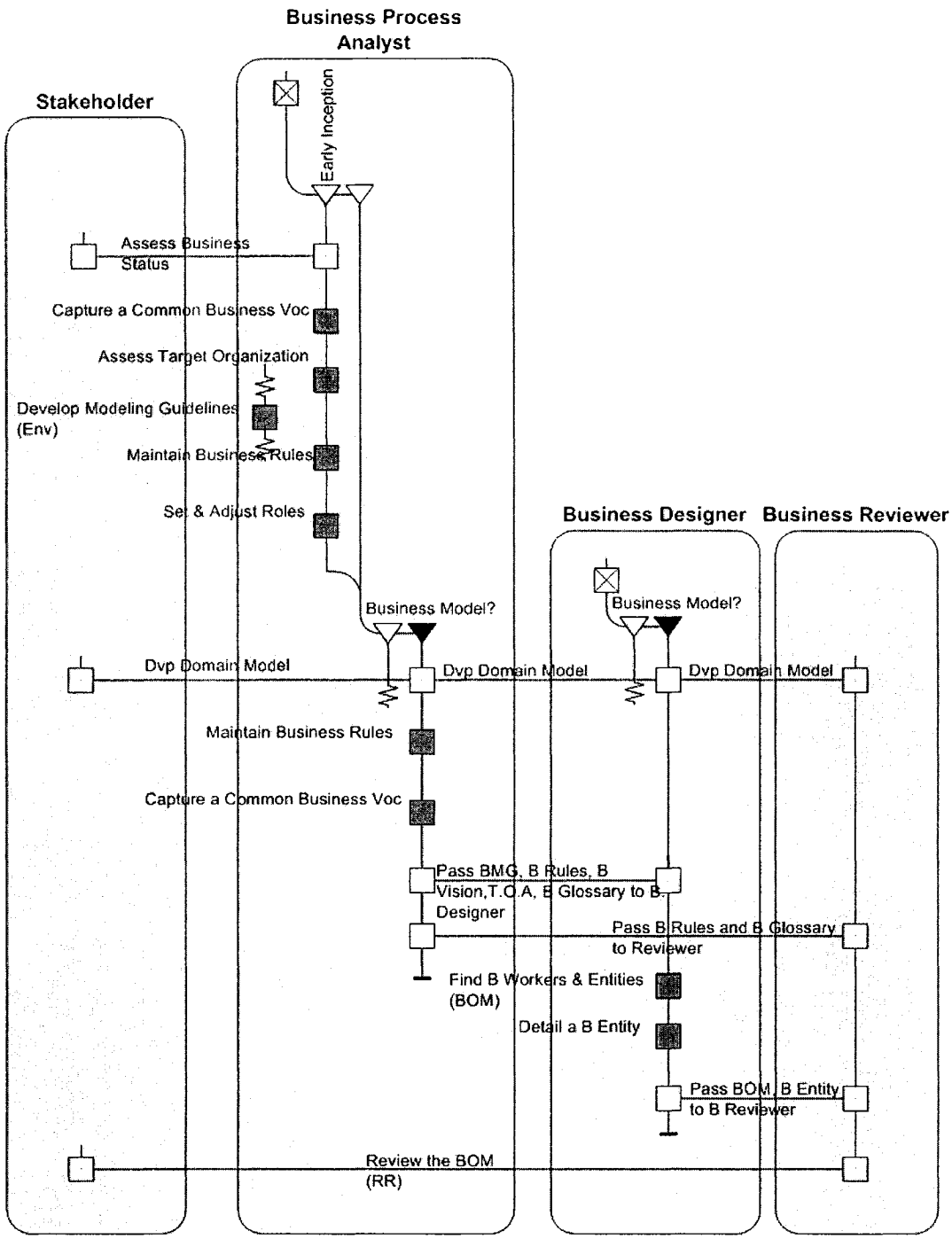


Figure 124: Role Activity Diagram for the *Business Modeling* discipline entities flow

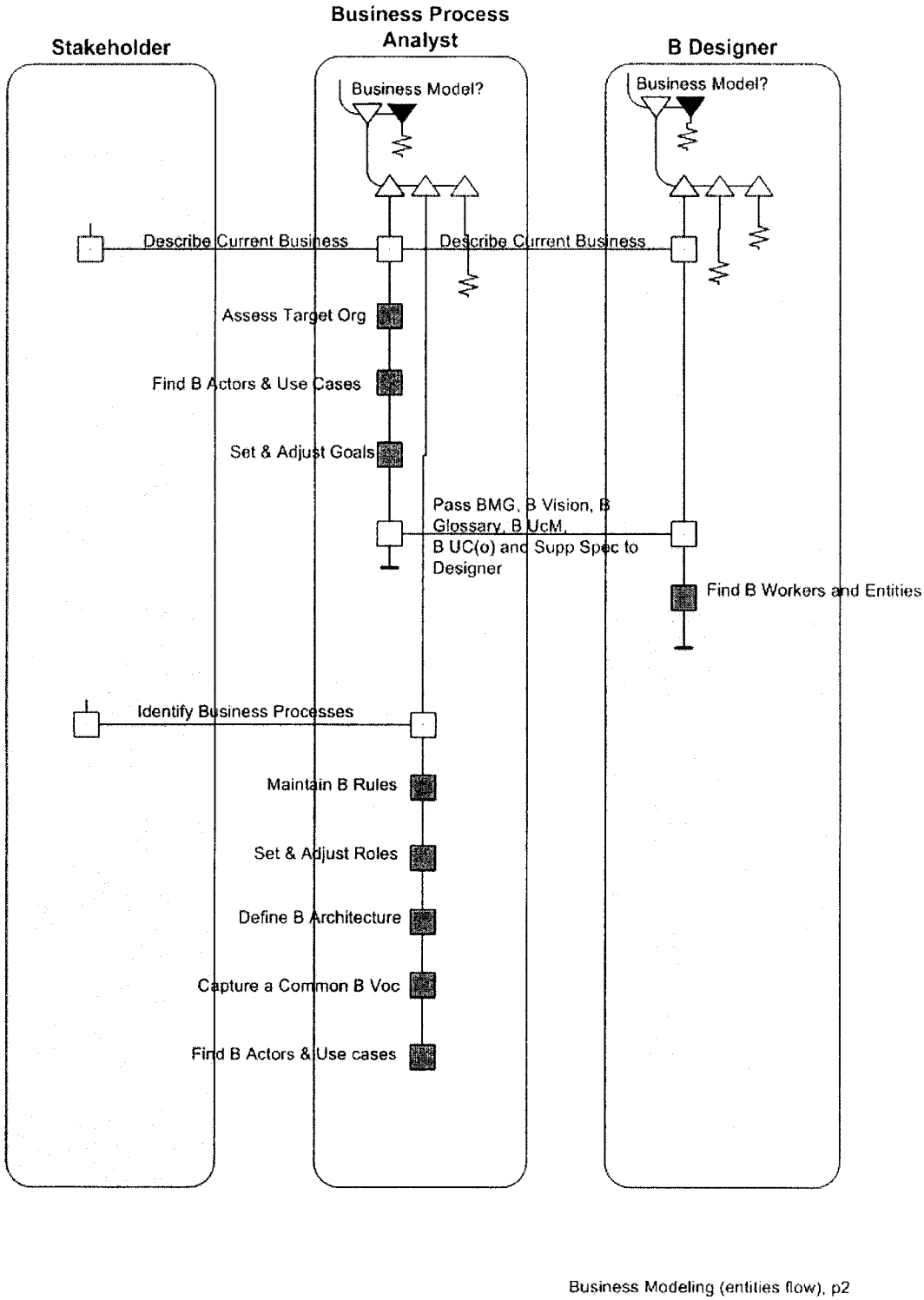


Figure 125: Role Activity Diagram for the *Business Modeling* discipline entities flow (contd.)

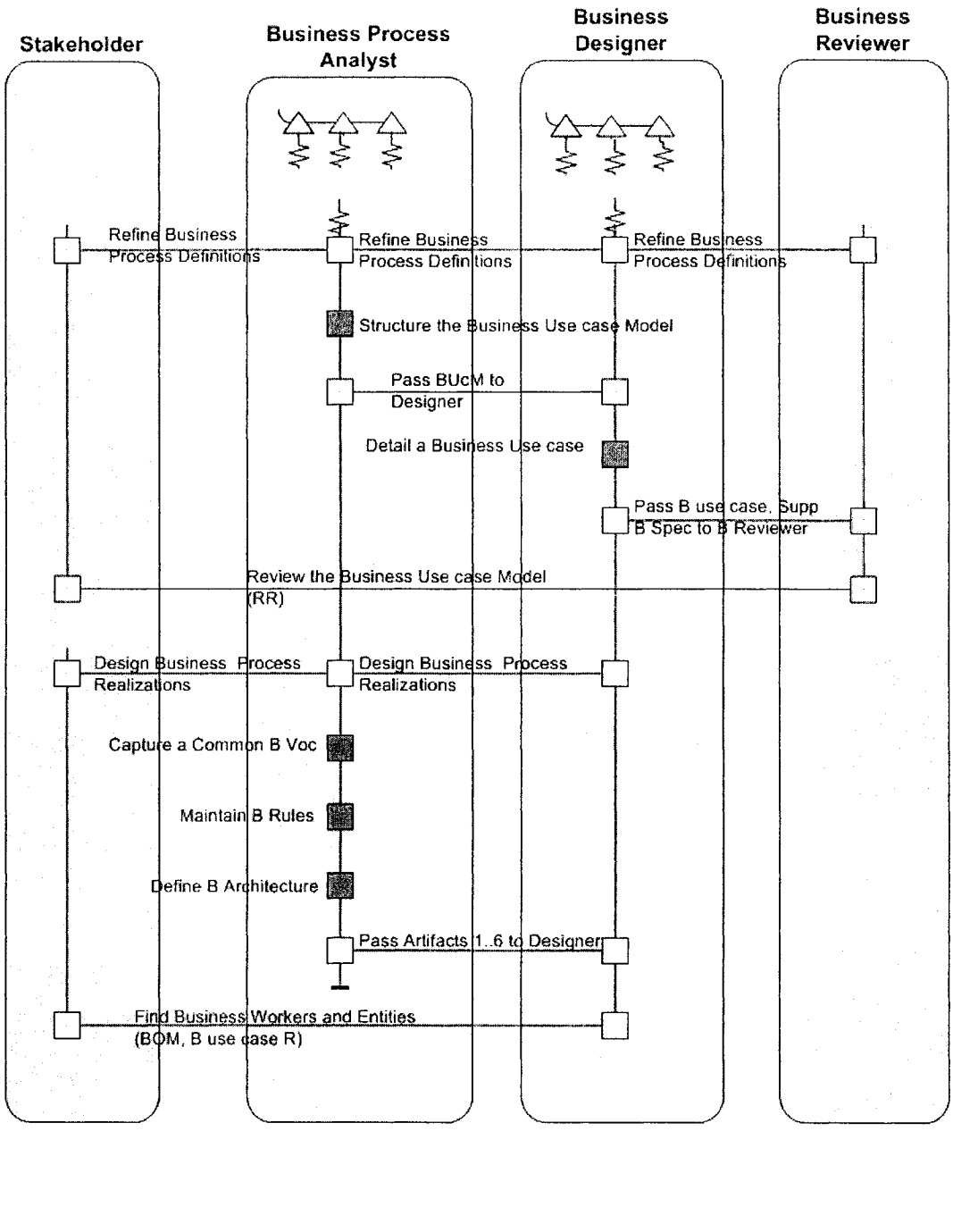
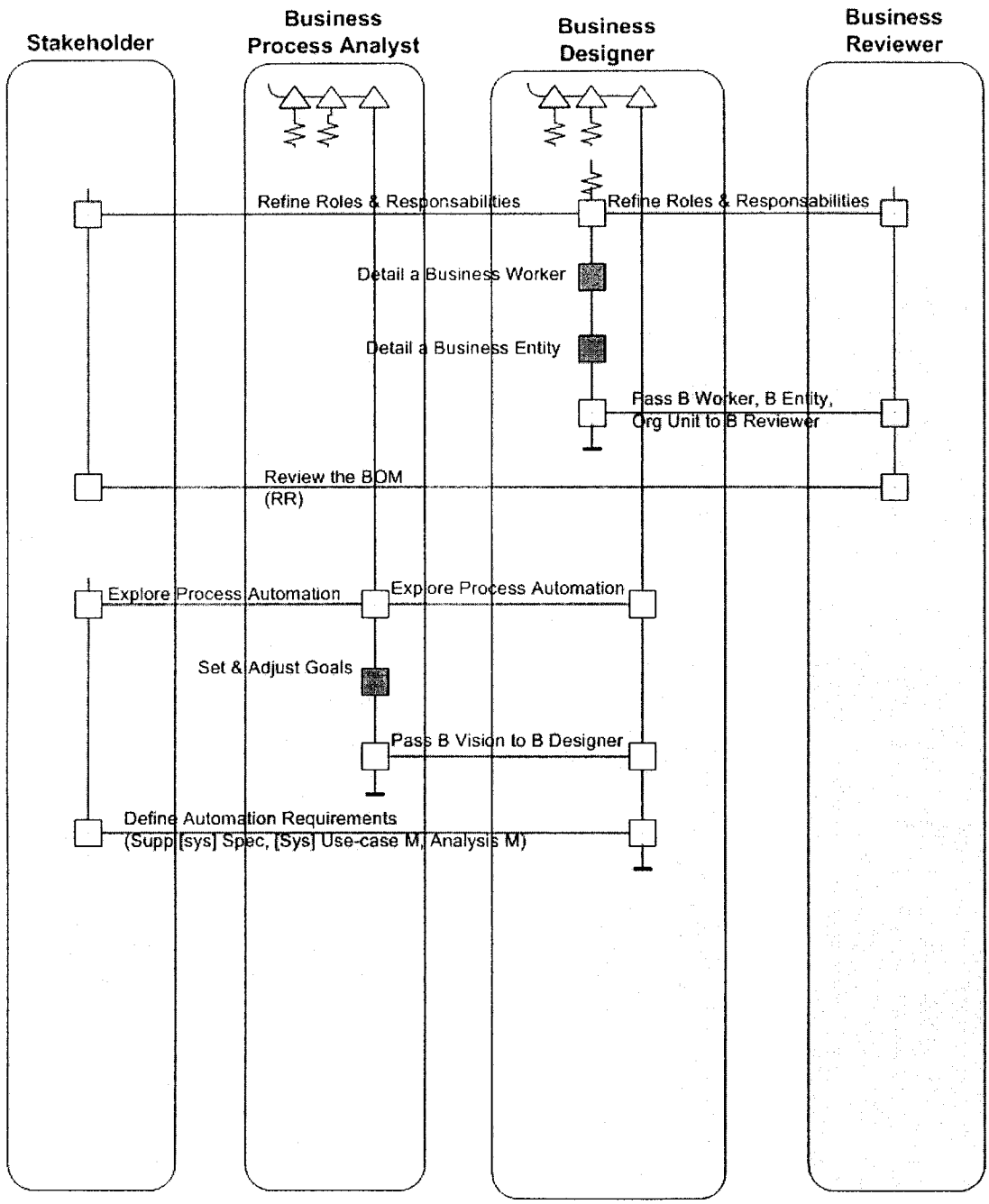


Figure 126: Role Activity Diagram for the *Business Modeling* discipline entities flow (contd.)



Business Modeling (entities flow) p4

Figure 127: Role Activity Diagram for the *Business Modeling* discipline entities flow (contd.)

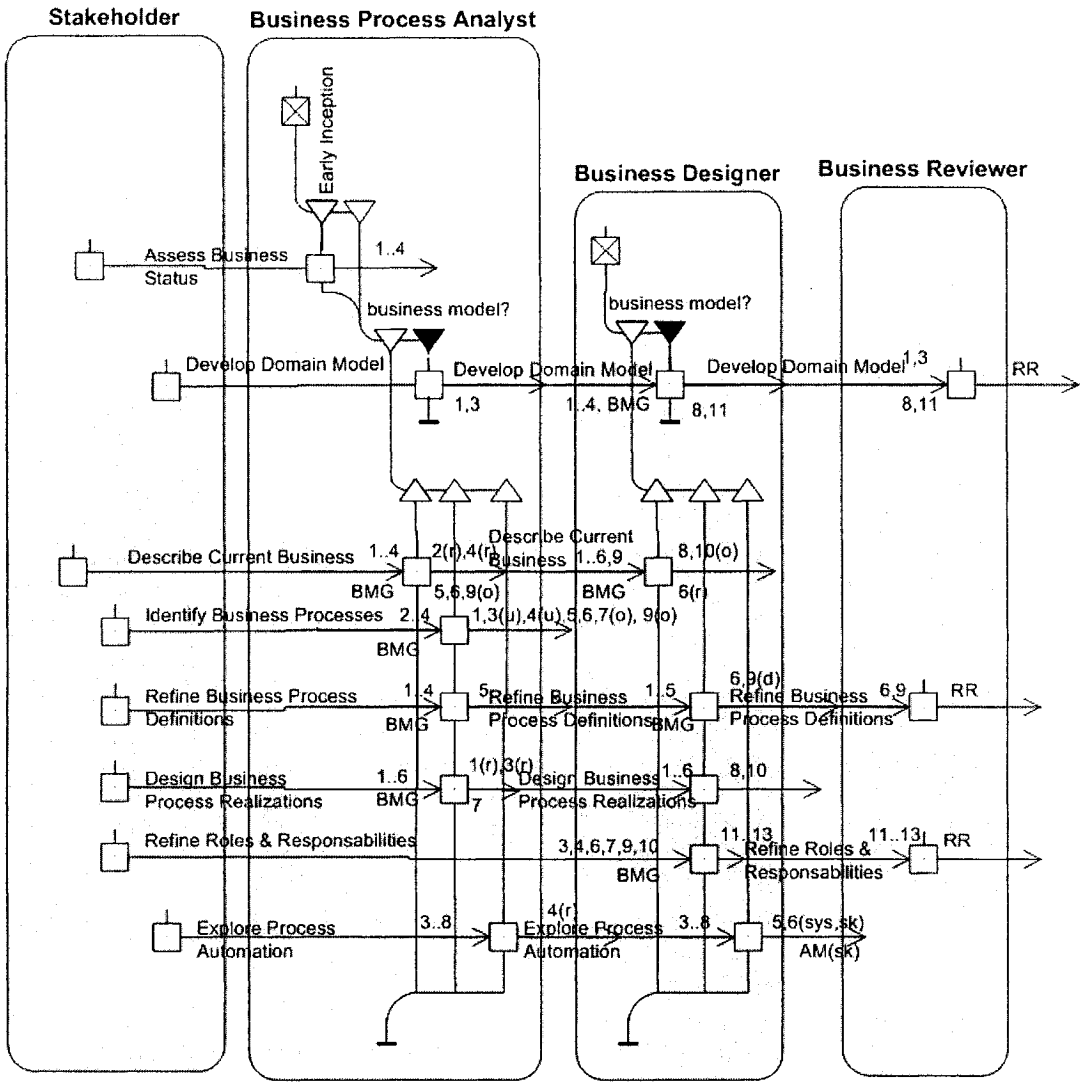


Figure 128: XRAD for *Business Modeling* discipline entities flow

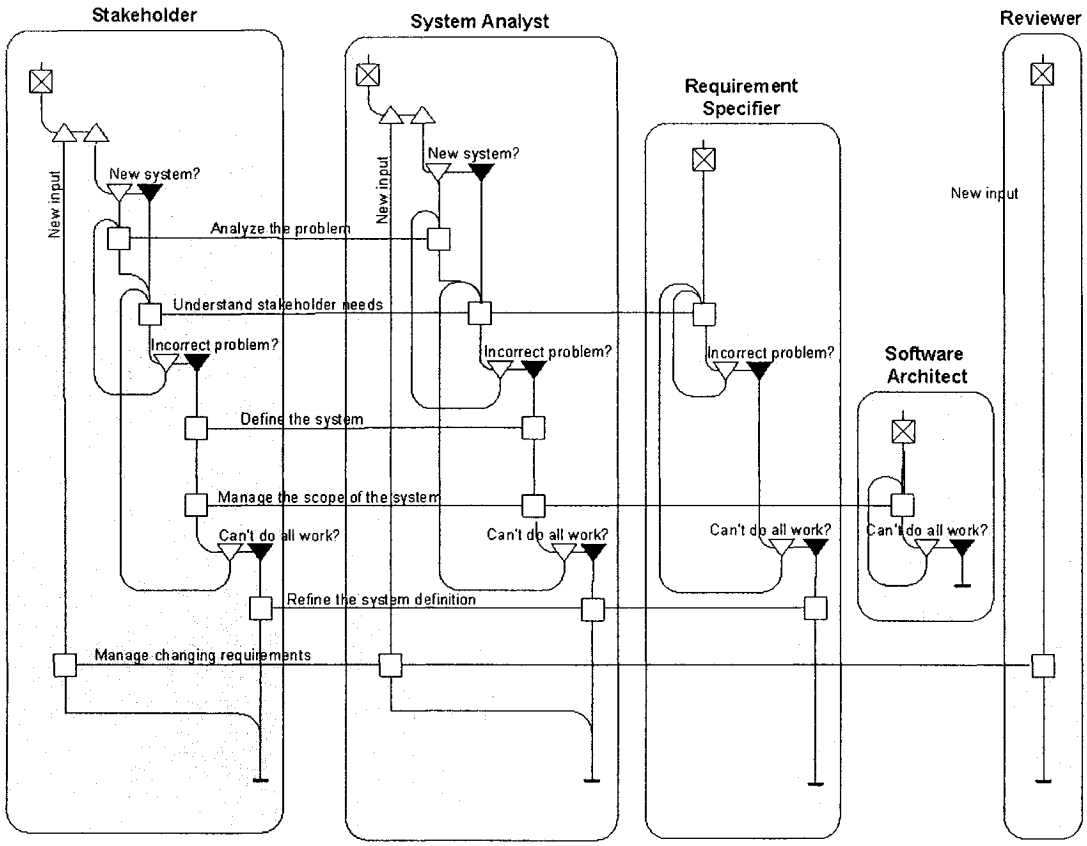


Figure 129: Role Activity Diagram for the *Requirements* discipline (v. 1)

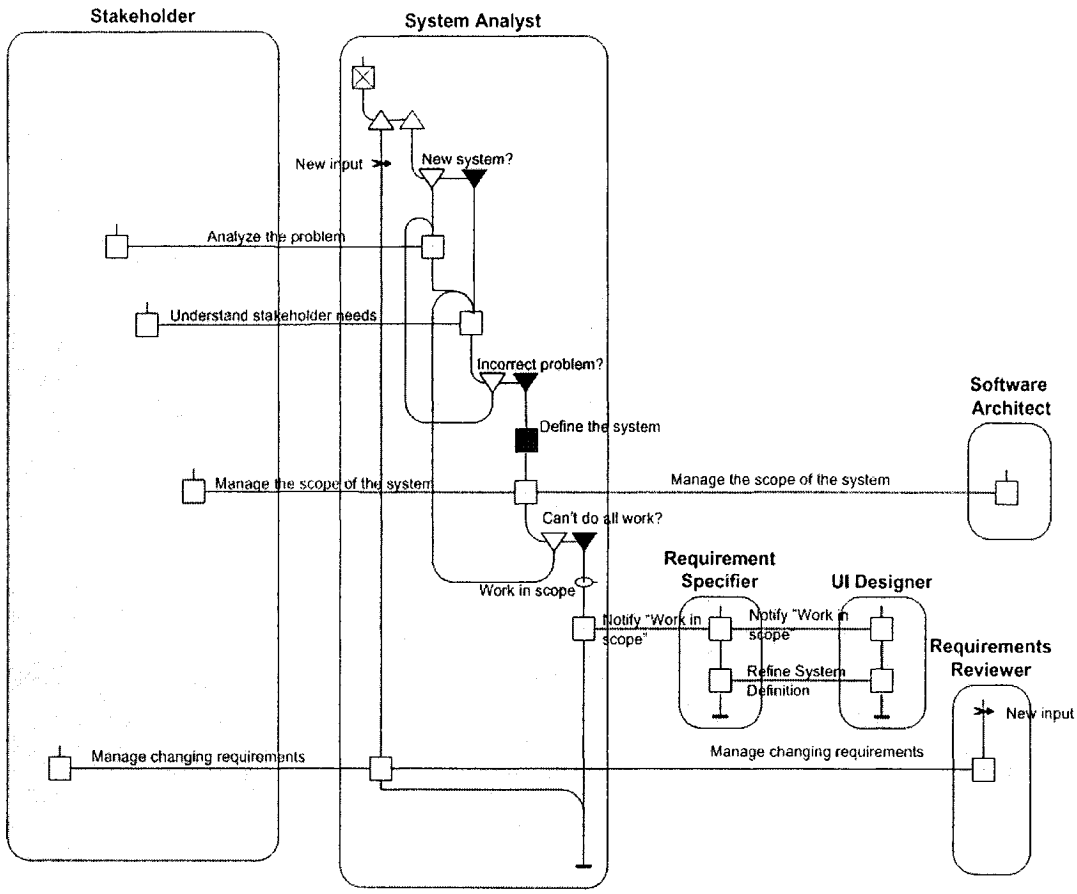


Figure 130: Role Activity Diagram for the *Requirements* discipline (v. 2)

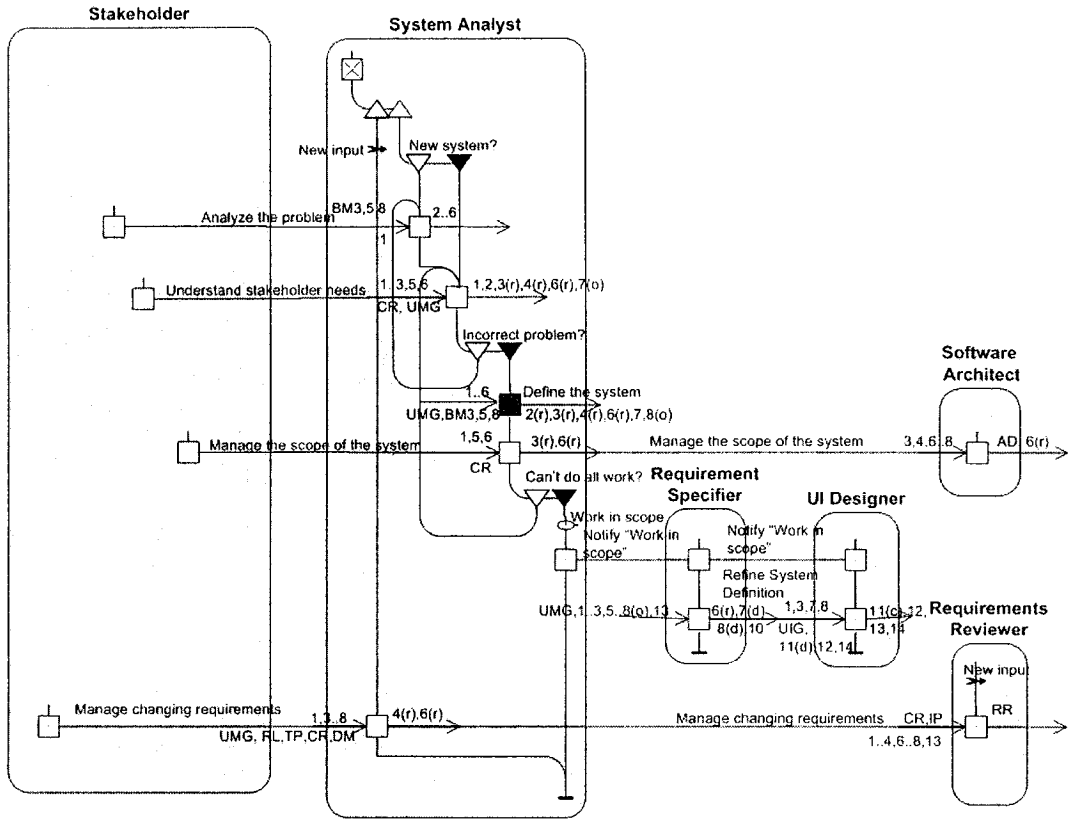


Figure 131: XRAD for the *Requirements* discipline entities flow

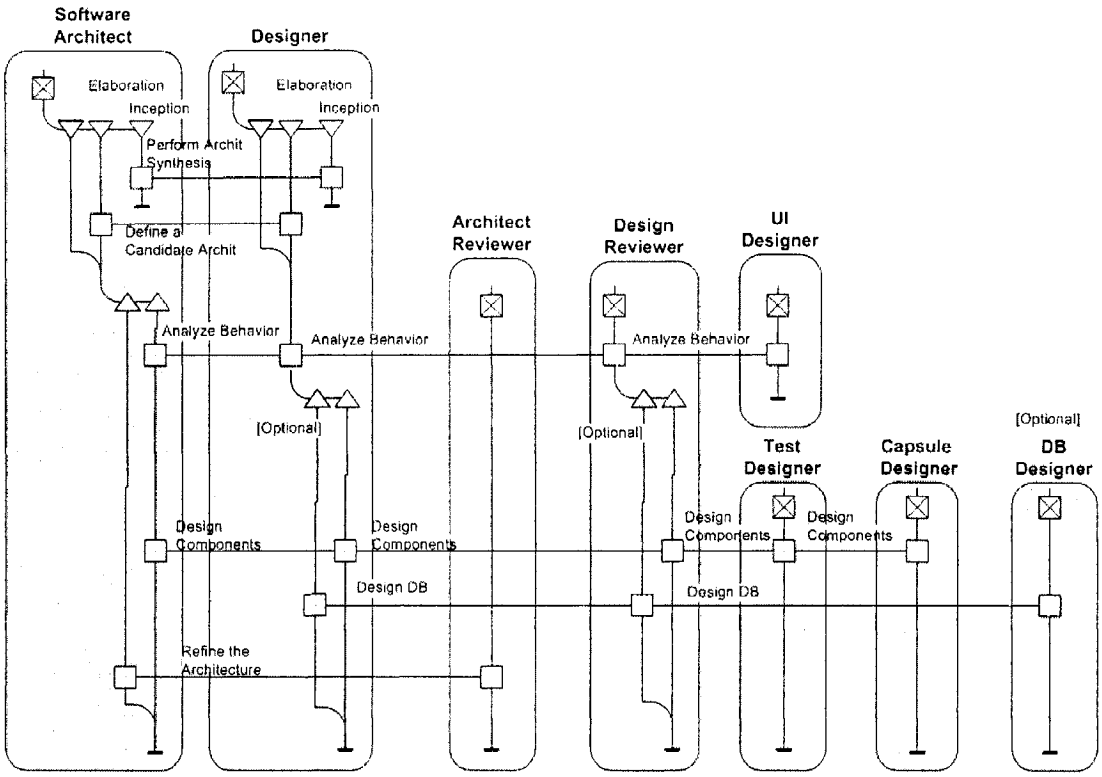


Figure 132: Role Activity Diagram for the *Analysis and Design* discipline (v. 1)

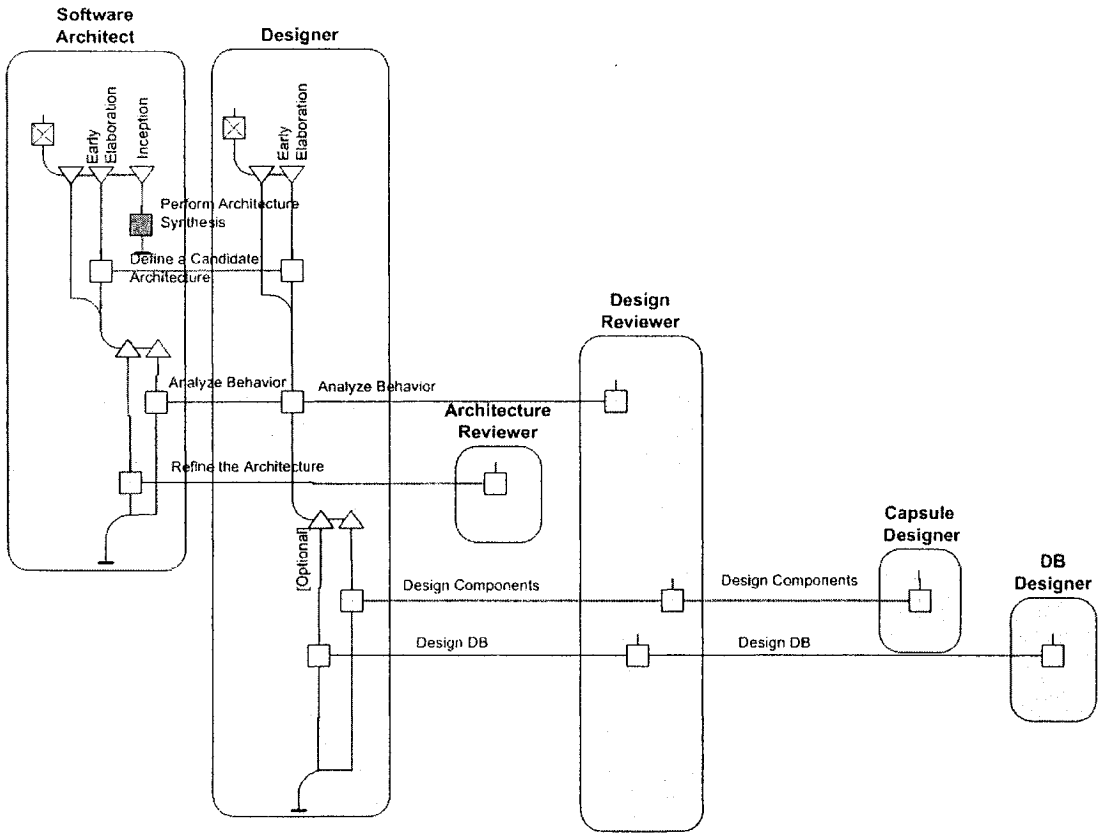


Figure 133: Role Activity Diagram for the *Analysis and Design* discipline (v. 2)

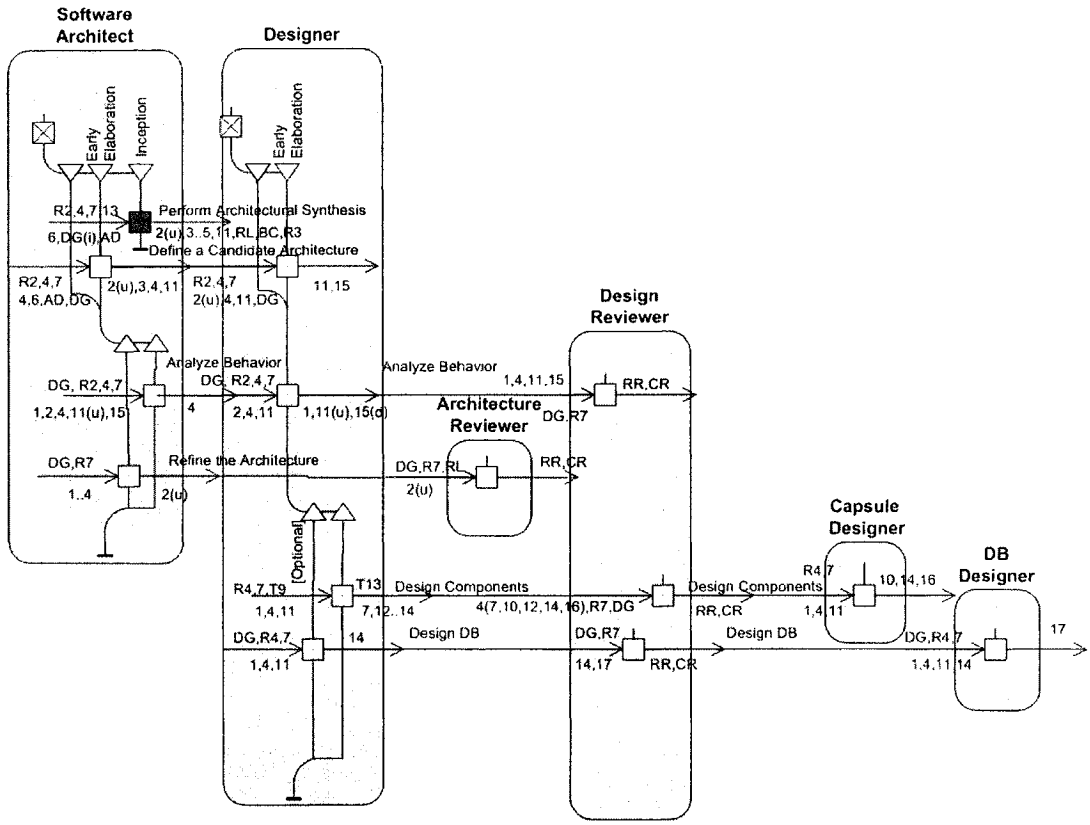


Figure 134: XRAD for the *Analysis and Design* discipline entities flow

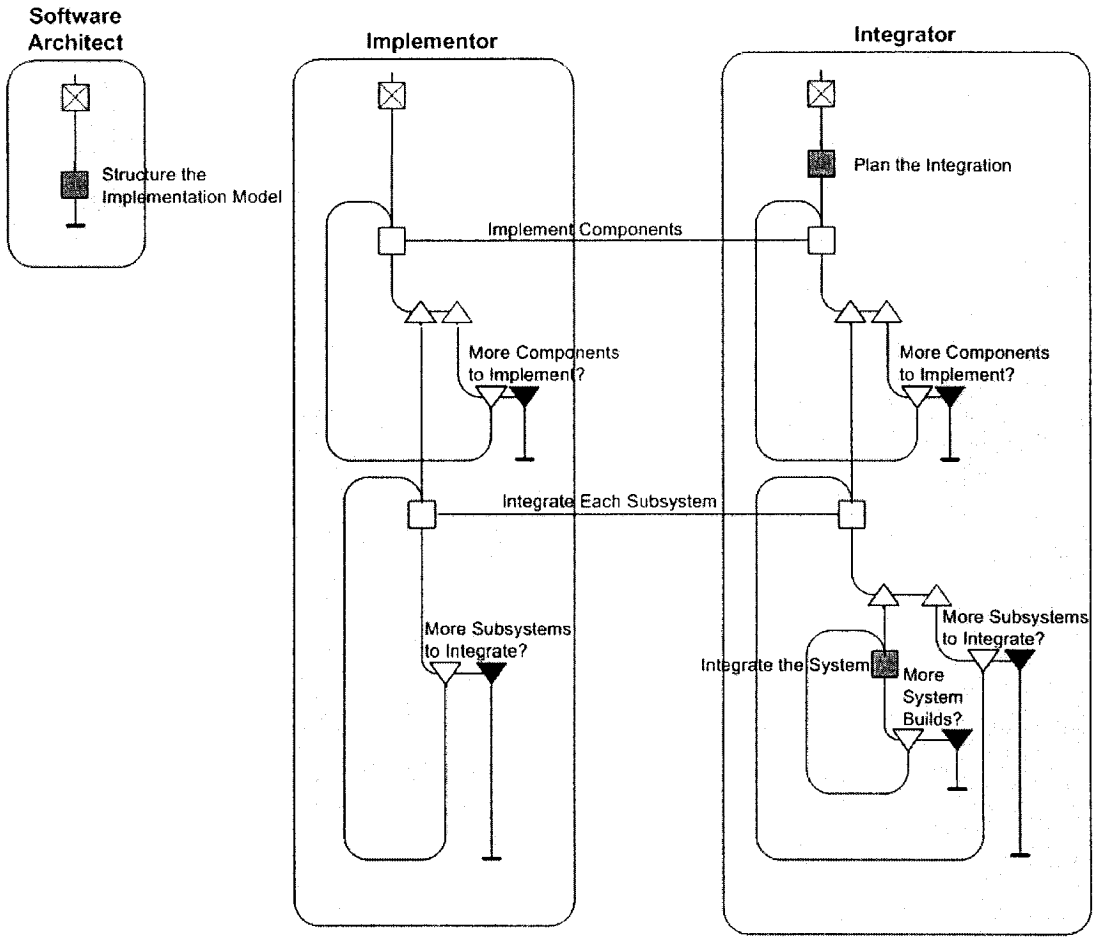


Figure 135: Role Activity Diagram for the *Implementation* discipline (v. 1)

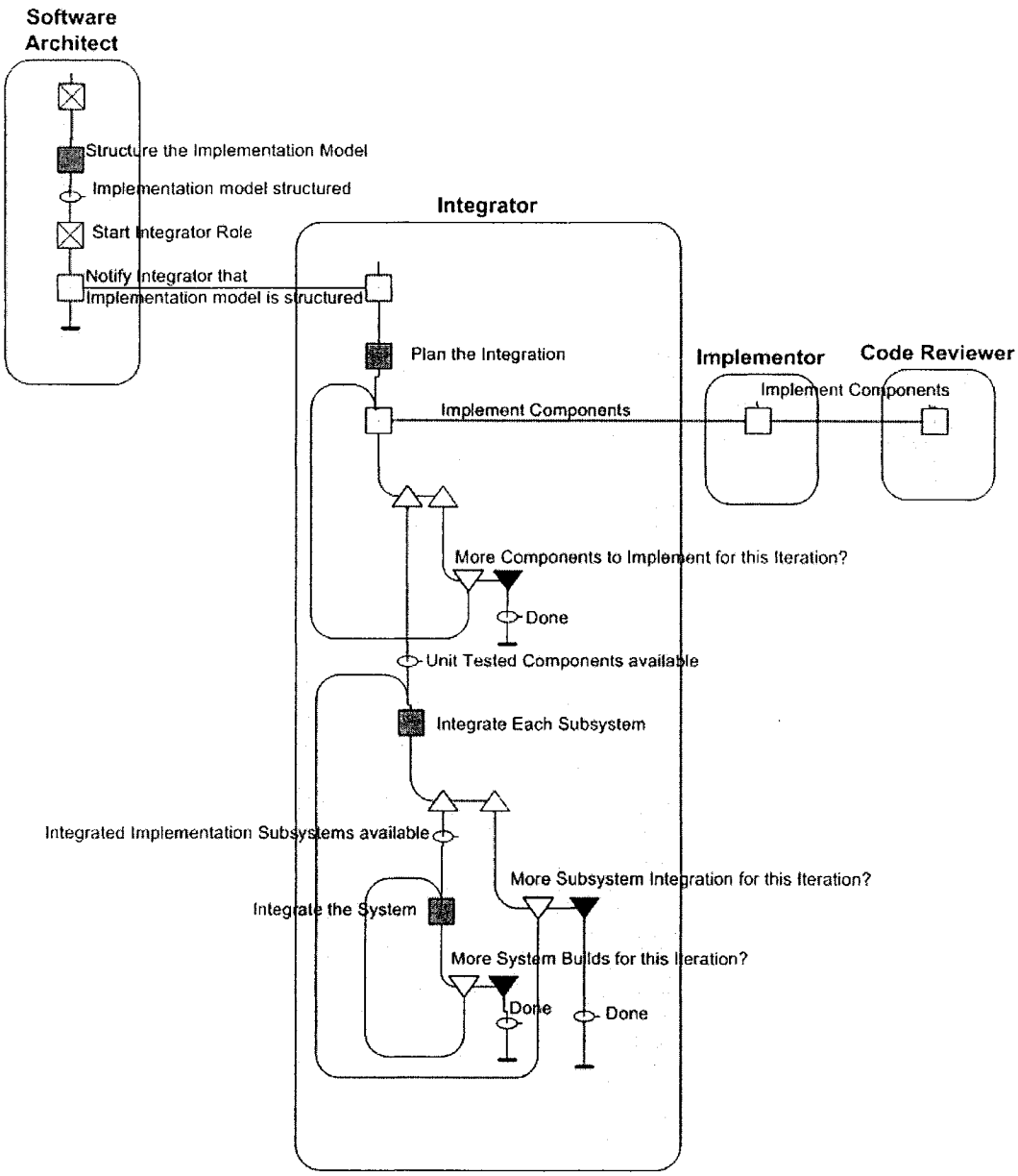


Figure 136: Role Activity Diagram for the *Implementation* discipline (v. 2)

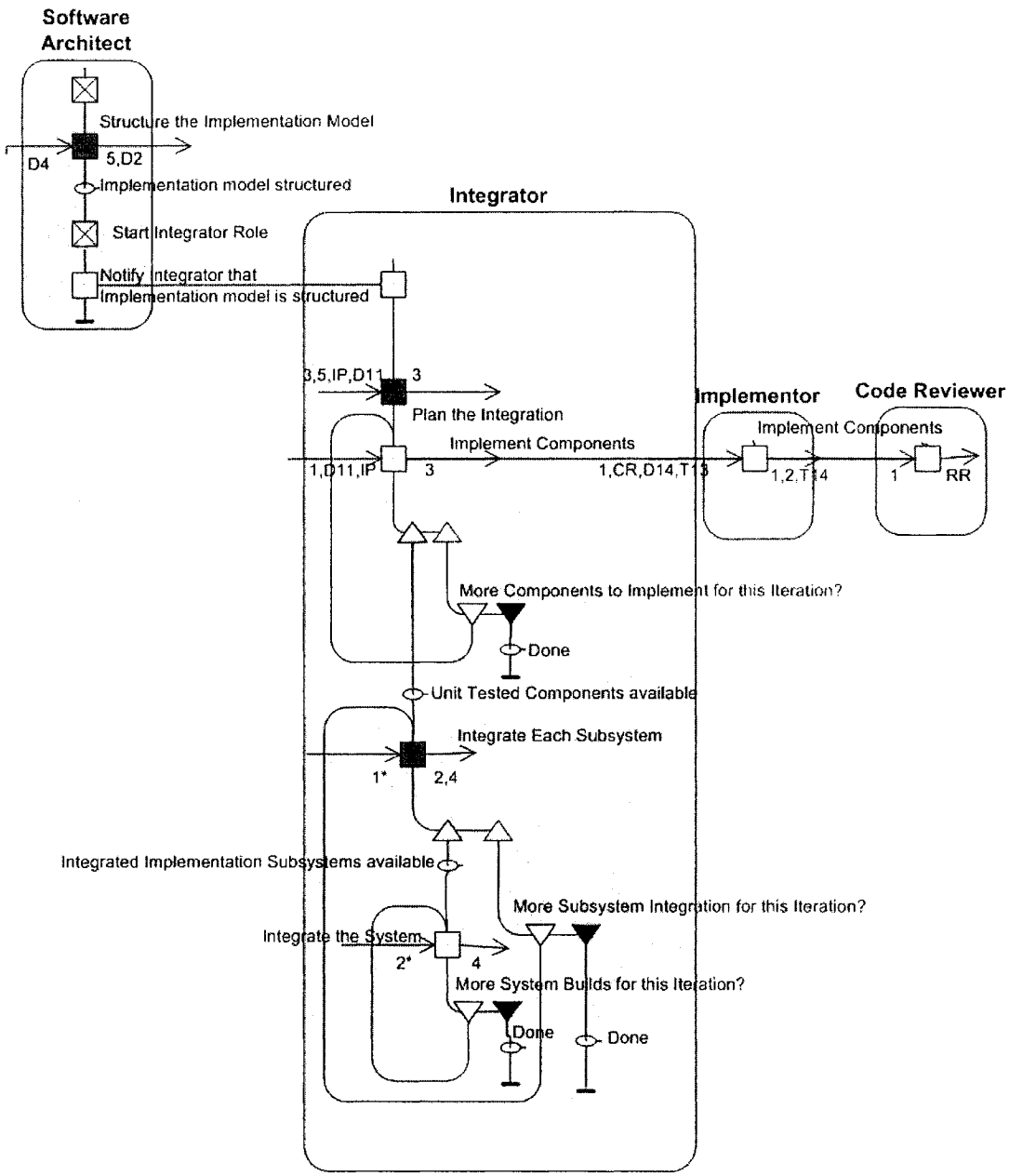


Figure 137: XRAD for the *Implementation discipline* entities flow

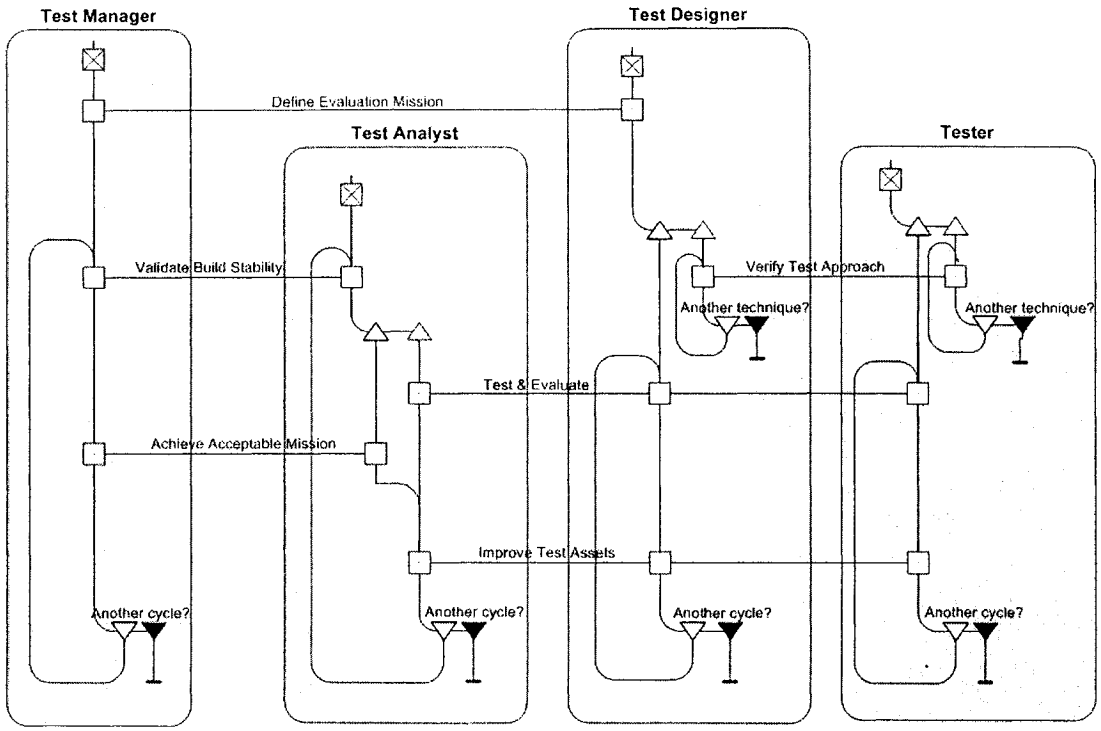


Figure 138: Role Activity Diagram for the *Test* discipline (v. 1)

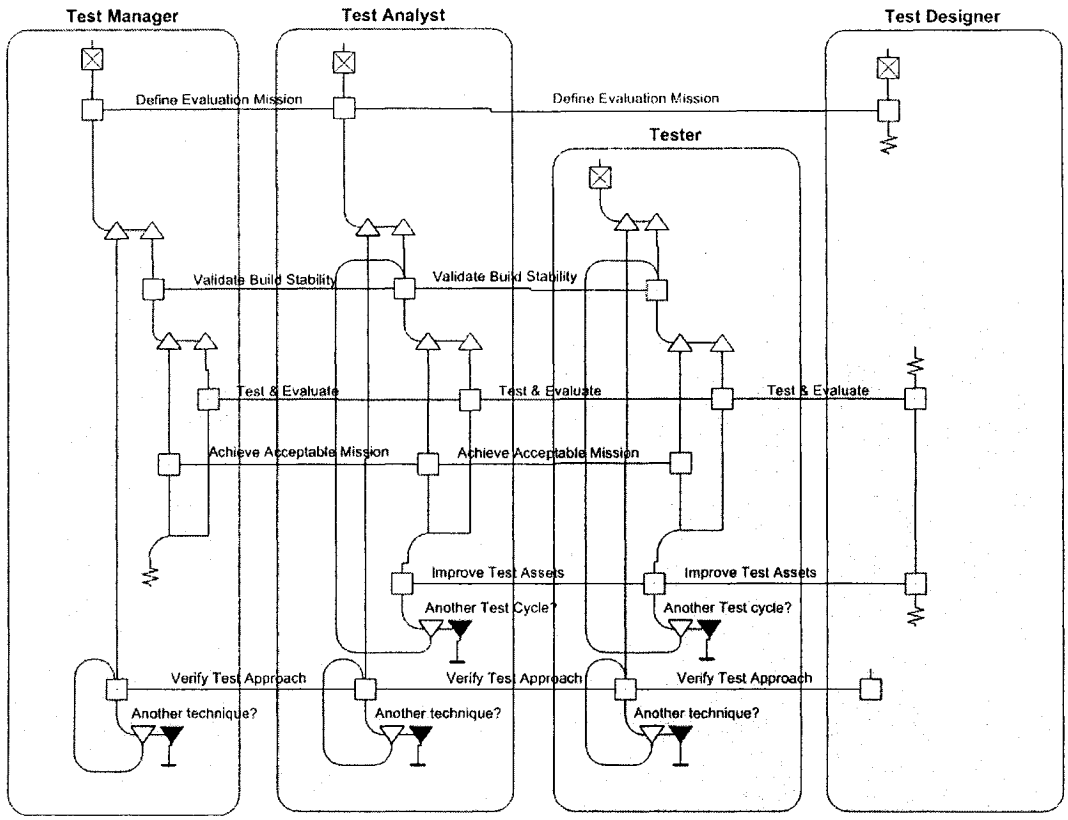


Figure 139: Role Activity Diagram for the *Test* discipline (v. 2)

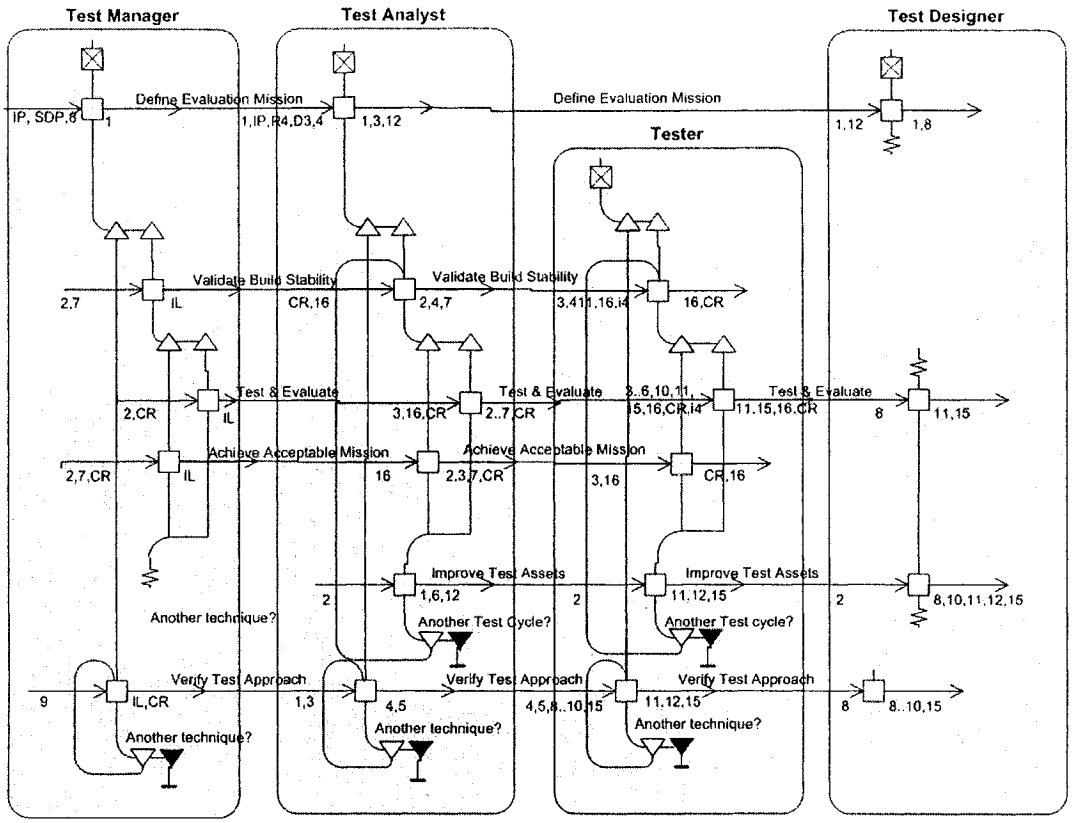


Figure 140: XRAD for the *Test* discipline entities flow

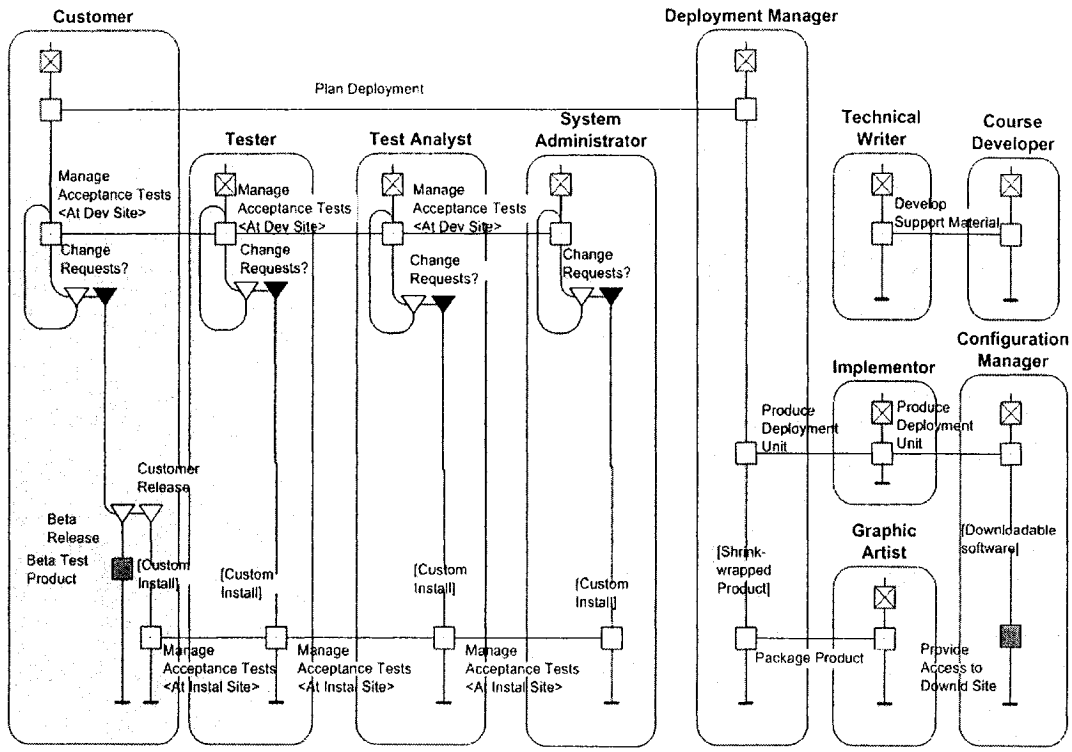


Figure 141: Role Activity Diagram for the *Deployment* discipline (v. 1)

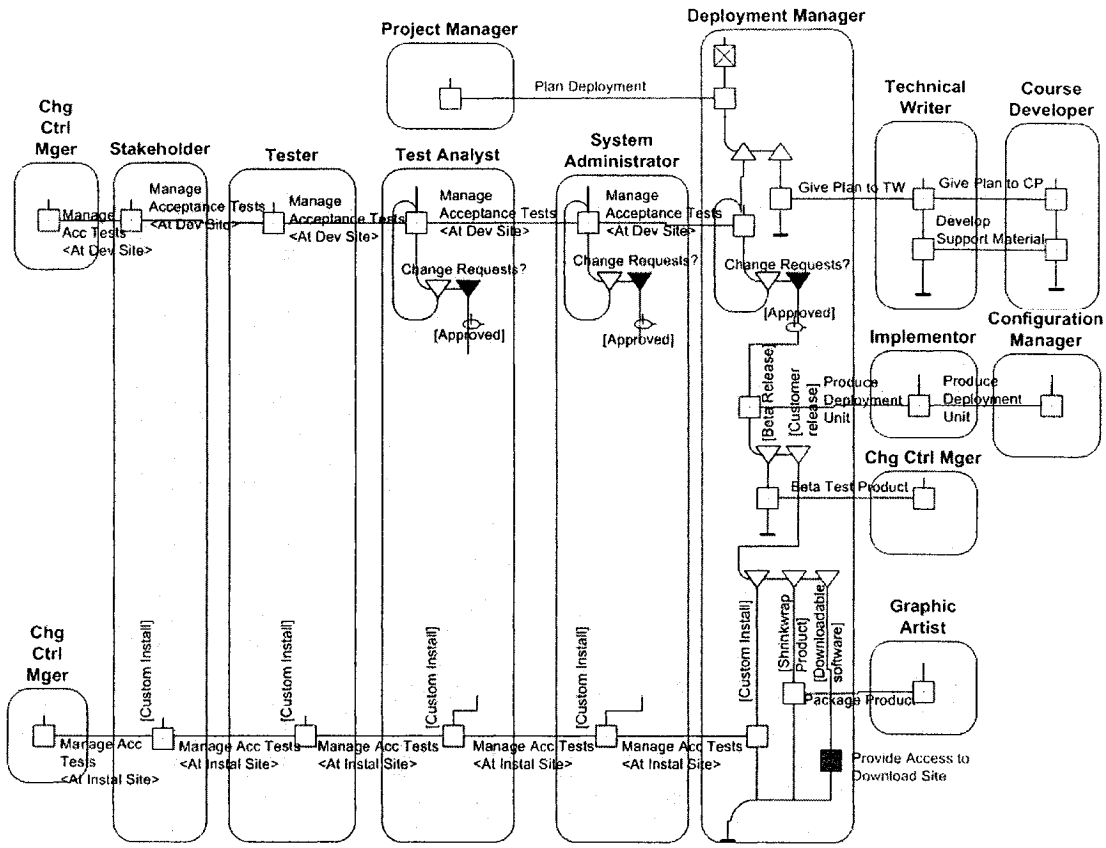


Figure 142: Role Activity Diagram for the *Deployment* discipline (v. 2)

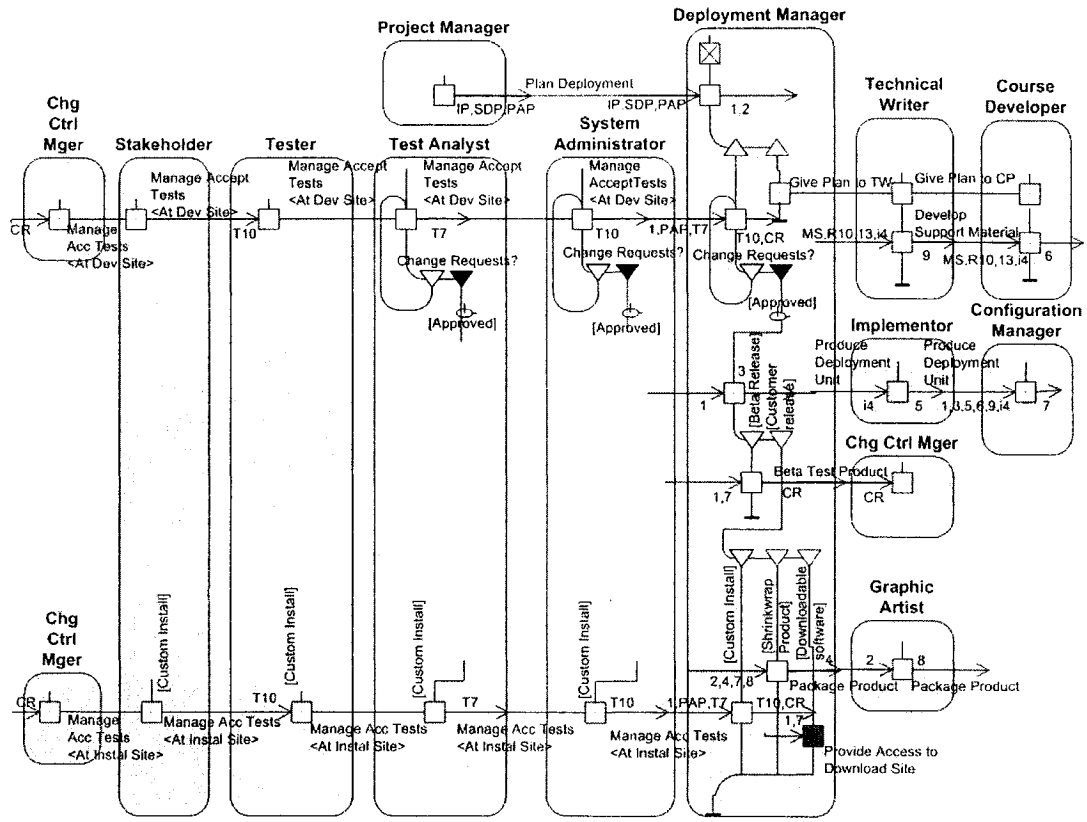


Figure 143: XRAD for the *Deployment* discipline entities flow