



Information Technology for European Advancement

Software Development Process for Real-Time Embedded Software Systems (DESS)

Deliverable D5.1

DESS process models and process development guidelines

Version 02 - Public

Edited by Luigi Lavazza & Vieri del Bianco

Release note

Version 0.1

The DESS process model is not thoroughly specified. This will be done in next release.

Version 0.2

Additions (in chapter 5): High-level definition of the process added. Consideration on component-based development process added.

Version 1.0

Synchronization with Deliverable D1.

Table Of Contents

1.	INTRODUCTION	5
1.1	WORK PACKAGE 5: SOFTWARE DEVELOPMENT PROCESS.....	5
1.2	OBJECTIVES OF THIS DOCUMENT.....	5
2.	REQUIREMENTS FOR THE DESS PROCESS MODEL	7
2.1	DEVELOPERS' REQUIREMENTS.....	7
2.2	SYNTHESIS OF REQUIREMENTS.....	7
3.	PROCESS MODELING NOTATIONS.....	8
3.1	IDEF0.....	8
3.2	UML FOR PROCESS MODELLING.....	9
3.2.1	<i>On the formality of UML as a PML.....</i>	<i>11</i>
4.	TOWARDS THE DESS PROCESS MODEL.....	13
4.1	ANALYSIS OF THE EXISTING MODELS	13
4.2	SPECIFICITY OF DESS	16
4.2.1	<i>User and system requirements</i>	<i>17</i>
4.2.2	<i>Component-based approach</i>	<i>17</i>
4.2.3	<i>Formal methods</i>	<i>18</i>
4.3	DEALING WITH COMPONENTS.....	18
4.3.1	<i>Concepts and terminology.....</i>	<i>18</i>
4.3.2	<i>Process</i>	<i>20</i>
4.3.3	<i>The role of formal notations in component-based development</i>	<i>23</i>
5.	DESS METHODOLOGY	25
5.1	REALIZATION WORKFLOW.....	25
5.2	VALIDATION & VERIFICATION WORKFLOW	26
5.3	REQUIREMENTS MANAGEMENT WORKFLOW.....	27
6.	DESS SOFTWARE DEVELOPMENT PROCESS.....	28
6.1	CUSTOMIZATION	28
6.1.1	<i>Order of Activities.....</i>	<i>28</i>
6.1.2	<i>Cyclical phases.....</i>	<i>28</i>
6.1.3	<i>Format of Artifacts produced.....</i>	<i>28</i>
6.2	DESS PROCESS MODELS.....	29
6.2.1	<i>Case 1.....</i>	<i>29</i>
7.	REFERENCES	32
8.	APPENDIX A: THE QUESTIONNAIRE.....	33

8.1	INTRODUCTION	33
8.2	CURRENT SITUATION.....	33
8.2.1	<i>Product</i>	33
8.2.2	<i>Process</i>	34
8.3	REQUIREMENTS FOR THE DESS PROCESS.....	35

1. Introduction

1.1 Work package 5: Software Development process

The development of complex software like real-time and embedded applications requires an equally complex software process.

Work package 5 aims at supporting the development process under several respects:

- Provide guidance to developers, indicating how the various development activities have to be carried out according to the methodology defined in WP1.
- Support coordination and communication among developers.
- Complement and integrate the functionality delivered by tools (especially those developed in WP2).

Work package 5 will address the aforementioned goals by means of the following actions:

- Define suitable models describing the process development activities, specifying the working details, to what phase they relate (i.e., when they have to be carried out), what are the conditions for initiating and finishing the activities, what resources (both human and tools) they need, etc. The models will comply with the DESS methodology. On one hand they will be specifically tailored according to the applications' features, on the other hand it will be possible to customize them for the different developers' environments.
- Specify the functionality of a process infrastructure supporting the cooperation among the tools to be developed in WP2 and process-specific, human-oriented tools (such as developers' agendas, planning tools, e-mail, etc.). The DESS project will build on existing infrastructures as required by the methodology, the tools and the process models. The process infrastructure will support data exchange (according to a common format, possibly based on XML), and coordination information exchange (e.g., relevant event notifications).

The result of the work package will be a process support environment (equipped with a library of reference models) flexible enough to accommodate application- and user-dependent features, while specifically suited for real-time and embedded applications.

The activities in work package 5 will be the responsibility of Barco, Bull, Siemens, THOMSON multimedia, CEFRIEL, Uni Paderborn, Philips.

1.2 Objectives of this document

The goal of this document is to define a (general) process models consistent with the DESS methodology, and to derive useful development process guidelines for organizations adopting the DESS methodology.

In order to achieve this goal, several activities have been carried out:

- An exploration phase has explored the development processes in use at the industrial partners' sites. This exploration was carried out mainly by means of a questionnaire, which is reported in Section 8.
- The information collected by means of the questionnaire were analyzed, in order to better understand the requirements of developers. The result of such analysis is reported in Section 2.
- A suitable notation for the representation of process models was selected. In particular, two candidates, IDEF0 and UML were considered. The discussion of process modelling notations is reported in Section 3.

- Taking into account the results of the activities above, we defined a generic process models consistent with the DESS methodology and development process guidelines for organizations adopting the DESS methodology. These are reported in Section 4.

2. Requirements for the DESS process model

2.1 Developers' requirements

In general, rather mature processes are in place.

UML is widely adopted; several developers employ tools by Rational, especially Rose.

Needs:

- ❑ Simple, useful, adaptable process (Thomson multimedia)
- ❑ Take into account continuous activities, such as Risk Management, Configuration Management, Requirements Management or Project Management (Thomson multimedia, Philips)
- ❑ Scalability of the process definition (Philips)
- ❑ As to the technical activities to be supported:
 - ❑ Component-based realtime development (Daimler)
 - ❑ Test involving manufacturers and suppliers (Daimler)
 - ❑ Tools must be particularly well adapted to the Test and Integration activity (Thomson multimedia and Philips)
 - ❑ Requirements tracing and change (Philips)

Standard communication and coordination means are in use (email, web, ...).

In conclusion, there is the clear need to define a generic process that can be adapted to the different process instances.

Support by tools is a general (sometimes implicit) need. Of course we shall try to include in process support at least the tools developed within DESS¹.

2.2 Synthesis of requirements

In practice the fundamental requirements for the DESS process model are:

- ❑ To be compliant with international standards, the most important being those defined by the Workflow Management Coalition
- ❑ To be easy to translate into OPSS entities, being OPSS the environment chosen to experiment with DESS process support
- ❑ To satisfy the indications provided by partners through the questionnaire.

¹ DESS tools include the existing ones which have been adapted in order to support the DESS methodology.

3. Process modeling notations

Process modeling is needed at different levels:

- At the management level, a relatively high-level, abstract description of the process is needed, in order to provide the management with the descriptive means to define the most effective process. Such a model will be used as a reference to guide the work of developers.
- At the enactment level, a more precise, operational “interpretable” notation is needed. Such notation should be precise enough to allow tools (in particular the process support environment) to understand it and behave accordingly.

3.1 Idef0

Idef0 [1] is a graphical, semi-formal notation, derived from the well-known data-flow diagrams (DFDs).

The main element of an Idef0 diagram, the activity, is described in Figure 1. Inputs are on the left of the activity, output on the right, resources are represented as arrows from below, and control and enabling events are represent as arrows from above.

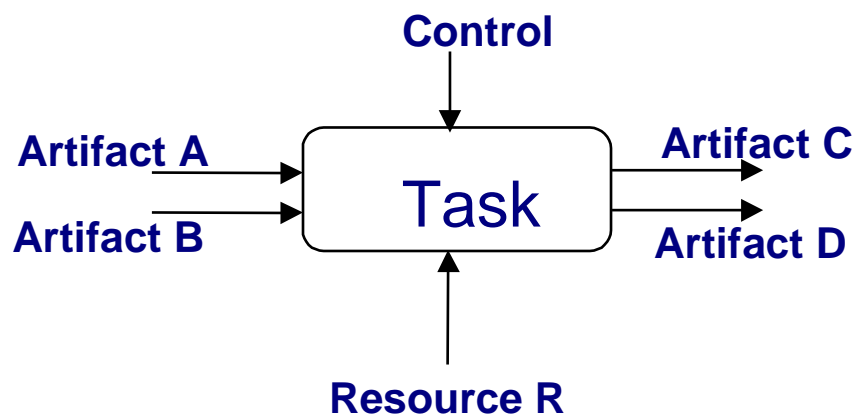


Figure 1. An activity in an Idef0 diagram.

Idef0 is quite popular, since it is easy to use, and easy to understand.

Being semiformal, Idef0 is not usable for enactment. Its best usage is for the aforementioned high-level, management oriented descriptions.

A sample Idef0 diagram, representing a high-level view of a development process, is reported in Figure 2.

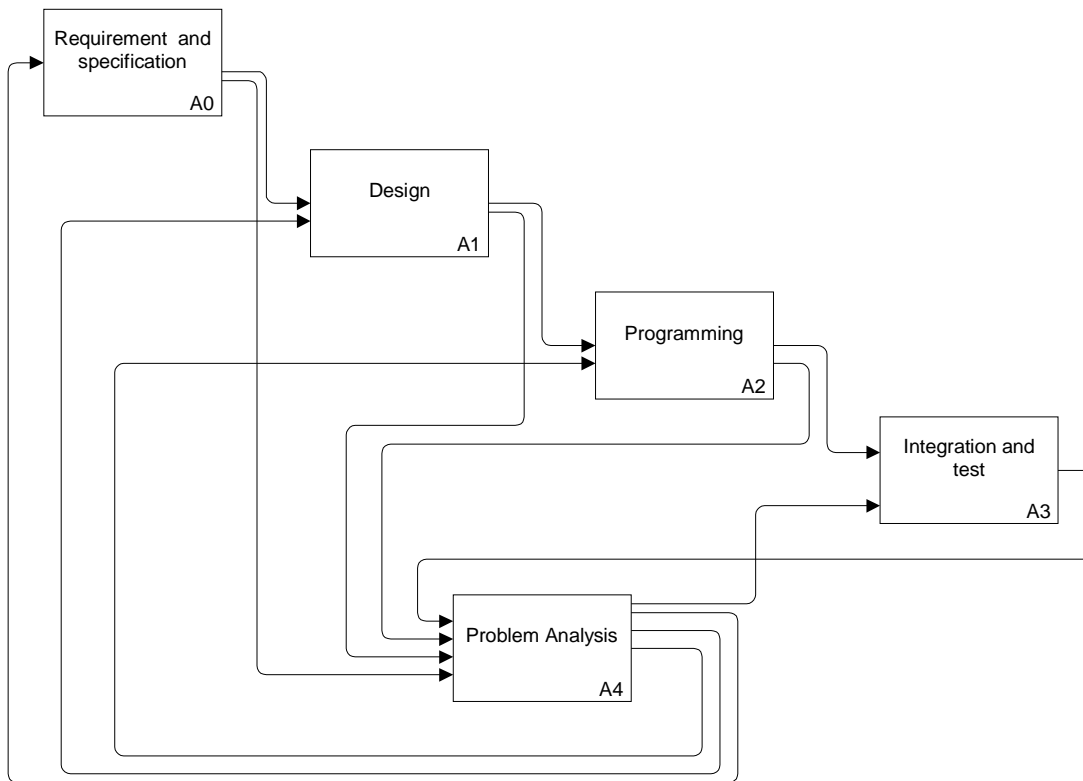


Figure 2. An Idef0 diagram.

3.2 UML for process modelling

As a low-level notation, given the strong relation of the DESS methodology with UML, it seems quite obvious to consider employing UML itself.

UML provides a level of formality which is much higher than IDEF0's. It is also more expressive:

- Data can be modelled by means of class definitions.
- Roles can be defined as new stereotypes.
- Samples of process instances can be given by means of sequence or activity diagrams.
- State diagrams and activity diagrams can specify the dynamic behavior of the process.
- ...

In practice the main process-oriented parts of UML are the class diagrams, and the state and activity diagrams.

A state diagram is shown in Figure 3.

Activity diagrams (Figure 4) are special cases of state diagrams, where states are mainly characterised by the activities carried out, and transitions are not generally caused by events, but by the completion of the activities.

Activity diagrams exist in different flavours. In particular, activity diagrams can be decorated with information concerning the flow of objects among activities, and responsibilities, indicated by the so-called swimlanes (Figure 5).

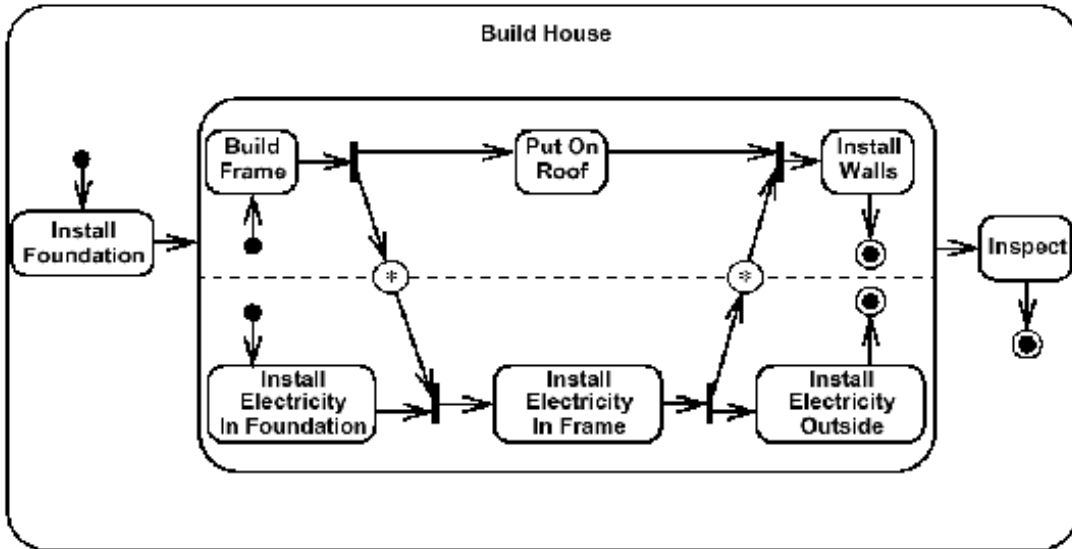


Figure 3. A state diagram.

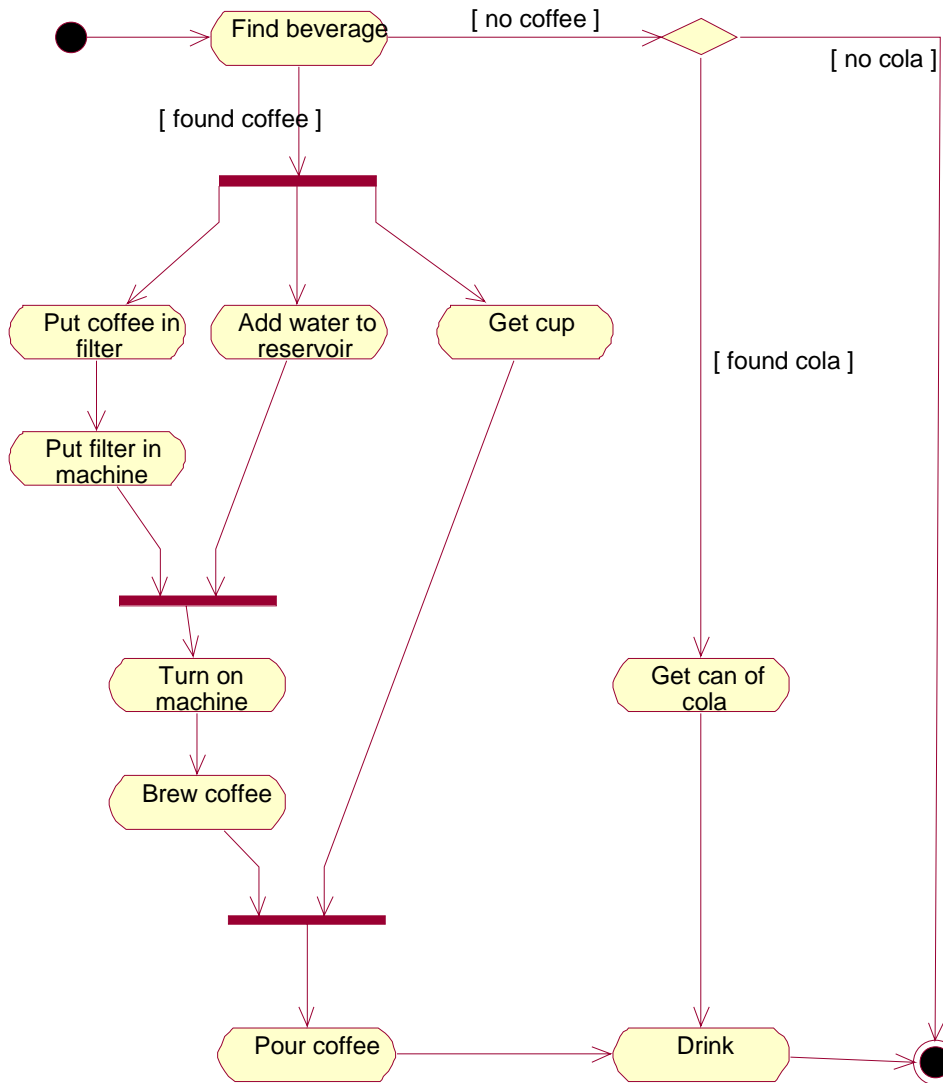


Figure 4. An activity diagram.

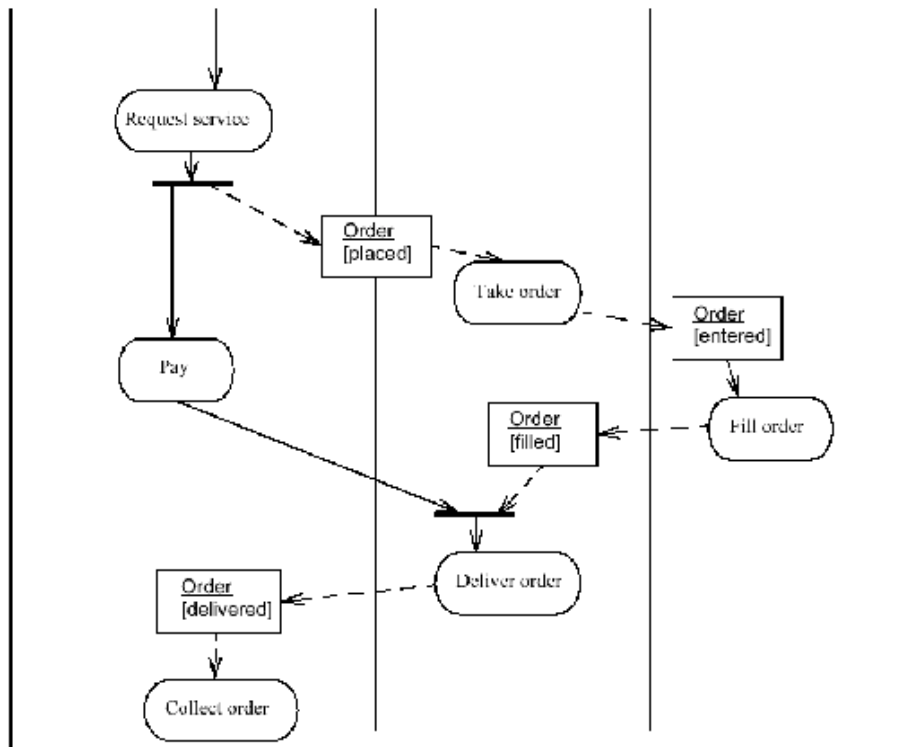


Figure 5. An activity diagram with objects and swimlanes.

3.2.1 On the formality of UML as a PML

Before adopting UML as a process modeling language (PML) it is necessary to be sure that it is able to carry enough detailed information to allow a process support environment to enact the process.

We have tested the expressiveness of the aforementioned diagrams for representing processes in a few sample process models (derived from past work by CEFRIEL). In particular we have employed UML to model the processes described in [2] and [3]. We have also converted sample UML diagrams into formal models (e.g., using Petri nets). In particular the diagram reported in Figure 3 was translated easily into the Petri Net reported in Figure 6.

We can therefore conclude that UML is fundamentally well suited for process modelling.

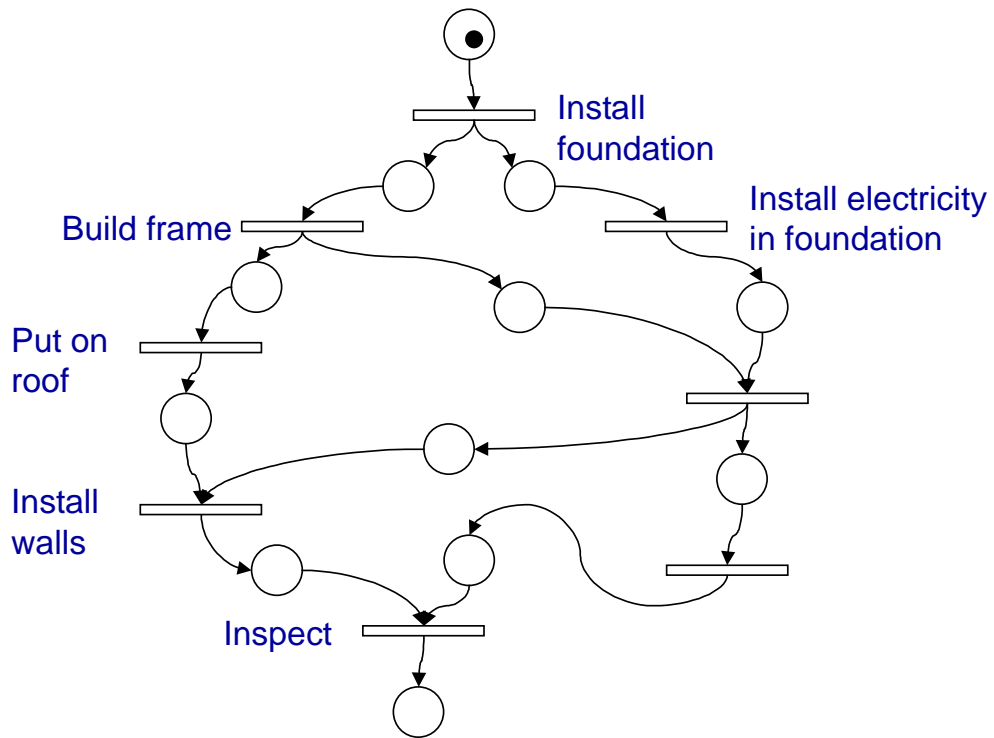


Figure 6. Petri net equivalent to the state diagram given in Figure 3.

Our conclusions confirm other previous evaluations [4].

4. Towards the DESS process model

4.1 Analysis of the existing models

Before proceeding to the definition of a specific DESS process model, we briefly considered the most frequently employed process models. The analysis was carried out at the highest level (often recognized as the life-cycle level). The goal is to take inspiration for the definition of a process model coherent with the current practice and the currently adopted best practices.

A first model to be considered is the so-called V Model, depicted in Figure 7. The V model is well-known and does not require further comments.

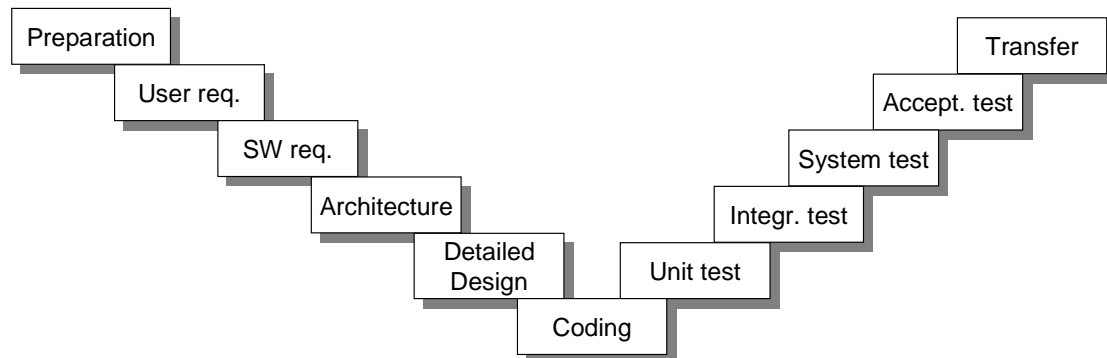


Figure 7. The V model.

Another model that deserves attention is the RUP (Rational Unified Process) model. The RUP [5] was introduced by the authors of UML as a set of guidelines derived by the best practices in UML-based developments. The RUP is an instantiation of the Spiral model by Barry Boehm, shown in Figure 8 and described in any software engineering book.

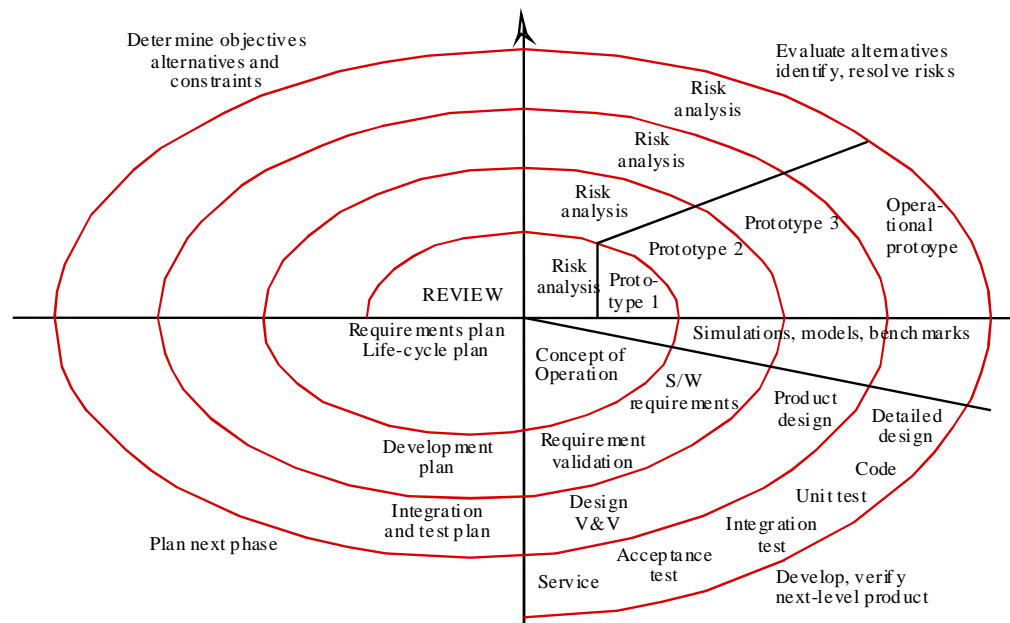


Figure 8. The spiral model.

The fundamental idea of RUP is to retain the organization of the spiral model (that prescribes to plan a phase, analyse risk, and do the step) in an iterative process (every macro-phase is organized in an incremental way, by means of several iterations).

The macro-phases of the RUP are described in Figure 9. Every phase is concluded by a milestone, corresponding to the delivery of a major artifact.

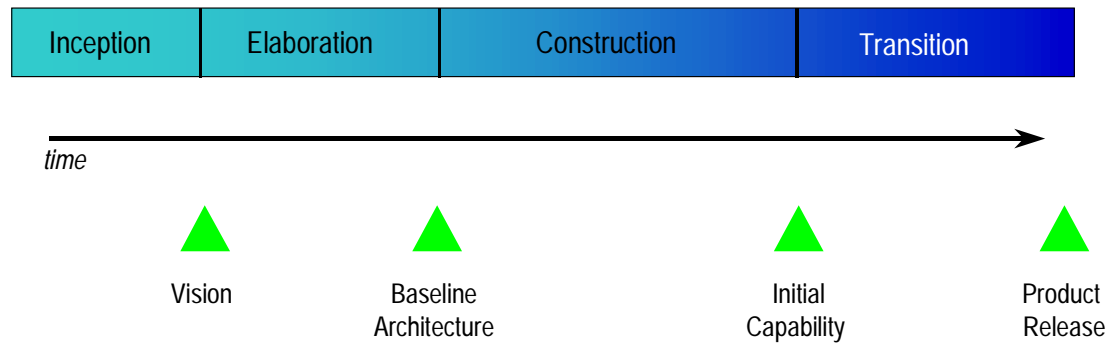


Figure 9. The phases of the RUP.

Iterations are due to the incremental construction of the system: at each iteration a new part of the system is added to what was built during the previous work. In addition, new requirements are taken into consideration, and known problems in the existing part are solved.

Iterations are highlighted in Figure 10.

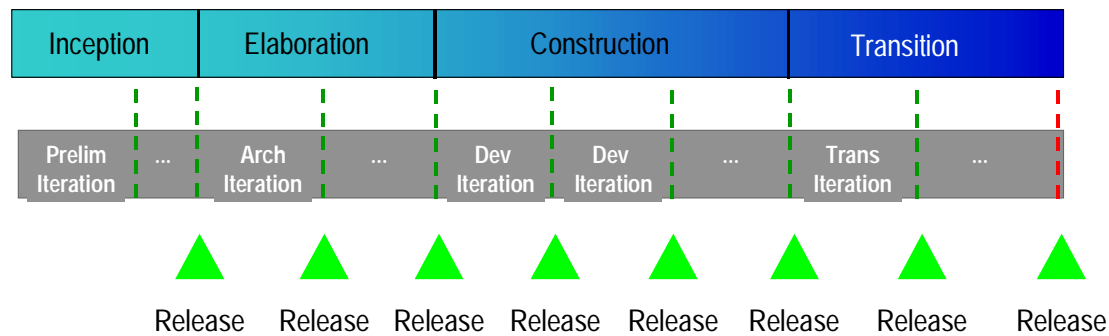


Figure 10. Phases and iterations of the RUP.

Of course the RUP must enclose the traditional activities described in any life-cycle model. Figure 11 shows how activities are carried out through phases and iterations.

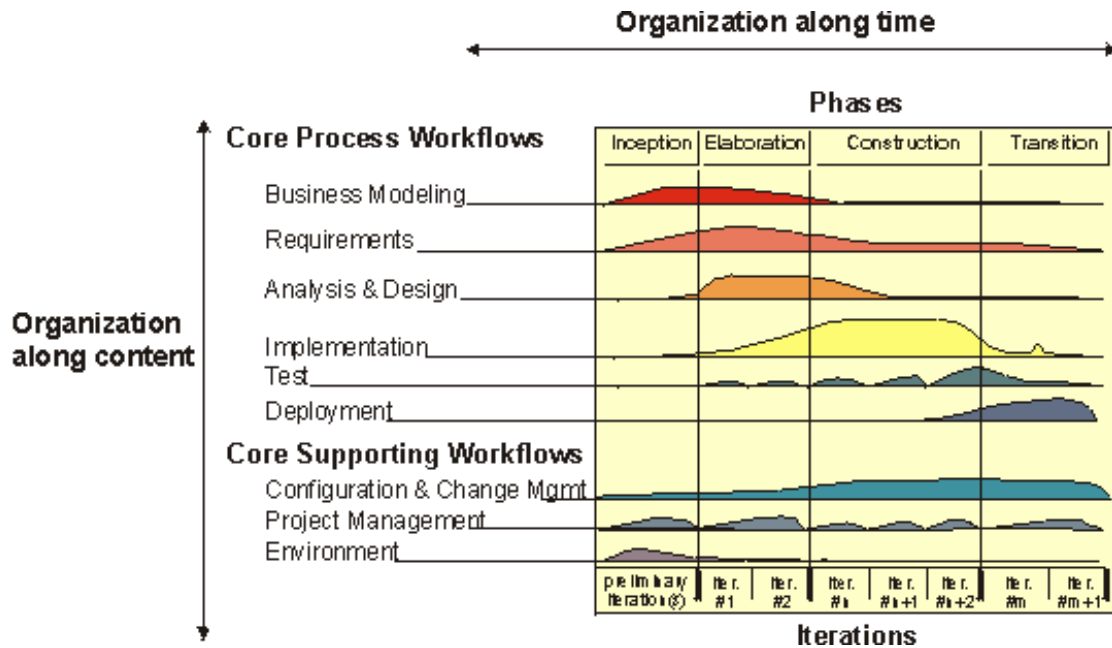


Figure 11. Phases and activities in the RUP.

The nature of RUP iterations is depicted in Figure 12, where it is easy to see the consistency of the RUP with spiral model.

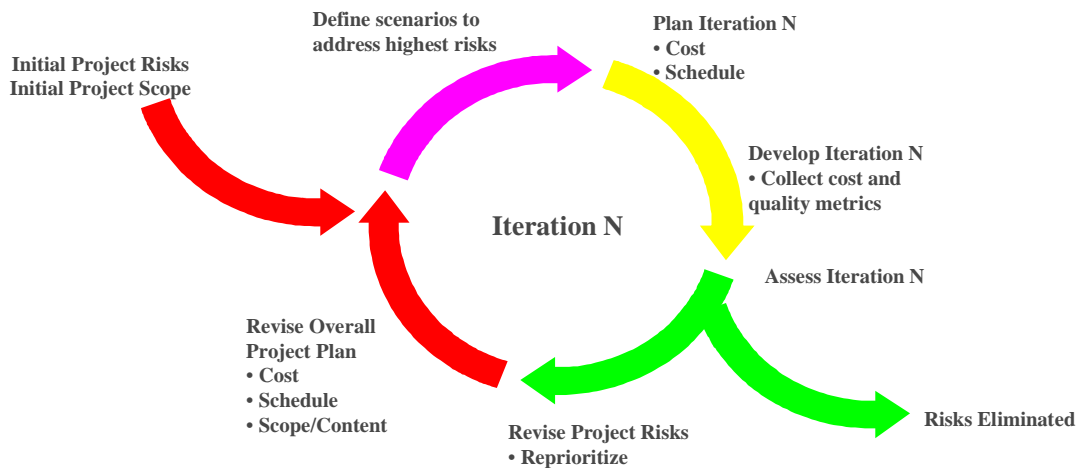


Figure 12. Iterations in the RUP.

Of course, RUP overall organization being iterative does not imply that every single activity is carried out in an iterative fashion. In particular, the phases within every iteration can be organized sequentially, e.g., as mini-waterfall processes. This situation is schematically described in Figure 13.

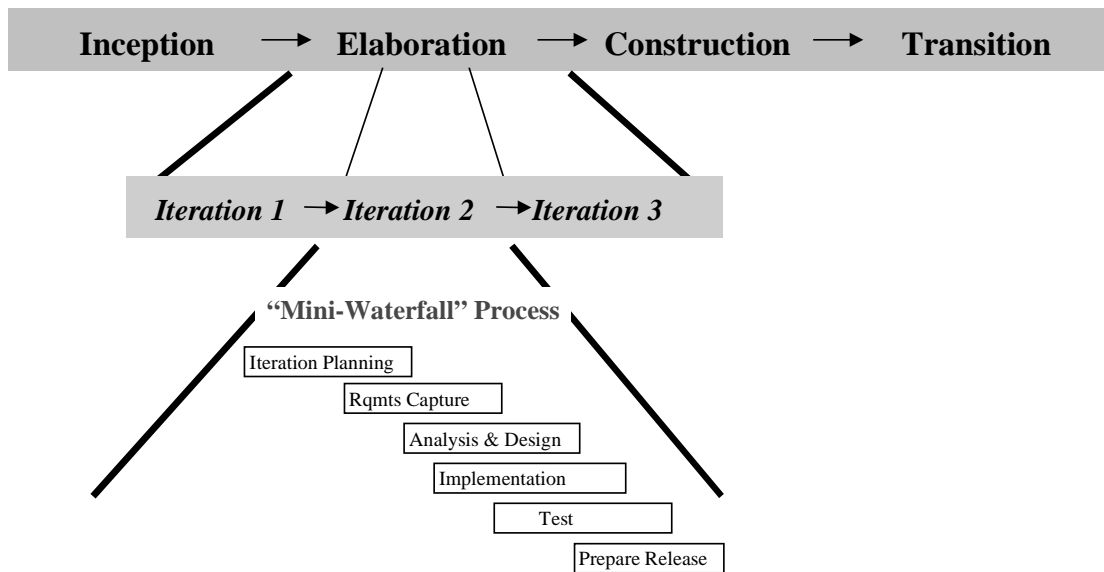


Figure 13. Iterations are internally organized as sequential processes.

A feature of the RUP which is well suited for DESS is that the RUP is not a detailed process model; rather it is a sort of generic framework in which every process owner is expected to insert his/her specific “process blocks”. This situation is synthesized in Figure 14.

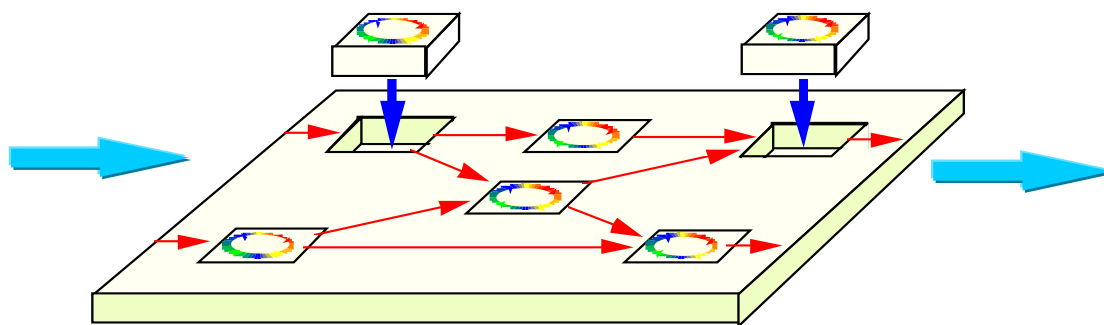


Figure 14. RUP is a customizable framework.

4.2 Specificity of DESS

In order to define the DESS process model, the most effective way is to simply adopt a surely effective and sufficiently general framework like the RUP, and adapt it to DESS-specific features.

In fact RUP has some limitations with respect to our goals:

- ❑ it does not make reference to real-time development
- ❑ it does not take into consideration critical developments
- ❑ it does not take into account component development
- ❑ ...

In particular, the first and most delicate point is that the iterative approach proposed by RUP must be coupled with the need for the robust development demanded by safe-critical applications. In DESS, when a change in requirements occurs, specifications have to be modified and checked (often formally), in order to verify that the basis of the system is still

safe and that constraints are satisfied. Therefore, architecture, components, etc. must be checked at every iteration, especially in presence of changes of requirements.

In the rest of the section we describe the points of RUP that must be adapted in order to comply with DESS goals.

4.2.1 User and system requirements

UML proposes the use case diagrams as the main representation of user requirements. In fact, the use cases play a central role in RUP itself, as every iteration takes into account additional use cases. The role of use cases in UML is synthetically described in Figure 15.

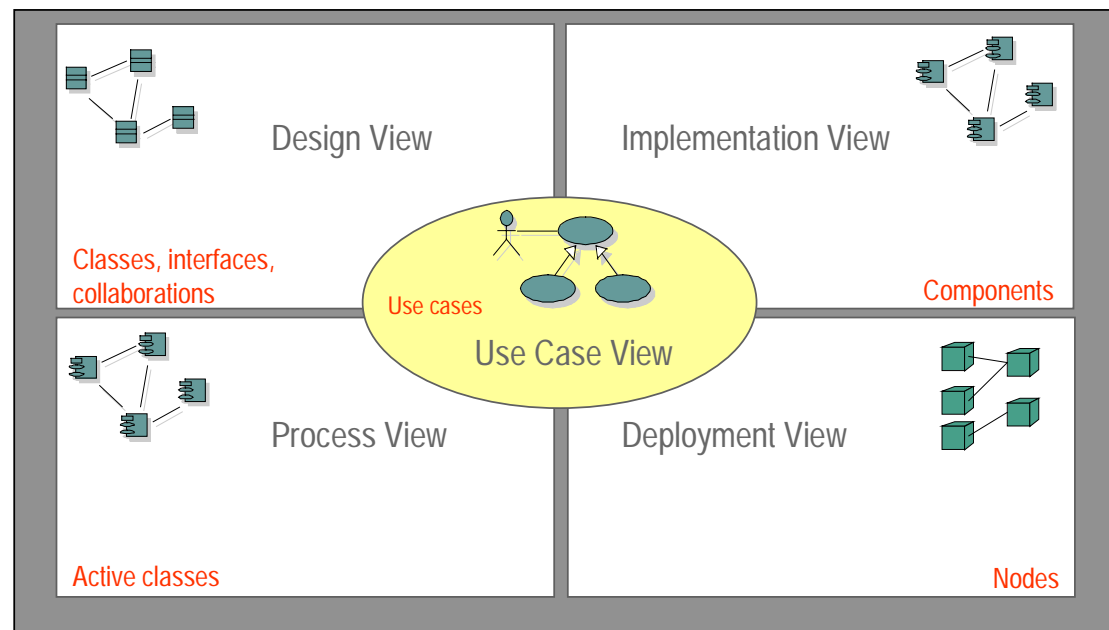


Figure 15. Use cases in UML-based development.

In the DESS process use cases can no longer be the central unifying view of the system, as they are too weak to represent the real-time features of the systems, the resource constraints, and the other non-functional requirements which are of outstanding importance in DESS.

4.2.2 Component-based approach

The RUP prescribes that the architecture of the system is basically defined at the end of the elaboration phase. This is a strong requirement for adopting an incremental approach. In fact it must be possible to perform increments without changing the architecture. Otherwise, a relatively small increment could require a big rework effort.

This requirement affects the DESS process by imposing that the components' organization is defined before the architecture is released. In other words, when we release the architecture, we must be sure that it will be able to accommodate the components that we plan to use.

This means that architecture-sensitive features of components are defined in the Elaboration phase, and are thoroughly studied before we proceed to the construction phase.

Details of components that do not affect the architecture can be defined during construction. Of course, the definition of components that are at the bottom of the system (i.e., those that are used by other parts of the system) should be defined as soon as possible.

4.2.3 Formal methods

Formal methods generally require that the specification of the whole system is analysed. This can be avoided if the method is incremental, but we do not consider this possibility here.

A fundamental hypothesis is that once a specification has been tested for correctness (with respect to some properties) it is then transformed into a correct implementation. Of course, changes in the user requirements (or the problem domain in general) will cause changes in specifications, and specifications' correctness will need to be checked again.

As a consequence, formal verifications are performed always in the Elaboration phase, and during the every construction phase which needs it (i.e., in presence of changes in the problem domain or in the responsibilities of the system).

4.3 Dealing with components

This section is inspired partly by [6] and partly by the outcome of WP4.

4.3.1 Concepts and terminology

In the rest of this section we give for granted that the reader is familiar with the concept of component and of connector.

On the contrary, we report below the definition of other terms that are used here.

Connector schemes. There are several way components can be connected. A connector scheme defines one of these possible ways.

Component Kits: a kit is a collection of components that have been designed to conform to a particular set of connector specifications. They don't necessarily come from one supplier, and haven't necessarily been built all at the same time; but they can be configured into working systems (or larger components) because they have been designed according to a well defined connector scheme specification.

Kit Architecture: the document that describes the connector schemes. In fact, it's the Kit Architecture that defines the essential nature of the kit, more than its population of components.

Business model: in general the kit architecture should include a business model. This takes into account that in non-trivial systems the inter-component communication protocols the objects transferred or referred to are not elementary values, but instead more complex and specific data (e.g., customers, orders, payments, etc.). For the components to act in a coherent way is fundamental that all the components have the same ideas about what these "things" are. Therefore, the kit architecture must include common definitions of these business objects, including the specification of their possible states, what operations may be applied to them, etc.

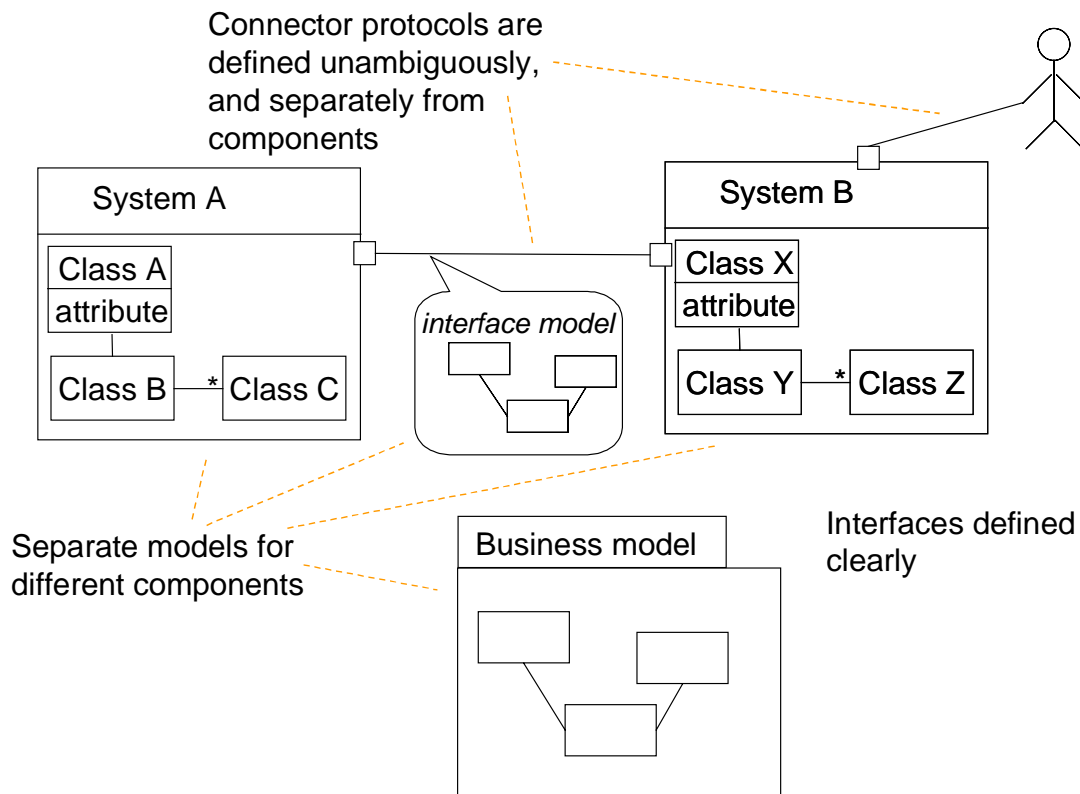


Figure 16. Component-based system.

Parts in a component package

A component is going to be assembled with others that its designer had no knowledge of. The component has to be provided with some documentation describing what it can be used for. It will be up to verify the behaviour of the component in the resulting configuration. A component should therefore come with test and monitoring software. In general a component should be equipped with:

Specifications: what the component “does” and what it needs.

Interface: where you plug the component in and how you plug further components (if any) into it

Executable: is generally not available.

Design information , i.e., how it works, what it's made of, as well as the source code, are generally private to the designer, not available to the assembler.

Sample usage and configuration descriptions

Validation suite: To test for conformance things you propose to plug into it; and to check its performance when operating in your context. The latter can often be limited to monitoring facilities, as it is difficult for the designer of the component to figure in which context the component will be used.

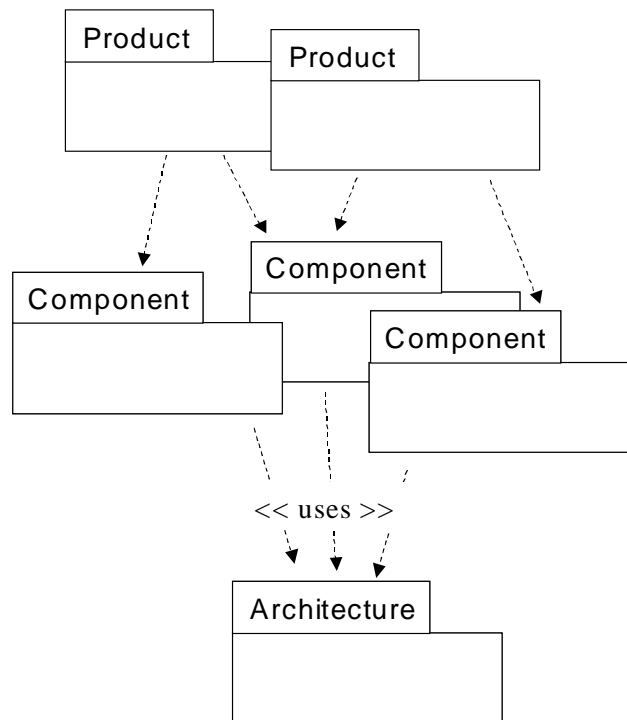


Figure 17. Component-based system layers.

4.3.2 Process

Roles of people involved in a component-based development

There is a separation of design roles in component based development, corresponding to the artefacts:

- Component Kit Architect designs the connectors.
- Component Designer creates components that fit the kit.
- Component Assembler creates end-products (or larger components) from the kit.

These different roles involve different skills and points of view:

The Component Assembler typically works with users to understand their requirements and then to build a system as satisfactorily and rapidly as possible. This is done by properly selecting components, by assembling them and testing the resulting system.

Component Design is a more careful activity, focusing on producing good general robust components that are likely to meet their specifications in a wide variety of configurations. The need to produce a component that behaves correctly when preconditions are satisfied, and acceptably even when they are not in a wide range of situations makes component design a delicate job, requiring high skill.

Component Architecture design is also a very skilled job, requiring strong in-sights into the future direction of the kit, and how to make it open and flexible.

It is also possible to identify the role of Component Strategist, who decides what components to populate the kit with, in order to be able to produce the products that will be needed in future. Of course, for components developed in the same organization where they will be used, this task is facilitated by the knowledge of the application domain.

Processes for Component Based Development

The key activities in the process have already been sketched above.

The process can thus be arranged according to the following guidelines:

- Component assembly, component design, and component kit architecture should be treated as separate activities. In general an organization's process could involve just a subset of such activities, e.g., some companies develop components for the market (they do not assemble) while others use components developed by third parties (they do not develop components). Separate life-cycles should exist for each activity, plus a cycle for developing the component repository.
- A specific activity should address the creation of a domain model. The domain model is useful both in the case a kit architecture has to be defined in order to allow the subsequent design of components and when you just have to select existing components (in the latter case the components must understand the domain objects). The domain model should not just be entities and relations: it should contain invariants and dynamic constraints too.
- The specifications and designs of the components are suited for short-cycle incremental development (and accompanying principles). A process along the lines of RUP would be all right.
- Formal notations can be used to describe components, interfaces and connections. Writing precise specifications helps reasoning clearly on the system. They are also good for prototyping, and for driving the test process.
- Other kinds of formalizations, such as pre- and post-conditions and invariants are useful to drive the coding and testing of the software.

Three development processes

Component-based development requires:

- One or more set of components. One could use different sets of components in different part of the application (vertical specialization) or for components corresponding to different abstractions (horizontal specialization).
- An architecture were to lay the components. The architecture defines the glue to compose components, the system views that the components share, the communication protocols, the services that the architecture provides to components, the common and general predefined interfaces.

The development of the components, the architecture and the application are logically separate activities, although in practice they could all be carried out in the same project.

So we can identify three development process:

1. architecture development process
2. component development process
3. system development process

These development processes obviously are deeply connected, they influence each others. These inter-dependencies are shown in Figure 18, which presents a very high-level view of a component-based development process. It is interesting to note that a thorough architecture definition&test cycle is necessary before proceeding to assembling components into the required application. This cycle could be oriented to the selection of one architecture among several available, or it could be oriented at the validation of a new architecture (or any combination of these two goals). It is also worthwhile noticing that the architecture selection

The requisites for the components defined according to systems requirements, and with respect to the architecture. It is possible that the architecture itself needs to be equipped with some basic components which provide architecture-level services.

In general there are several ways to map the requirements for the system to requirements for components. In fact a system can be organized (decomposed) into components in several ways. Moreover, components can be made more generic than strictly needed, in order to be more easily reusable in different contexts. Of course developing a generic component is generally more expensive than developing a specific one: genericity becomes convenient when components are actually reused. However, when it is difficult to foresee the way a component will be reused, it becomes very difficult to decide the correct level of genericity.

Architecture

A good architecture needs a careful definition, evolution, consolidation and validation. The problems that a constantly changing architecture poses can void the effort spent to create reusable parts: it is quite obvious that a big change in the architecture can break the assumptions of the various components already created. Maintaining backward compatibility with previous versions is often impossible (especially in the early phases of the architecture development process) and it will however disrupt the cleanness of the architecture.

The need to avoid these effects shapes the development process of the architecture, that will in its turn affect the other two processes.

In the early stages of the architecture definition “good and reusable” components will hardly be developed, because of the following risks:

- a –very likely– change in the architecture will make the first components unusable;
- little experience with the architecture;
- no or little services are offered by the architecture in its early stages: it is therefore not clear whether the services should be incorporated in components or they will be provided by the architecture (in such a case it is however not known how they will be provided).

Only when the architecture finally arrives in a point where the change rate becomes acceptably low components repositories are populated, and the time employed for component development is largely repaid by the use of components in different contexts.

As usual, the whole process is largely customizable, in particular we could have just a few projects where the main focus is the evolution and consolidation of the architecture, or, in the other way, many application-oriented projects where just a small percent of time is devoted to the development of a common architecture.

4.3.3 The role of formal notations in component-based development

Although semi-formal notations such as UML can be employed in CBD as in traditional development, it is interesting to note how formal notations can help the development process based on components.

Traditional role of formal methods

Some activities in a CBD are not different from their traditional counterparts. For instance, requirements analysis and system testing do not depend on the component-orientedness of the process. This is a very important observation, as several formal methods have their main application just in requirements specifications and testing.

So, you can use a formal notation to represent the requirement of the system, then you can apply some formal method to automatically (or semi-automatically) derive test cases, independently from the usage of components in the design and implementation phases.

Formal methods for component-based development

Given this, can we exploit formal methods in the very component-oriented activities?

It is in fact possible, as the following process model briefly sketches:

1. Specifications are written in some formal language (if requirements have also been written formally, it is generally relatively easy to derive specifications from them). In this stage component-orientation is still not considered.
2. The code of components is often not available. Since the designer in charge of assembling components into a piece of application must be taught about the capabilities of the components, they are generally provided with some sort of specifications, which describe their behaviour. That is, when a component is developed, it is released with a specification that describes it. Let us suppose that these specifications are written formally (if they are not, we can still write them, based on the informal specs provided by the component producer).
3. Similarly, let the formal specification of the architecture be available.
4. It should be possible to assemble component specifications and the architecture specification, thus obtaining a formal description of the behaviour of the resulting system. This specification of the overall system should allow the designer to verify whether the resulting system has the same properties as the component-unaware specification mentioned at point 1 above.

This kind of process has several advantages:

- It allows the designer to try several assemblies without the need to actually get and combining the components, but using just their specifications. This is course a big advantage whenever you can test the behaviour of a component that otherwise you should buy, or when the component is still being developed.
- It allows the designer to test the properties of the resulting system. This is of fundamental importance for real-time systems. In other words, you test that the way you assembled components is correct with respect to the requirements.
- If each component implementation satisfies perfectly its specification, then the resulting implemented system will be equivalent to the specification of the whole system, whose properties have been proven correct (see the preceding point). This implies that the implementation is provably correct. Again, this is an extremely remarkable result, especially for real-time systems. Note that the hypothesis that the implementation of each component satisfies its specifications is quite demanding. Nevertheless, components, being used and reused in several different products are end up being thoroughly tested in several different contexts, and this provides reasonable guarantee that they do satisfy their specifications.

5. DESS methodology

DESS methodology is defined in D.1 [13], DESS methodology is roughly based on the reflections made in 4 Towards the DESS process model.

DESS methodology borrows from the Unified Process (UP) model, reusing concepts defined in UP to describe the process:

- artefact: a tangible piece of information that is created, changed, and used by workers when performing activities, represents an area of responsibility, and is likely to be put under separate version control.
- activity: tangible unit of work performed by a worker in a workflow that implies a well-defined responsibility for the worker, yields a well-defined result (a set of artefacts) based on a well-defined input (another set of artefacts), and represents a unit of work with crisply defined boundaries that is likely to be referred to in a project plan when tasks are assigned to individuals.
- workflow: a realization of (a part of) a business use case. Can be described in terms of activity diagrams that include participating workers, the activities they perform, and the artefacts they produce.

The development methodology process is based on the V-model. Mixing the iterative and incremental life cycles present in UP with the well defined separation of artifacts and activities present in the standard V-model. The V-model, however, is expanded to actually comprehend 3 separate V-models working in parallel:

- Realization Workflow
- Validation and Verification Workflow
- Requirements Management Workflow

5.1 Realization Workflow

The boxes that relate to the V-shaped DESS Realization Workflow V (Figure 19) are workflows. Workflows have artifacts as inputs and outputs. Each workflow contains activities constructing new output artifacts from previous input artifacts.

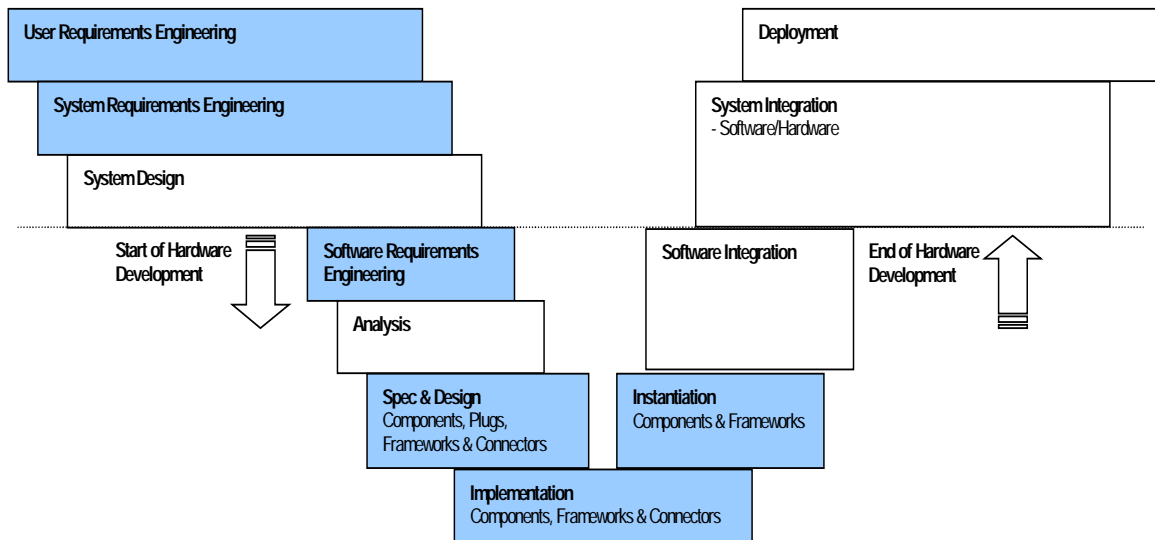


Figure 19. DESS Realization Workflow

There are two routes of information flow between boxes in the Realization Workflow V:

- information flows along the Workflow V, starting in the upper-left corner and ending in the upper-right corner. The output artifacts of each box are the input artifacts for the next one. This route corresponds to the realization path from user requirements to deployed system: each new artifact is a further development of an earlier one.
- information flow across the model. Output artifacts on the left branch may be input artifacts for the corresponding process on the right branch. This route corresponds to the fact that information, used to achieve decomposition of the system during specification is used again to compose the implemented parts.

5.2 Validation & Verification Workflow

The DESS Validation & Verification Workflow V is congruent with the Realization Workflow V in that it is also V-shaped and furthermore, that for each box in the Realization Workflow V there is a corresponding one in the V&V Workflow V. The phases in the V&V Workflow V are, similarly as for the Realization V, processes that now have artefacts as inputs and outputs.

To arrive at the interpretation of the V&V Workflow V, it should be kept in mind that the aim is to assure that development was done right, i.e., that the construction of artefacts following the steps in the Realization V was done right, i.e., that the information used there and the construction steps carried out are in correspondence. This observation explains the congruence between the models: a box in the V&V Workflow V to check the correctness of the construction carried out in the corresponding box in the Realization V. There are again several routes of information flow, most importantly between Realization and V&V Workflow V.

Firstly, artifacts are used as input for the corresponding processes. This input from the Realization V suffices for the left branch of the V&V Workflow V.

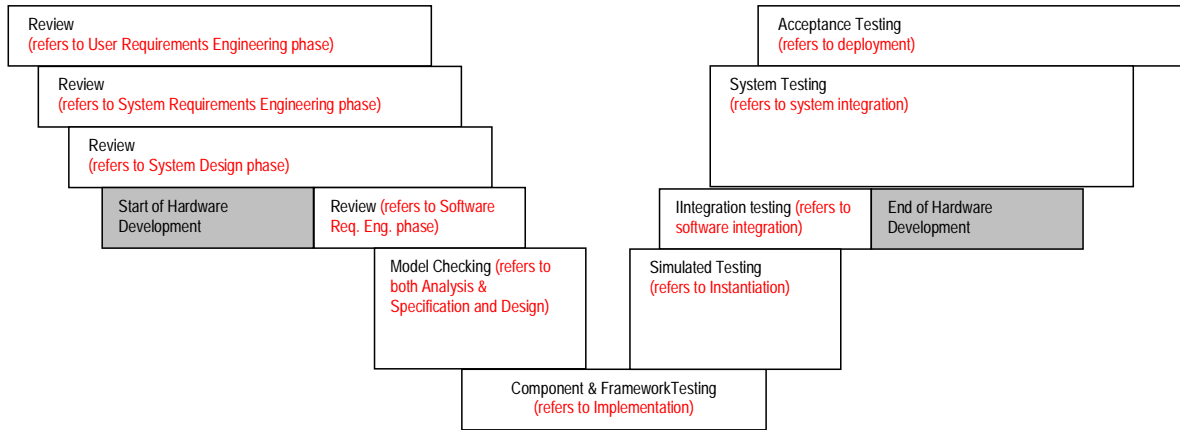


Figure 20. DESS Validation & Verification Workflow

5.3 Requirements Management Workflow

Requirements Management involves establishing and maintaining an agreement with the customer on the requirements for the software project. This agreement is referred to as the "system requirements allocated to the software. The agreement covers both the technical and non-technical requirements. The agreement forms the basis for estimating, planning, performing, and tracking the software project's activities throughout the software life cycle.

Within the constraints of the project, the software-engineering group takes appropriate steps to ensure that the system requirements allocated to software, which they are responsible for addressing, are documented and controlled.

Whenever the system requirements allocated to software are changed, the affected software plans, work products, and activities are adjusted to remain consistent with the updated requirements.

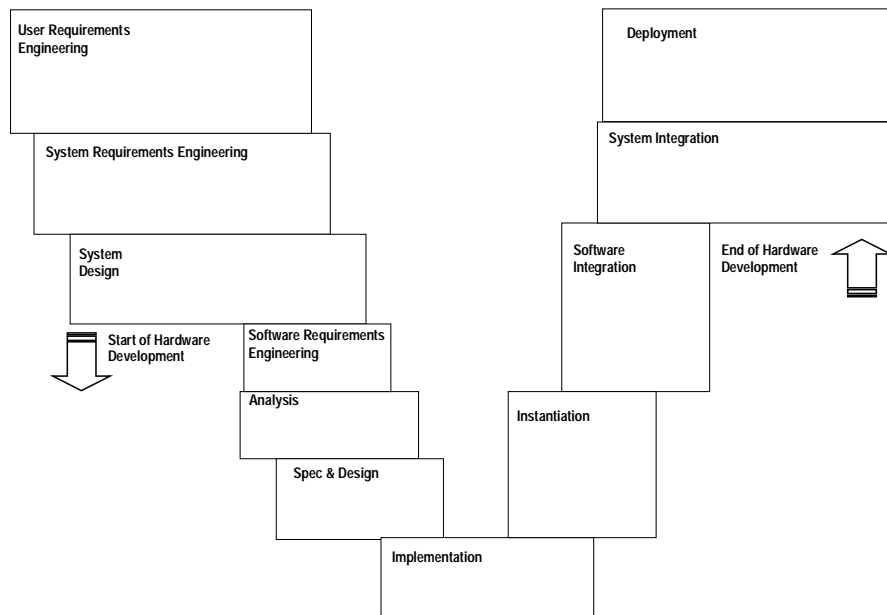


Figure 21. DESS Requirements Management Workflow

6. DESS software development process

The development process can be viewed as a specific instantiation of the process defined in the DESS methodology.

The process in DESS methodology is based on UP and organized in 3 parallel V models (Realization, Validation & Verification, Requirements Management). Flexibility in the overall structure is given by UP, which is a meta-process, that is UP can be heavily customized to adapt to the specific software process needs.

6.1 Customization

Customization of UP is possible following various aspects. The most interesting ones, from the DESS methodology point of view are:

- Order of activities
- Cyclical phases
- Format of Artifacts produced

6.1.1 Order of Activities

The order in which the various Activities are presented in the three parallel V models are not strictly fixed, the V model shouldn't be seen as sequential steps to be taken to enact a strictly water fall process.

The V models only represent the logical structure of the three main workflows defined in DESS, inside each workflow the activities can be executed in any order, the obvious constraints are the input artifacts of the activity: the input artifacts must be in a state acceptable for the activity to proceed.

6.1.2 Cyclical phases

The activities detailed in each Workflow can be executed more than one time. So it is possible to instantiate a software development process iterative and incremental:

- Iterative: when an activity is executed adding with new input artifacts (or with new parts added to the input artifacts) to produce new (parts of) output artifacts.
- Incremental: when an activity is executed with refined input artifacts to produce refined output artifacts.

6.1.3 Format of Artifacts produced

The format and completeness of the artifacts produced are fully customizable. Each artifact produced can have a different level of formality. For instance, a requirements document can range from a full specification in a formal language using a well structured document to a collection of informal use cases and scenarios detailed in informal and loosely structured handwritten sheets of paper. An UML diagram can range from a well defined diagram written with a UML-aware tool, fully explained, documented and tracked with a configuration management tool (again, UML-aware) to an incomplete drawing in a blackboard. An artifact can be as informal as the simple spoken language can be.

This kind of customization enables the process to successfully treat risky and heavy projects as well as easy, very light and quick projects.

6.2 DESS process models

We will present some example abstract cases where the DESS methodology can be used, and how the process should change changing the context of use of DESS methodology.

6.2.1 Case 1

6.2.1.1 Context

The software system to build is a critical system, it is supposed to have rigid constraints and hard real time constraints. The consequences of a system failure or a system that do not respect the constraints are usually dangerous.

The problem domain is considered stable and well known, as a consequence the risks bounded to unexpected situations are low, and the process does not directly needs short cycles of analysis, design, implementation, test; this means that the process will be “moderately” iterative and incremental.

6.2.1.2 Process

Our main focus is concentrated in the workflows that involves components, since DESS methodology and process do not changes significantly the other aspects of a software process.

The normal component life cycles are shown in Figure 22. The flows of activity are almost linear, as we already noticed. The details of Components development and design are in Figure 23 and Figure 24.

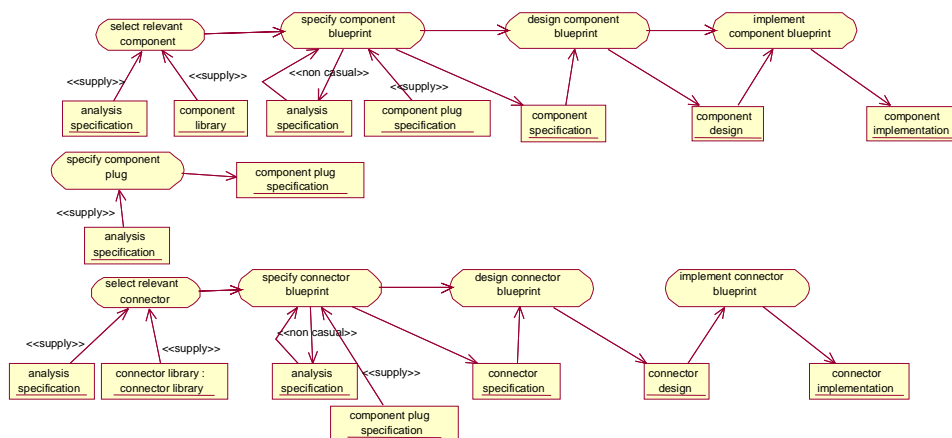


Figure 22. Component life cycles

During the specification of a component blueprint (or a component plug), even if the relevant analysis phase is quite complete (due to the well known problem domain) can arise the necessity to specify a new component plug or a new connector blueprint. These actions modify the analysis specification (as shown in Figure 22) and new activities must be started to complete the current activity (as shown in Figure 23).

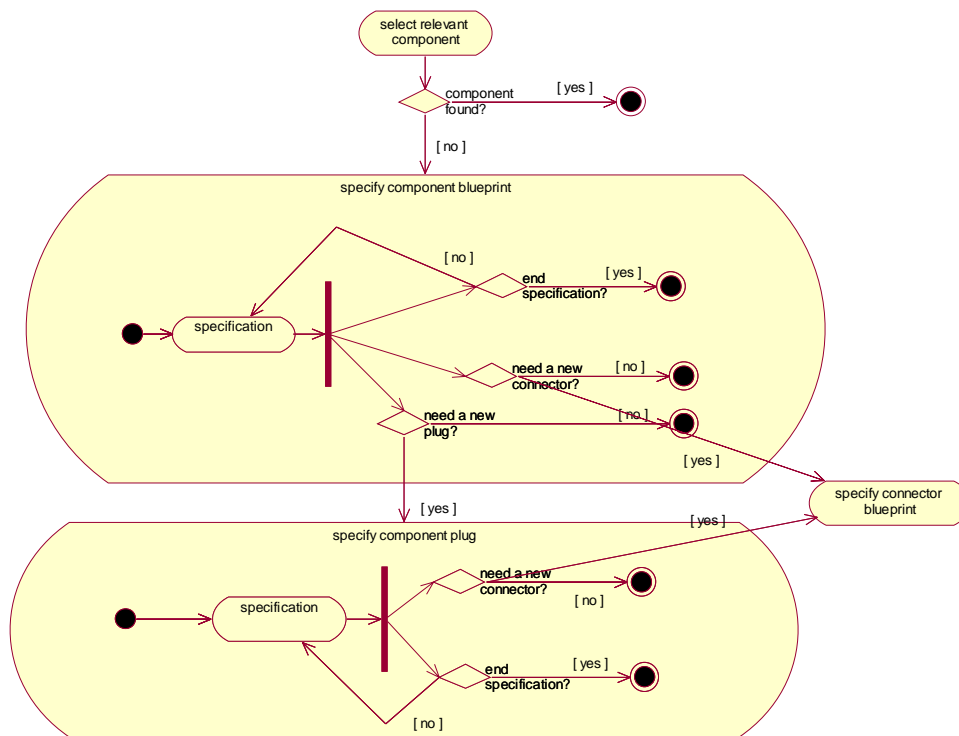


Figure 23. Components construction life cycle expanded

On the other end, when we are designing a component (or a connector) according to the component blueprint associated new subcomponents (connectors, plugs and components) may be needed to complete the design, further the new specifications can modify the analysis specification. The same argument can be applied when we are designing a connector. The processes are in Figure 24.

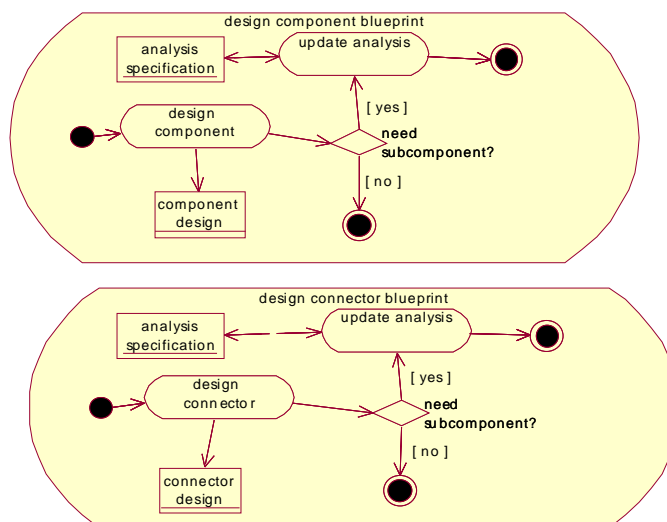


Figure 24. Components Design details

The Verification and Validation activities can be kept separated or glued with the Realization activities, that is they can be sequential or run in parallel. The Process is detailed in Figure 25.

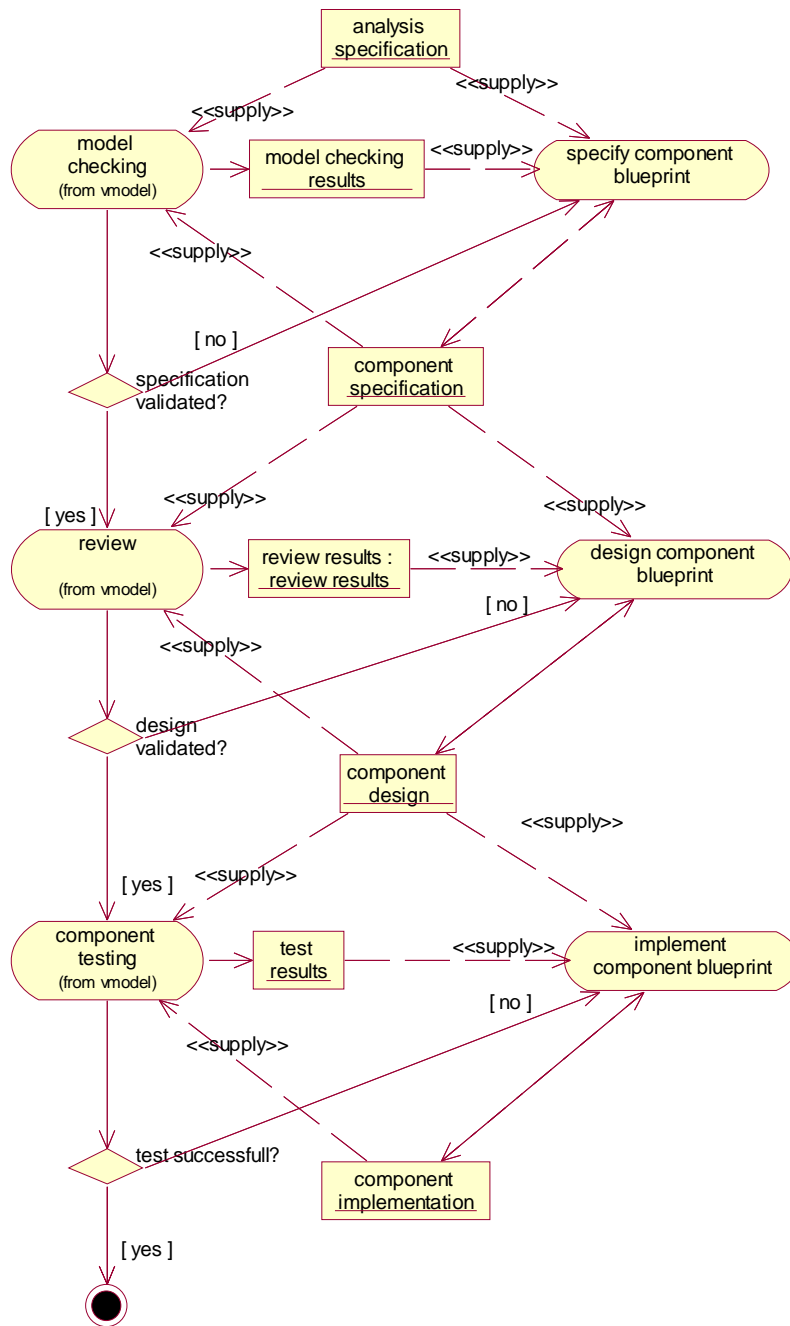


Figure 25. Verification & Validation workflow for components.

7. References

- [1] Integration Definition for Function Modeling, *IDEF0*. Federal Information Processing Standards Publications, December 1993.
- [2] S. Bandinelli, A. Fuggetta, L. Lavazza, M. Loi, and G.P. Picco. Modelling and improving an industrial software process. *IEEE Transaction on Software Engineering*, vol.21, n.5, May 1995.
- [3] G. Cugola, E. Di Nitto, A. Fuggetta, Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan April 1998.
- [4] Frank Berkers, Business Process Modeling with UML: Practical Experiences with Rational Rose, IPA Spring Days 2000 on UML.
- [5] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley 1999.
- [6] Alan Cameron Wills, Designing Component Kits and Architectures with *Catalysis*, TriReme International Ltd, <http://www.trireme.com>
- [7] B. P. Douglass Real-Time UML, Addison Wesley, 1998.
- [8] OMG, Unified Modeling Language Specification, Version 1.3, First Edition, March 2000.
- [9] B. Selic, G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modeling, Wiley, 1999.
- [10] The precise UML group, <http://www.cs.york.ac.uk/puml/>
- [11] Catalysis, <http://www.trireme.com>
- [12] C.A. Gunter, E.L. Gunter, M. Jackson, P. Zave, "A Reference Model for Requirements and Specifications", *IEEE Software*, vol. 17, n. 3, May-June 2000.
- [13] DESS, "D.1 The DESS Methodology", 2001.

8. Appendix A: the questionnaire

This appendix reports the questionnaire sent to all partners on July 13th 2000.

8.1 Introduction

This questionnaire is to be used in the preparatory actions for WP5 but also for WP1.2. WP5 aims at the following:

- Investigate the existing formalisms that could be used for representing the process models. Investigation should span both the formalisms currently used within the consortium (if any) and the existing standards. Among the latter are outstanding IDEF0 (or IDEF3) and RUP (the Rational Unified Process). RUP is not a standard, but it is quite easy to predict that it will become very popular.
- Investigate the existing formalisms that could be used for representing the process artifacts (i.e., specifications, design document, code, test cases, error reports, etc.). Investigation should span both the formalisms currently used within the consortium (if any) and the existing standards. Among the latter are outstanding the Entity/Relationship diagrams (in all the numerous variants) and UML.
- Produce process and product models concerning the current *actual* situation in the partners' organizations at various granularity levels. Note that it is important to model the current situations and the desired situation (i.e., the one that DESS should help to achieve). Note: the models we need are those in actual use. Models described in books that are never read are not relevant. The models should include as far as possible indications concerning tools used (especially environments, configuration management systems, etc.), constraints, strategic goals, standards adopted in production, etc.
- Investigate the existing tools and environments that are available (commercially or as research products) in order to support the development process.
- Collect information on tools currently used within the consortium and/or preferences and constraints concerning tools.

Note that the last two points have already been assessed in less detail in WP1.2.

The questionnaire is organized in two main parts: the first one aims at representing the current situation of DESS partners' development process; the second part aims at collecting the requirements for the DESS process to be defined and supported.

How to use the questionnaire

Questions are expressed in a rather free form. There are no predefined answers. In general it is possible to answer several questions by means of a few paragraphs. The goal is to understand the situation, so the important issue is that the global picture is clear, even though some question is not clearly/fully answered.

Please read all the questions before starting to answer.

The whole set of answers is expected to range from 3 to 10 pages. Additional material taken from organizations' standard documentation is welcome.

8.2 Current situation

8.2.1 Product

- 1 What kinds of product does your organization produce?

- 2 Does your organization use a standard notation to describe products?
- 3 Is there a reference definition for products in your organization?
- 4 Does your organization have a reference description for the artifacts (requirements, specifications, design documents, etc.) produced/used throughout the whole development process?
- 5 Can you provide a few samples of these descriptions?

8.2.2 Process

- 1 Describe structure and phasing of the process, e.g., waterfall, V-model, iterative approaches like the spiral process (repeatedly going through phases like Analysis/Design/Coding/Testing), other, differently structured, incremental approaches.

Also indicate the larger phasing of the process, i.e., the kind of larger milestones that are reached in the process, e.g., those occurring after some iterations in the iterative approach.

- 2 Indicate time allocated (percentage) for the different activities at different stages of the process (e.g. for Analysis/Design/Coding/Testing in the different stages of the spiral model).
- 3 Broad description of the main (formal) notation and tooling used in the larger phases of the process. It also should be indicated where informal notation is used.
- 4 Has your organization defined a reference process?
- 5 Does your organization adapt a reference process to the specific projects depending on projects' features and needs?
- 6 Does your organization's process conform to any standard?
- 7 Has your organization been certified with respect to process quality (ISO9000, CMM, etc.)?
- 8 Own assessment of how what you consider as the best practices [see, e.g., Kruchten00] are followed. Preferably for each item a few explanatory lines as well as an assessment of current status:

Is there a prescribed process? (P - yes/no.)

Is it followed? (F - yes/partially/no.)

Is it efficient and effective? (E - satisfactory (+), moderate ([]), to be improved (-).)

- Iterative development (Example answer: P(y), F(p), E([]).)

- Explicit requirement management

- Use of Component Based architectures

- Visual modeling (e.g., class diagrams)

- Explicit software quality verification (not just correctness, also structure, documentation etc.)

- Controlling changes.

- 9 Own assessment on current status of handling key aspects for DESS.

- Real-Time

- Resource constraints

- Embedded system aspects
 - Formal methods
 - Component-Based
 - Code generation
- Validation and Testing.
- 10 Is your organization pursuing a process improvement program? If so, following which methodology?
 - 11 Does your organization carry out regularly process/product measurement?

8.3 Requirements for the DESS process

- 1 What are the issues of process development that you would like the DESS process to address (and thus –hopefully– improve)?
- 2 In particular, what technical and managerial activities should be supported by the DESS process?
- 3 What of the existing tools do you consider important to integrate in the DESS process support?
- 4 What are your needs in terms of cooperation and communication?
- 5 What are your needs in terms of (geographically) distributed development management?
- 6 What tools would you like the DESS process support to integrate (or at least to support in some way)?

We would also welcome a short comment on current status and desirability of formal methods in combination with automated verification.

PLEASE INDICATE WHAT YOU WOULD LIKE TO BE ADDED TO THIS QUESTIONNAIRE.

For example, if in 8 or 9 (section 8.2.2) you find that further items would be useful, please add such.

8.3.1.1 References

Kruchten00 P. Kruchten, The Rational Unified Process, An Introduction, Addison-Wesley, 2000.

Please, remember that possibly helpful reading material is available at

http://www.cefriel.it/~dess/PrivatePart/document_WP5.htm