Integration of the Different Elements of Object-Oriented Software Engineering into a Conceptual Framework: The 3D-Model

Andreas Zamperoni

Leiden University
Dept. of Computer Science
Niels Bohrweg 2
NL-2333 CA Leiden

Tel.: ++31 - 71 - 27 7103
Fax.: ++31 - 71 - 27 6985
email: zamper@wi.leidenuniv.nl

May 1994

Abstract

This report describes a conceptual framework for supporting software engineering. The approach relies on object-oriented concepts, but is not limited to object-oriented software engineering. The basic idea is the notion of a three-dimensional graph structure, which serves to capture and to classify development activities and their resulting documents (the *elements* of software engineering). The three dimensions - views, components, development process - of a (software) system serve as fundamental organizing principles, according to which all elements are placed into a uniform, intuitive, but formal meta-structure. From this meta-structure - the so-called *3D-model* - relevant development elements (activities, specifications) and their relationships with other elements can be derived, together with suitable notations and a process. Hence, the 3D-model offers a mechanism to define an effective and tailored software development process, which offers freedom to select its notations, methods, architectures, briefly, all its essential ingredients, to fit in the current development situation.

Contents

Chapter 1

Introduction

In the last years, two related subjects have gained more and more importance in the software engineering world:

- the maturing of the object-oriented paradigm in development and programming

and, with that new approach

- the emerging need for new software processes which match the particularities of the object-oriented paradigm.

Many textbooks and other publications on software engineering introduce new, **object-oriented development methodologies** [Boo91, JCJO92, RBP+91, SM88, SM92, NGT92, CAB+94]. But experience shows that if a developer has decided for and adopts one of these textbook methodologies, he will inevitably come to a point when the chosen methodology fails to capture certain aspects important in the specific development situation or environment. On the other hand, he might be forced to stress certain details of the development, e.g., produce certain specifications, although they are not relevant for his actual development effort. This holds especially for the development of innovative, experimental software products, where requirements and design might not be complete before coding starts [Ger93]. An example for such a development situation is given at *TNO Institute of Applied Geo-Science, Research Group for Information Systems for Oil & Gas Exploration*, which funded this study.

Developing software is quite an individual process, depending on the application domain, the structure of the organization, its management style, the relative importance of the various software qualities, the team involved, the expertise about the problem domain, and other factors, as described in [REE89]. As consequence, the need for a **framework** which is **tailorable** in the selection of its notations, methods, architectures - briefly in all its essential ingredients - arises.

Example: The concept of 'static structure' of a system is an example for the variety of concepts in software engineering. Static structure, i.e., the set of objects and relationships between objects can be looked at and described from different perspectives. Inheritance, generalization/specialization structures can be the focus of the description of static structure, but also synchronous and asynchronous communication. The choice which of these two different types of relationships - hence perspectives - to emphasize is biased by factors like the application domain, the development stage, or the type of target system.

Most of the offered software engineering methods are biased by the particular life cycle of their authors' development situation. Such a **methodology bias** can be difficult to overcome, especially if a software engineering environment, which supports a specific method through the whole life cycle, is used. On the other hand, approaches, which try to describe the development from a more independent point of view, are often too abstract to provide more than guidelines of how to proceed and succeed in software development. In those approaches, terms like the *traditional waterfall model* or the *spiral model* represent only models, i.e., **abstractions**, of software engineering, not practicable methods or standards, which are needed for successful large-scale software development.

Recently, efforts to realize this approach of variability in software engineering have been undertaken by providing tool sets which support multiple methods. These efforts can be summarized by the term **meta-CASE technology**. By offering a larger range of tools, meta-CASE tries to provide the developer with an enhanced means of software development (e.g., ObjectMaker, TBK [Ald91]). The problem is that by providing a tool that supports different notations, the underlying idea of an enhanced approach to software engineering is not automatically supported. The same variability in selecting also the suitable software development process and the proper software engineering principles is required, but goes beyond the state-of-the-art meta-CASE. An **integral approach to adaptable software engineering** would enable a developer to identify from a larger set precisely those activities and elements (e.g., views of the system, software architectures, etc.) he regarded as important in his development situation, and a mechanism would:

a. offer him possible, suitable notations to specify those elements, and

b. show him how the identified elements are related to each other, i.e., how they depend from other specifications within the development.

This mechanism is, at first place, **a conceptual framework** of software engineering, but, analogously to the meta-CASE tools, serves as conceptual basis for a system that supports the developer in his engineering work.

Developing such a conceptual mechanism for the issues of software engineering takes several steps. The basic units of development and the activities to produce them have to be identified, classified and described. They form the constituents of the software development. Subsequently, the potential constituents have to be integrated into a framework that captures them all in a uniform way. This framework can be modeled in terms of a **graph**, where the nodes represent the constituents of the software development, and therefore also the activities to realize those constituents. The edges represent the relationships between the constituents, and therefore indicate different types of dependencies between the constituents. From this framework, a sound set of constituents and activities for a certain application area can be derived by specifying a subset of the above mentioned graph. Using a graph to model the network of activities and dependencies in software development is a sound basis for the realization of the conceptual framework. Furthermore, different types of reuse are supported. Reusing object descriptions or components is done by reusing the respective nodes and the information they contain. But also whole sub-paths or sub-graph structures can be reused which corresponds to reusing fragments of the development process.

This report focuses on several issues:

1. describing the model used to express the framework of constituents and relationships in terms of a "three-dimensional" graph structure, the so-called *3D-model* (chapt. 2),
2. establishing a first set of constituents and categories for these constituents (chapt. 3),
3. describing how the constituents are treated in known textbooks, and how they are integrated to software engineering methodologies (sec. 3.1.3,3.2.3, and 3.3.3),
4. giving two case studies that show which and how constituents have been addressed in recent software development projects at TNO (chapt. 4).

More interesting topics that are part of the 3D-model, but not treated in this report, are:

- categorizing or formalizing the relationships between the constituents,
- describing precisely the contents of a 3D-model node,
- formalizing the graph structure of the 3D-model,
- deriving a sound software engineering approach for a certain application area from that graph structure,
- many more open problems.

Chapter 2

The 3D-Model of Software Engineering

When software systems grow in size and complexity, their development extends beyond the question of algorithms and data structures. Suitable units of decomposition (or composition) have to be defined in order to divide the various development tasks into pieces of manageable size. This separation of concerns does not only apply to 1) the components of a system, but also to 2) its life cycle and to 3) the perspectives from which a system is described. These three categories, called **dimensions** in the following, form the basis to specify and categorize the various issues of software engineering separately from each other, especially separately from issues of other dimensions (cf. fig. 2.1). The units of decomposition of each dimension will be called **constituents**. Describing the three dimensions is possible and useful on different **levels of granularity** regarding the system. The different categories apply to the coarse grain units of a software system (the software architecture) as well as to the finer decomposition units as e.g., subsystems, modules (processing units), and even single classes. The term "dimension" was chosen to stress that a certain software development approach derived from the 3D-model could be modified within one dimension, without that such a change necessarily affects the development regarding one of the other dimensions. To prevent ambiguity, the term three-dimensional **grid** will be used to indicate that the arrangement of the constituents of a dimension doesn't imply any (temporal or other) order, but only serves to categorize and structure the development within the three-dimensional space.

Example: A software process includes the specification of an object communication diagram (views dimension) of the high level system components (components dimension) in the design phase (process dimension). Changing the set of components should have no influence on the other two dimensions, i.e, on the requirement of producing an object communication diagram in the design phase.

2.1 Overview of the three dimensions

The first two dimensions - views of the system and development process - have already been mentioned in the introduction. They are complemented by a third dimension, the components of a system. This distinction of a third dimension will be motivated later. All three dimension are introduced here and broken down into more detail in chapt. 3.

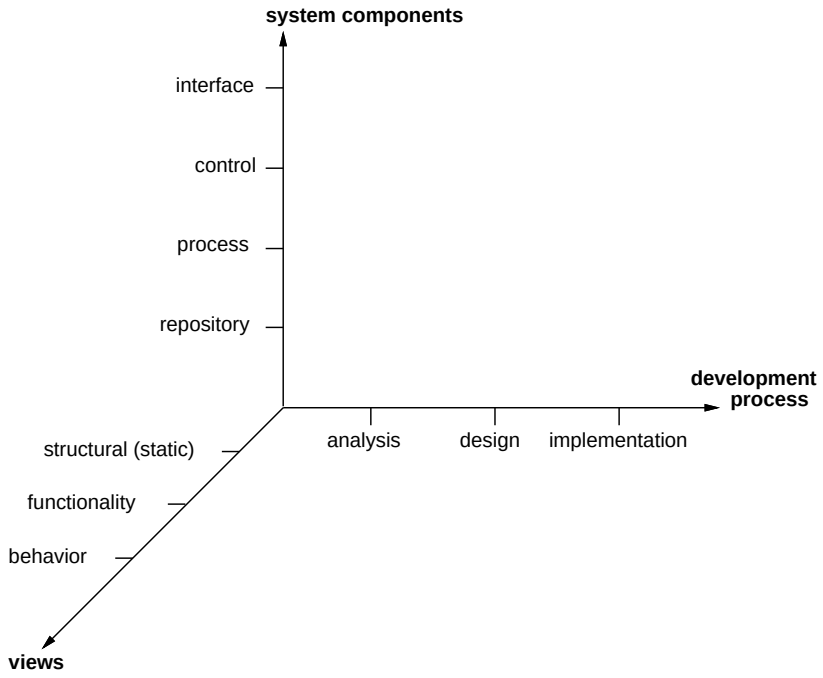

Figure 2.1: The dimensions of software development. The labels at the axes show basic constituents of the dimensions.

The first dimension is

1. **views of the system**

The three basic views (or also: perspectives) from which a system can be described are the *structural* view which describes the static structure of a system, i.e, its parts (objects) and their relationships , the *behavioral* view which describes the behavior of a system, and the *functional* view which describes the processing units of a system and the flow of the data along them.

The second dimension is

2. **development process**

The "classical" software development process distinguishes three consecutive development phases: *analysis*, *design*, and *implementation*. This distinction of three development phases can be adopted for the development process dimension at first sight, and is applied in most object-oriented software development methods, too. But it has to be taken into account, that the object-oriented paradigm allows a relatively independent development of components, and supports also evolutionary prototyping. This is an indication that the development process dimension has to take into account also forms of organization of the phases other than the classical one.

The independence of components and their development, mentioned above, leads to the identification of a third dimension, the dimension of

3. **system components**

This dimension goes beyond agreeing on a standard set of system components or their functionality, as proposed in certain *software frameworks*, as e.g., in [Nat91]. The system components dimension gives a generic classification to capture components that build up an arbitrary system. In the other dimensions, the constituents (of a dimension) represent fundamental principles of organization and classification which apply to all levels of specification. In the same way, constituents of the system components dimension should give a generic classification of the logical parts of a system, where "system" ranges from a whole software product to a single class.

The generic organizing principle for systems, used in this context, adapts the organization of a class of the object-oriented approach. It incorporates the component categories *interface* (comparable to the public features of a class), *repository* (the encapsulated data structures), *process* (the private part or body that realizes the functionality on top of the data structures), and *control* (which in class definitions can be expressed e.g. by pre- and postconditions of methods). The idea behind this categorization is that different parts, i.e., system components, of the system provide different sets of services, comparable to the MVC-Concept [KP88].

Are the more dimensions?

Potentially, software engineering encapsulates also **other principles** which could be used as **categorizing dimension**. The different application areas and software product families clearly have an influence on the appearance of the 3D-model graph and hence are candidate for the constituents of another dimension. The same holds for the set of notations with which the elements are described. But extending the 3D-model with an "applications domain" dimension and with a "notations" dimension would unnecessarily decrease its understandability. Therefore, these two specializations are rather regarded as applying the generic 3D-model for a specific development effort rather than as new "dimensions". By limiting the number of dimensions to three, the fundamental structuring mechanism is kept as simple and basic as possible. The choice of the three dimensions introduced above guarantees a well-structured software engineering approach.

2.2 One-dimensional integration

Each dimension deals with another aspect of the software system (perspective, time, components). To come to a complete picture of a dimension, its constituents have to be related to each other (cf. fig 2.2). A first step is to relate the constituents of one dimension to each other. This holds within each of the three dimensions, as the following three examples show:

Example:

1. *The structural view of the system is related to the behavioral view and to the functional view. State transition diagrams (STDs) can describe the life cycles of the objects, (complex) activities within the states of the STDs can be decomposed into processes. Processes involve flow of data, which are objects themselves, thereby closing the circle (cf. sec. 3.1.2).*
2. *In the process dimension, objects identified during analysis will be found back throughout the whole life cycle of the system. This kind of "traceability" is often a requirement demanded from the software engineering methodology. It indicates that constituents are related to each other also in the process dimension.*

3. The decomposition of a software system into a set of suitable subcomponents implies relationships between these subcomponents on that (lower) level of decomposition. The specification of subcomponents of a system and their collaboration is addressed as "system architecture" in the literature [STO89, Obj90, FkNO92, PW92, GS93] (cf. sec. 3.3.3).

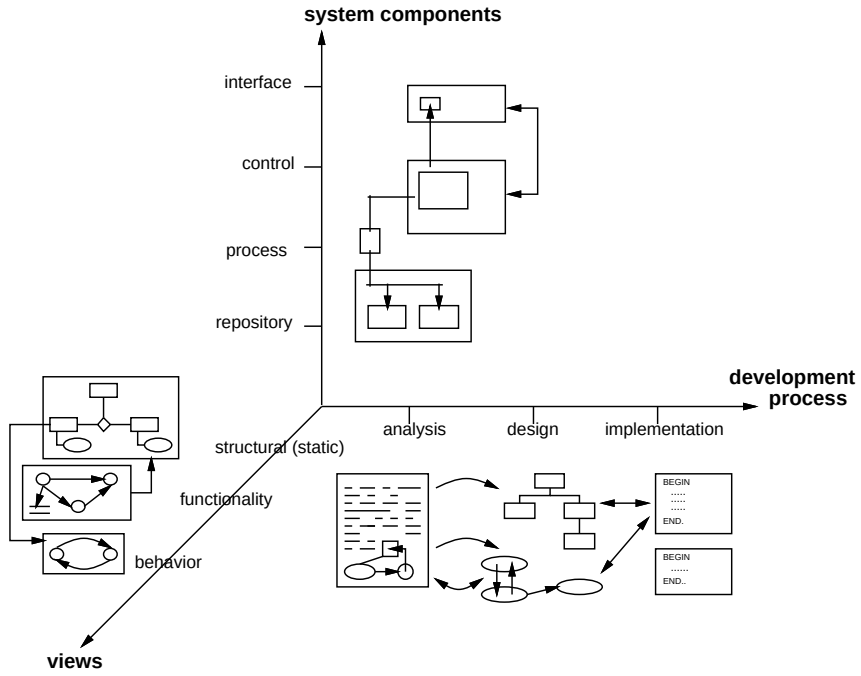

Figure 2.2: *One-dimensional integration*: Relationships between constituents of one dimension integrate them within that dimension.

The one-dimensional integration sketched in fig. 2.2 results in a coherent description of each dimension. This one-dimensional integration refers to the construction of (the static structure of) the generic 3D-model, i.e., the generic dependencies between constituents within each dimension. The construction of the generic 3D-model has no implications to the (dynamic) activities of a certain software development process modeled *using the generic 3D-model*. which integrates the different specifications that have to be worked out.

2.3 Three-dimensional integration

Extending integration to all dimensions captures the whole picture of software development. This means that each specification belonging to the software development combines one (or more) constituents of each of the three dimensions. In terms of the 3D-model, we obtain a sort of **three-dimensional grid** in which each cross point represents a certain combination of constituents (cf. fig. 2.3).

This three-dimensional grid is the intuitive representation of a **graph** where each node describes a certain view of a certain system component at a certain moment the development process (the combination of constituents of all three dimension). The nodes will also be called

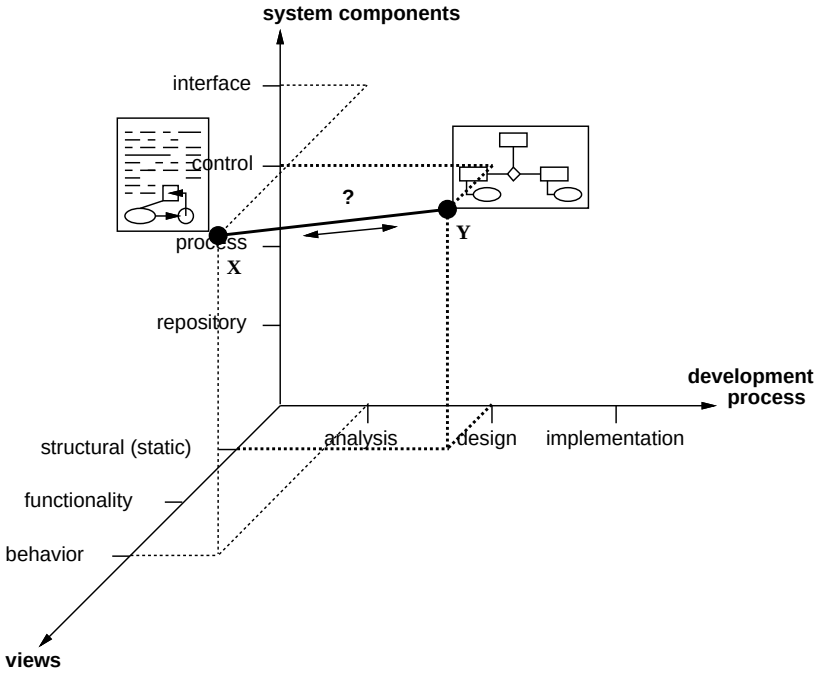

Figure 2.3: An example for the combination of constituents from each of the three dimensions making up one software engineering element (= node of the 3D-grid). Here, node X represents the state model of the (user) interface component in the analysis phase, node Y the object model of the system control component in the design phase. The question mark labeling the connection between the two nodes indicates that a relationship between those two nodes might exist, but its semantic is not specified in this figure.

elements in the following. A software engineering "element" is a triplet of constituents of each of the dimensions. The edges between the nodes represent the dependencies, implications, and other types of relationships between the 3D-model graph nodes.

2.4 The "ingredients" of a software engineering element

A 3D-model node is more than just a location in the 3D-model graph. The software engineering element which is represented by such a node is a "container" for relevant engineering information, development constraints, knowledge, etc.. A first tentative to depict the "ingredients" of a software engineering element in the scope of the 3D-model results in the following list:

- the **combination of constituents** of each dimension of the 3D-model that places the respective node at the appropriate place in the software development
+ a **document**, or **deliverable** that might capture a **specification**
+ its (possible) specification **formalism**(s)
+ an **activity** to produce the specification
+ a **tool** to support the specification process
+ a **commitment** (personnel, resources, i.e, management aspects)
+ a list of **versions** of the specification as result of iterations in the specification process
+ a set of outgoing and incoming **dependency edges**, that relate the respective node to other nodes, and hence the underlying element to other elements or decompositions of that element.

The different "ingredients" of an element of the 3D-model don't fall all in the same category. While the first (combination of constituents) and the last (dependency edges) determine the structure or the 3D-model graph, the other ones can be seen as attributes of the nodes.

2.5 Decomposition: A fourth dimension?

The 3D-model describes a system at a certain level of granularity. As mentioned before, in software development the notion of "system" might extend from the coarse-grain system architecture of a whole software system to the detailed specification of certain components to the specification of single classes. This has as consequence that each of the elements of a 3D-model, i.e., each of the nodes of the graph, may be decomposed and represented by its own "3D-model" (cf. fig. 2.4).

Decomposition can occur concerning all the three views of the system, and repeats recursively to an appropriate level.

Example:

1. *Functional decomposition is known from SADT [YC79, You89]. It means decomposing the processes that represent the system functionality.*
2. *Structural decomposition partitions a system into its subcomponents which consequently are treated as "systems" of their own.*
3. *An example of behavioral decomposition is the transition from "black-box" state transition diagrams which describe the behavior of a system as perceived from the outside (e.g., by users) to the behavioral description by "white-box" state transition diagrams, where the control flow between the different parts of the system and their individual actions are distinguished.*

Decomposition can follow different decomposition principles. A node can be specialized into subtypes of that node, applying inheritance, or it can be split up into subcomponents by aggregation. In both cases, decomposition means to go from an abstract level of specification of a node to a more detailed level, but the semantics of the decomposition relation is different for both types of decomposition.

Whether a certain node has to be decomposed, i.e., has to be specified in more detail on a lower level, or not, depends on a variety of factors, such as e.g.,

- to what degree that element is already available (e.g., as commercial product)

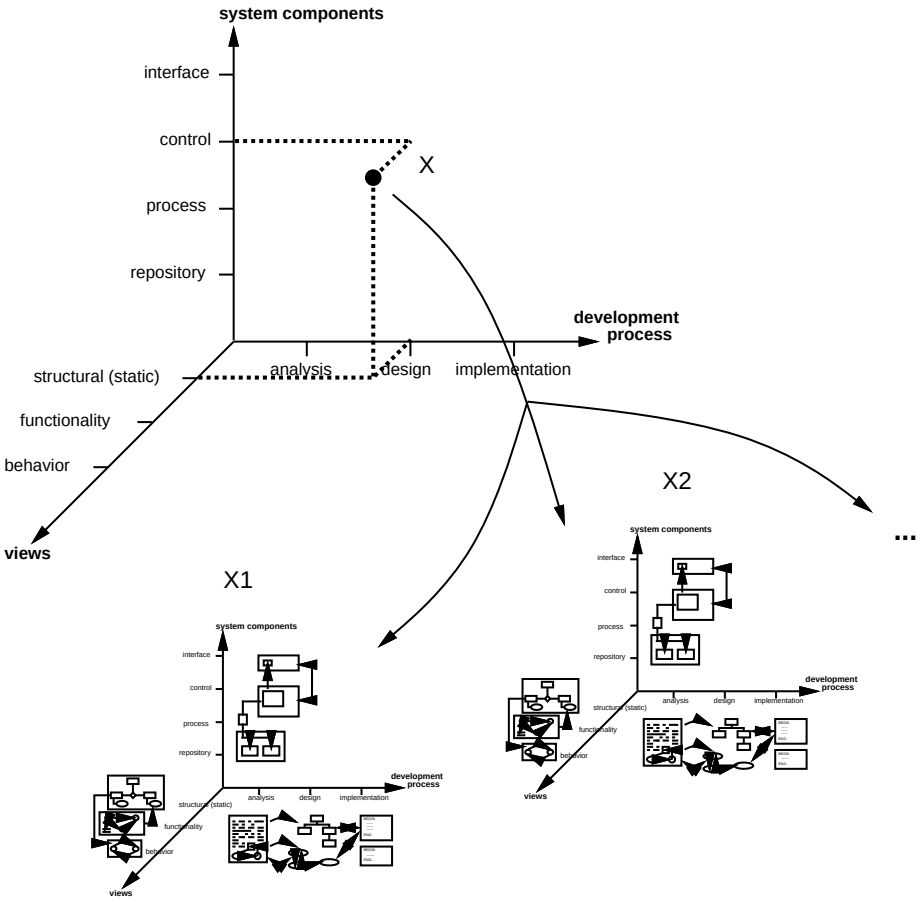

Figure 2.4: The description of a system ("X") in terms of the 3D-model at a certain level of granularity might be decomposed into more fine-grain 3D-model descriptions of its sub-subsystems ("X1", "X2", ...).

- to what degree that element is understood, complex, or trivial
- what the scope of the element is (e.g., give a first overview or specify in detail all aspects of it)

The granularity levels, i.e., the step-size of the decomposition, are also not generally predefined, but may be variable.

Example: A certain part of the software system might be described only superficially at the beginning of the development. In that case, the related node of the 3D-model, i.e., the specification connected to that node, would not be decomposed. Instead, a related node, representing the same part at a later stage of development, would be decomposed.

2.6 Applying the 3D-model

Up to now, the 3D-model only represents an *abstraction* of software development. But as mentioned in the introduction, the goal is to employ a model such as the 3D-model to derive a **practical framework** for software development which defines the necessary activities and guides the developer through the whole abstraction cycle.

Modeling software development by a graph allows parallel, non-sequential, and non-deterministic development activities. The graph notion itself, doesn't imply a specific software process, i.e., no total ordering of the activities (the production of documents) to obtain the system, but leaves it open to use the edges of the graph to define a development process. Instead of life cycle, the term **abstraction cycle** seems more appropriate to describe integration in the development process dimension. This indicates that the unities of the process dimension "axis" are levels of abstraction from the problem domain towards a software product rather than chronologically ordered.

To derive a framework for a *specific* development situation from the generic 3D-model, three steps have to be performed:

1. Potentially, the 3D-model grid implies a large number of possible elements, more precisely, the power set of possible combinations $\{SYSTEM\_COMPONENTS \times VIEWS \times DEVELOPMENT\_PROCESS\}$. So, the first step is to **chose which of the elements**, i.e., nodes of the 3D-model grid, **are relevant for the development of a specific software product**, or better, for a whole class of software products (e.g., for the development of GIS's).

   Example: For the specification of a database, only the static structure of the data might be of interest in the analysis phase. In that case, the node with "functionality", or "behavior" in their views dimension would not have to be filled in (cf. sec. 4.1).

   Identifying the relevant elements means to select those specifications that are meaningful in the context of a certain application (area).

2. The next step is to **fill in the "attributes" of the 3D-model nodes**, selected in the first step. Concerning the software engineering elements, this means to determine their "ingredients", i.e., to give activities, notations, commitments, etc. to be used when treating that element.

3. A **software process** has to be established. This means to formalize the course of activities encapsulated in the software engineering elements, i.e., to produce all required specifications. The course of activities is described and constrained by the various types of relationships that can exist between the elements of software engineering, i.e., between the nodes of the 3D-model graph, and by what is defined in the elements. As mentioned above, the relationships between the elements can define different types of dependencies between the various documents or their production. These dependencies imply a partial ordering of the activities, indicated by the direction of the edges of the related 3D-model graph.

   Example: A dependency edge from node `A` *and node* `B` *could be labeled "is_implemented_by". That would imply that the specification represented by node* `A` *will be implemented by module represented by node* `B`.

Other types of relationships between elements might not imply a fixed ordering of the activities to produce the underlying specifications.

Example: Decomposing different system components of a system can mostly be performed independently.

This report doesn't focus on the possible types of relationships. Attempts to categorize relationship types between software development activities can be found e.g. in [SIWyS93].

The software process can be derived individually from the (non-coherent) paths of the 3D-model graph, because they constrain only partially the order of the related activities. By covering all relevant nodes in the order constraint by the relationships among them, a full software process is described.

Chapter 3

The Dimensions

In this chapter, each of the three dimensions introduced in the previous chapter will be described in more detail. As no formal language will be used to specify formally the 3D-model graph in this report, the descriptions will more or less represent an "extended tour through the dimensions". The concepts, classifications, and integration principles described represent **generalizations from established methodologies and evaluated projects** and not formal definitions. Before the 3D-model can be specified formally, it has to be understood intuitively. Therefore, apart from presenting the dimensions of the 3D-model, another important purpose of this first description of the dimensions is to gain an overview of the requirements for a suitable formalism to describe the 3D-model graph.

As stated in the introduction, existing software engineering methodologies are often biased by their authors, i.e., by their working context. This doesn't mean that they are not suitable in any other context, but their usage is often limited by implicit integration and valuation of aspects that then are combined in one specification document. The result is that different methods differ in their focus on different aspects and principles of integration.

Example: [SM92] describes very detailed and formally the transformation from analysis to design, while [Boo91] gives only heuristics for that transformation. Concerning the integration of constituents in the development process dimension). [RBP+91] gives guidelines for the integration of system components ("prototypical architectures" - system components dimension), while [SM92] describes extensively integration of views (views dimension).

A **higher level of abstraction** reveals the basic, generic constituents of each dimension of the 3D-model which later combine to the elements of software engineering. The constituents will be described in the following, illustrated by examples from known textbook methodologies.

Example: In the views dimension, a list of identified entities, an object model, and a class hierarchy are concepts to describe a system ranging from the same point of view, i.e., the structural view of the system, but at different points in the abstraction cycle (from the first requirements specification to the implementation). Hence "structural view" of a system is present along the whole development process dimension (cf. fig. 3.1).

This holds analogously for the system components dimension. The description of static structure is important for every component, for the repository (the structure of the persistent data), as well as for the interface (e.g., the structure of the user interface windows), etc. . "Structural view" is also present in the whole system component dimension.

Static structure is consequently an "invariant" in the development process dimension as well as in the system components dimension. This invariance is a hint that the "static structure" is a candidate for a constituent of the views dimension.

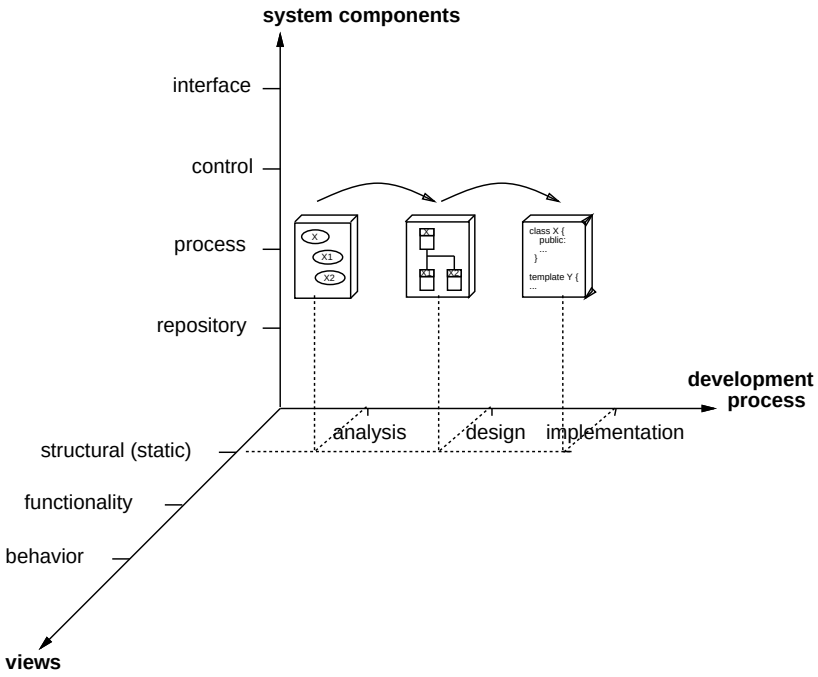

Figure 3.1: The necessity of static structure description is independent from the system components and development process dimensions.

While it does not seem difficult to recognize the static structure as constituent of the views dimension, other constituents are not so easy to identify. Even with the static structure, the case is not unambiguous. Looking at different methodologies, the specification of the static structure includes different aspects, the border to other constituents of the same dimension is sometimes blurred.

Example: [Boo91] distinguishes between class structure and object structure. While class diagrams capture the static design of a system, object diagrams are used to represent structure snapshots in time as results of operations. Other methods, like [RBP+91], don't make that distinction.

The following three sections try to identify the basic constituents of the three dimensions of the 3D-model. OMT [RBP+91] is used to illustrate graphically the concepts, and how they are related to each other.

3.1 The Views Dimension

The views dimension deals with the different perspectives from which a software system can be described, mainly:

- **static structure**,

- **functionality** (processes and data flow), and
- (time-dependent) **behavior** of the system and objects in it.

The description of each of these views can further be split up into several different **subviews** to emphasize certain aspects of modeling. None of the views is regarded as more important as the others a priori.

In the next section, the concepts of structural, behavioral, and functional modeling will be introduced.

3.1.1 Constituents of the Views Dimension

The Structural Perspective

Figure 3.2 gives an **overview of concepts** that can be found in structural system descriptions. The concepts are not always clearly distinguishable, this depends also form the language chosen for specification.

Example: Informal text, which is often chosen for describing a system (cf. sec. 4.2), and program code are two examples for languages where the concepts of structural description are not (always) so clearly distinguishable as, e.g., when a language as in OMT [RBP+ 91] is used.

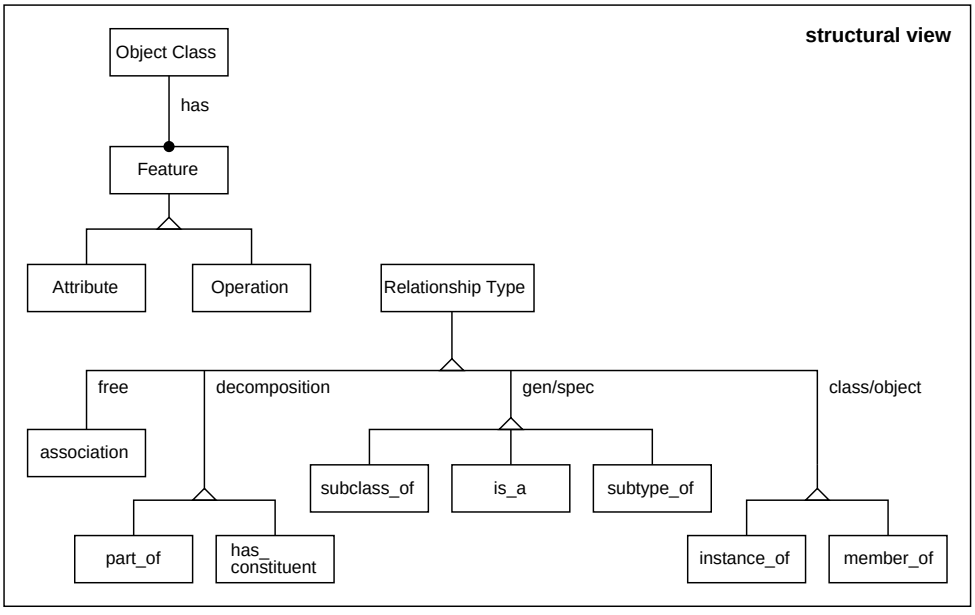

Figure 3.2: Concepts used for describing the static structure of a system

Contents of the static structure is the description of the *objects classes* and *objects* of a system (or of a part of a system), together with the various *relationships* among them. Object classes are characterized by their *features*, i.e., by their *instance variables* (*attributes*), and by the *operations* (*methods*) they provide. Several types of relationships among objects and object classes are possible. The list of concepts and relationships of the structural view is

mainly inspired by the object-oriented paradigm, because this paradigm is an excellent means to structure and modularize a system in a natural way.

The characteristics and purposes of the shown relationship types are:

free associations: "Free" means that the semantics of this kind of relationship is arbitrary, and is specified explicitly (e.g., by labeling them). Free associations are symmetrical and can include one or more objects of one or more object classes.

part_of: to model decomposition among objects.

has_constituent: to model aggregation among objects. In contrast to the part_of relationship where component objects are not necessarily depending from composed objects, the existence of aggregated objects depends from the existence of the constituting objects.

is_a: to model generalization/specialization structures among objects.

subtype_of: to model generalization/specialization among object classes.

subclass_of: to model inheritance of features on implementation level.

instance_of: to relate objects (instances) directly to object classes.

member_of: to relate objects (indirectly) to all super-classes of the class to which the object is related directly through an "instance_of" relationship; this kind of relationship is used in systems where objects can only be "instance_of" one object class.

The Functional Perspective

In object-oriented development methodologies and programming, the functionality of a system, i.e., its available and identifiable operations, is closely attached to the objects. Functionality is provided by the objects of the system, and therefore often specified together with the structure of the system. Nevertheless, a separate specification of functionality as an autonomous view of a system should be kept if necessary. Investigating the functionality of an application can be a useful and important part of the analysis [RG92, JCJO92].

Concepts of the functional view describe (for example) the units of processing (*processes*) and the intercommunication (*data* and *control flows*) between them [SM92]. It shows how output values (of a processing unit) are generated from (external) input values, and how data flows, intermediate processes, and *data stores* (as persistent data objects) are involved in the computation (cf. fig. 3.3).

Here, the same observation as for the static structure description holds: depending from the language chosen, the concepts cannot always be distinguished so clearly.

The Behavioral Perspective

In "classical" behavior modeling, the behavior of a system can be specified from two viewpoints: the behavior of a system as it is perceivable from the outside of the system, e.g., by a user (black box perspective), and the flow of control, the interactions, and the sequence of activities within a system (white box perspective). As a matter of facts, these two perspectives are not fundamentally different, but rather represent only different levels of granularity in the

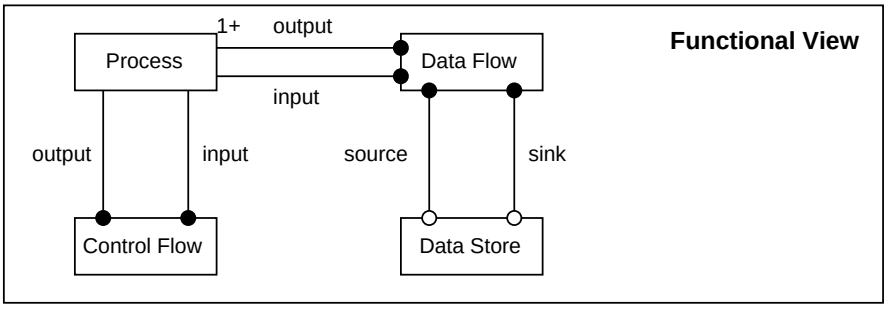

Figure 3.3: Concepts used to describe the functional perspective of a system

description of behavior. It should be remembered that the term "system" applies to different levels of granularity of the software system. So, the black-box perspective of the behavior of a lower-level "system", which is a subsystem of a higher-level system, is an essential part of the white-box perspective of the behavior of that higher-level system. The two specifications are mutually related because they express how the behavior of system and subsystem relate to each other.

Concepts to specify behavior of a system or its interacting parts are (for example) the *states* in which a system can be, the *transitions* between states, *conditions* for transitions (e.g., events as conditions for transitions triggered from outside the system), and *actions* (i.e., collection of operations) that are connected either to the states or to transitions. *Timing* and *synchronization aspects* can extend the behavioral view for special purposes (cf. fig. 3.4).

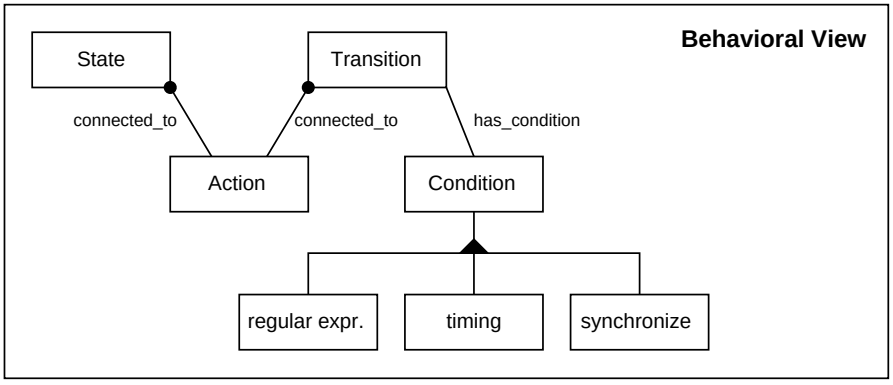

Figure 3.4: Concepts of behavioral perspective description

Here again, distinction of concepts might not always be so easy or possible, depending from the specification formalism.

3.1.2 Integration of the Views Dimension Constituents

The complementing views of a system described above can be integrated to provide an "overall" perspective of the system to be specified. This integration is necessary to ensure **con-**

sistency between structure, functionality, and behavior of the system description (cf. fig. 3.5).

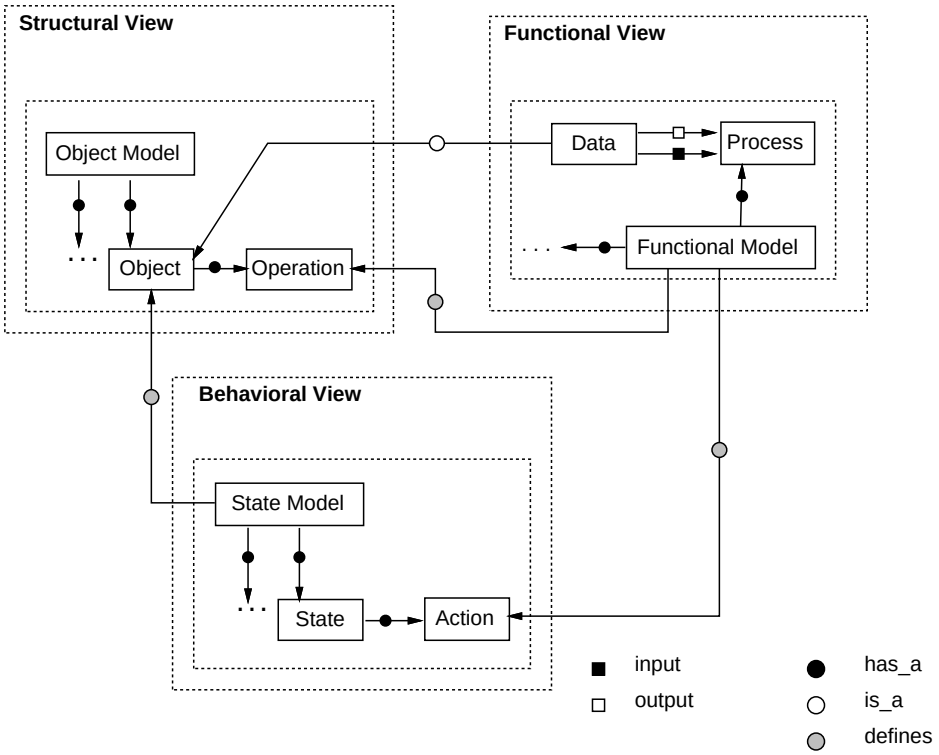

Figure 3.5: Integration of the different perspectives of the views dimension

Starting with the structural view, the behavior of objects (as instances of the object classes) with a non-trivial life cycle can be described with a4 behavioral view description. For every object class, the behavior of its objects is specified by, e.g., a state model. The actions which occur within the states of such a state model or during the transitions can be so complex that they might have to be specified in more detail with a functional view description. The functional model can be used also to describe the behavior of operations of the object classes defined in the structural view.

3.1.3 Examples

This section contains a collection of examples of notations for the views dimension of a system. They represent **possible notations** to describe the different views of the view dimension. As the focus is on the describing possible notations for the views dimension, the notations mentioned are listed outside their context, i.e., they may be taken from different development phases or suitable for different levels of granularity.

Structural perspective:

OMT [RBP+91]: Object Model (Diagram), Data Dictionary

Booch [Boo91]: Class Diagram, Object Diagram, Module Diagram, Process Diagram

Shlaer/Mellor [SM88, SM92]: Information Structure Diagram, Object and Attribute Descriptions, Relationship Descriptions, Class Diagram, Dependency Diagram, Domain Chart

Use Cases [JCJO92]: Structural Model, Entity Object Model, Subsystem

Fusion [CAB+94]: Object Model, System Object Model, Object Interaction Graphs, Visibility Graphs, Class Descriptions, Inheritance Graphs

Behavioral perspective:

OMT [RBP+91]: State Diagram, Event Flow Diagram

Booch [Boo91]: State Transition Diagram, Timing Diagram

Shlaer/Mellor [SM88, SM92]: State Model, Event List, Object Communication Model

Use Cases [JCJO92]: Interaction Diagram

Fusion [CAB+94]: Life Cycle Model, Scenarios

Functional perspective:

OMT [RBP+91]: Data Flow Diagram

Booch [Boo91]: -

Shlaer/Mellor [SM88, SM92]: Object Access Model, Action Data Flow Diagram, State Process Table, Process Description

Use Cases [JCJO92]: Use cases

Fusion [CAB+94]: Operational Model

3.2 The Process Dimension

There is no cookbook process to follow when developing software. Especially with object-oriented methods, where systems can evolve incrementally, idioms such as "waterfall model", "bottom-up" or "top-down", can't be applied strictly and exclusively to the software engineering process. It is only from the management perspective that software system, which are designed and constructed in a cyclic way, have to be phased sequentially in order to support time frame-driven project management [Ger93].

3.2.1 Constituents of the Development Process Dimension

In the literature, system development is subdivided in different sequences of phases, depending from the proposing authors. Some examples of classifications are:

- OMT [RBP+91]: analysis, system design, object design, implementation
- Booch [Boo91]: analysis, design, evolution, modification
- OOAD [SM92]: analysis, design, implementation
- Use-case driven design [JCJO92]: analysis, construction, testing
- Fusion [CAB+94]: analysis, design, implementation

All approaches distinguish three main phases from the problem statement to the "finalization" of the system. OMT, the first of these approaches, also distinguishes between the overall design of the system and the detailed design, reflecting the importance of identifying a proper system architecture as a substantial part of the design process.

It is notable that many methodologies, while generally recognizing the need or presence of some kind of testing and/or maintenance phase, include such a phase only very informally in their proposed categorizations of software development phases.

The names of the **constituents of the development process dimension** (i.e., the main development phases) used here in this section differ a little from the traditional phasing, i.e., analysis, design, implementation, also used in sec. 2.1. The names used here reflect the potential incorporation of evolutionary prototyping in the development, therefore *analysis*, *evolution* (= design), and *finalization* (= implementation) have been chosen. The phases are accompanied by the types of specifications (indicated by a "→") that will result from, or be modified by, the activities belonging to the phases. In sec. 3.2.3, some examples of possible notations to be used in the different phases are given.

In order to keep the first approach simple, this abstraction cycle describes only the development of the software system until completion. Therefore, the description of phases ends with the delivery of a new release. Although *maintenance* or *extension* of the system are not included in this report, they are important phases of the software life cycle and have to be considered in the development process dimension as well.

Analysis

1. *Preparation*
   → software project management plan (SPMP)
   → quality assurance plan (SQAP)
   → global test plan
2. *Concepts & ideas*
   → basic concepts
3. *Requirements analysis (Primary analysis)*

   Sub-activities:

(a) general system constraints, development constraints

(b) identify (interacting) entities

(c) describe interactions

(d) identify processes, services with initiators, participants, pre-/postconditions
→ global system requirements specification (global SRS)

(e) extract object model from (interacting) entities (hierarchical and contractual OM)

(f) specify object dynamics

- life cycle
- functionality

→ system requirements specification (first SRS)

4. *Global system design*

Sub-activities:

(a) identify available system components and their interfaces

(b) associate results of SRS (objects, processes, services, actions (of state models) to available system components

(c) define the overall system architecture and place available components using referential components and architectures
→ global system design document (global SDD))

(d) build first version prototype
→ (first prototype)

(e) identify "delta-design", i.e.:

- components that have been identified as required for the system, but which are not part of the referential architecture
- components that are part of the referential architecture but not identified as required for the system

Evolution

1. *Requirement analysis*

Sub-activities:

(a) modify SRS according to evaluation of prototype
→ system requirements specification (modified SRS)

2. *Object Design*

Sub-activities:

(a) design oo class hierarchy

(b) describe features of the classes
→ system design document (SDD))
→ detailed test plan

3. *Prototyping*
   → (modified) prototype

Finalization

1. *Finalization of Code*
   → alpha release
   → alpha test plan

2. *Alpha Testing*
   → beta release
   → beta test plan

3. *Beta Testing*
   → new release
   → maintenance plan

3.2.2 Integration of the Development Process Constituents

When defining the steps and phases of system development, it has to be taken into account that a clear **distinction of phases** that results in a sequential ordering of activities and specifications is not always possible or desirable, because:

- Object-oriented development and programming environments support an independent development of the individual components of a system. As consequence, different parts of the system may be in principle in different stages of completion.
- Especially in the development of experimental software, as is the case at TNO, system requirements and design evolve during the course of a project. Knowledge of the system grows progressively as work progresses. Prototyping can facilitate the evaluation of intermediate stages of the system, as it allows the developer to focus on certain properties of the system, and to allow them to experiment with a number of different design options.

Consequence could be an **incremental** and **iterative development** of the system, where switching between activities is normal. Nevertheless, this working style has to be underlaid by a sound and ordered process in order to prevent chaos. It is important to draw a line between development phases, and to design development cycles that fit in the frame of sequential software engineering phases. Thereby, stages should be established which do not require changing the results of earlier documents when later documents are changed. In other words: while iteration in the productions of the documents of one phase is promoted, iteration between the (main) phases should be avoided. Especially, it is important to know at least the full range of requirements, and to have a picture of the global system design, i.e., the system architecture, before starting any detailed design activities. This level of maturity is required particularly when certain system components are already in an elaborated stage (due to reuse or commercial availability), in order to avoid redundant development activities.

In the proposed software process, some of the activities and documents play a special, exposed role. When a **prototyping approach** is chosen, they serve as "links" between the

phases, i.e., they appear in more than one phase. This holds primarily for the prototype itself, of course. But also the requirements specification and parts of the system design are activities - and result in documents - that transgress the border between development phases.

In "traditional" software development life cycles, the documents of the current phase are obtained by transforming/translating completed specifications of a former phase into the formalisms of the current phase (cf. fig. 3.6a). With prototyping, certain specifications are directly carried into the next phase and used there as basis for further development (cf. fig. 3.6b). This imposes special requirements concerning quality, traceability, and evolvability to these specifications. Although these special requirements for the specifications look like implying a substantial increase of document production (i.e., work to do), this isn't the case, because documents are taken directly to the next phase, where they are extended or elaborated in more detail - they don't have to be worked out from scratch.

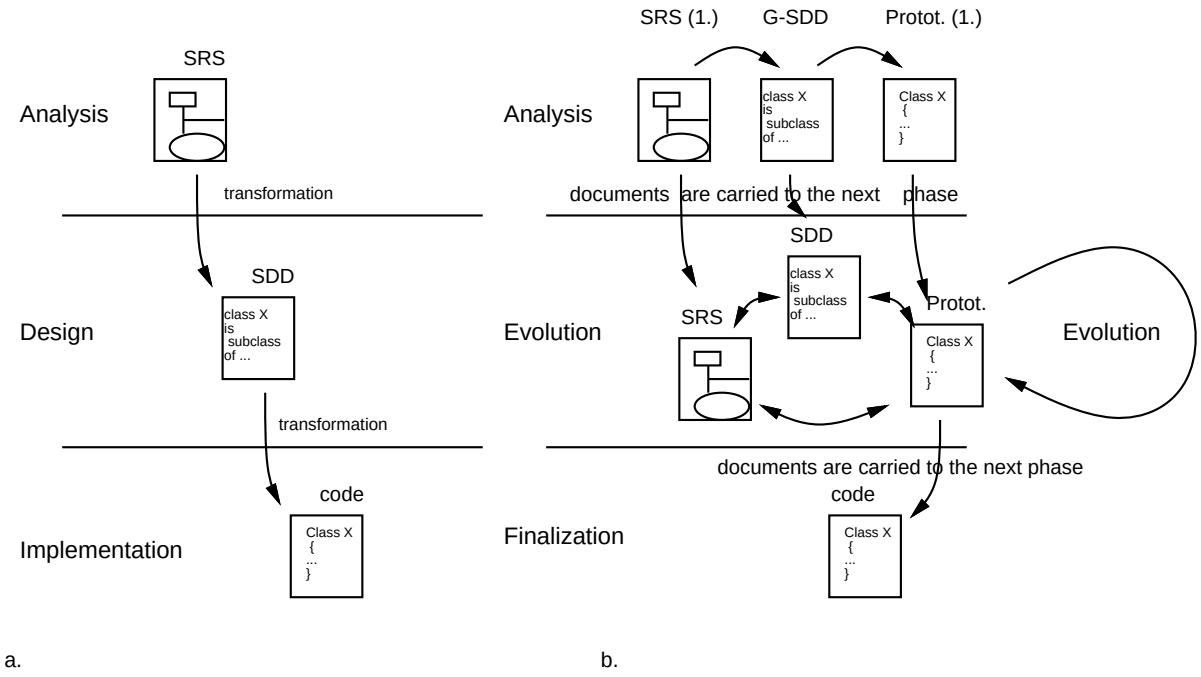

Figure 3.6: document progression in a) "traditional" software engineering, b) prototyping

An alternative integration approach without prototyping:

From the process outlined above and in fig. 3.6b, a software process without prototyping can be obtained, too, by "filtering" out a different subset of the constituents of the development process dimension. In order to achieve this, the activities presented in sec. 3.2.1 have to be integrated in a different way (cf. fig. 3.7). The system requirements specification (SRS) is not carried into the evolution phase, but completed in the analysis phase. The evolution phase is not cyclic, but rather "linear", i.e., system design is the main specification document of the middle phase. This middle phase, the design terminates with the completion of the SDD. Of course, there is no evolving prototype, so "prototyping" as coding activity is dropped, and

coding can not be found before the finalization phase which becomes an "implementation" phase. By then, system requirements and system design are stable.

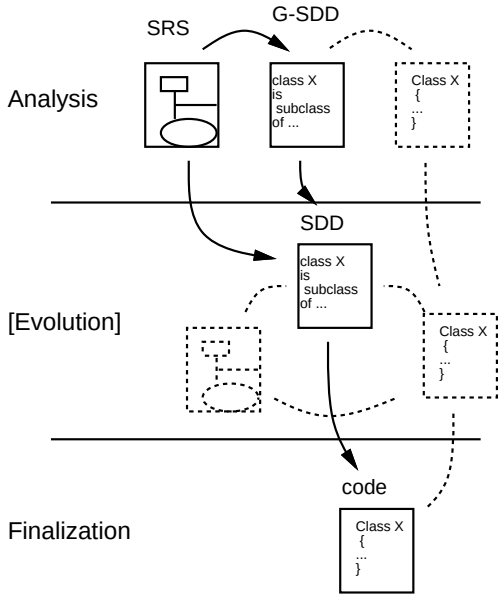

Figure 3.7: Integration without prototyping implies a different integration of development activities, but can be "derived" from the evolutionary development process

3.2.3 Examples

The distinction of phases and related document productions suggested in section 3.2.1 is realized in the **PROBE-project** [Ger93]. This process seems to match well the requirements of the development of experimental software. Prototyping takes a central role in the design of the system. Within the design phase, changes of the requirements specification and the system design are possible as consequence of the evaluation of the current prototype. Here, the prototype is not used as "toy" to visualize certain properties of the system (e.g., to illustrate the user interface), but evolves incrementally until it captures the whole range of system requirements and functionality. Thus, the "implementation" phase is reduced to the "cleaning-up" and optimization of the code.

The *O-O-O methodology* [HS93] replaces the traditional abstraction of the "waterfall model" by a *fountain model* . This model assumes a "pure" object-oriented development life cycle, i.e., a development process which uses the object-oriented paradigm in all its phases. The graphic image of a "fountain", where water rises in the middle, but falls back and is re-entrained at intermediate levels, is used to capture the iterative character of all activities which lead to the development of a software system. The O-O-O methodology distinguishes between seven basic activities which essentially cover the life cycle as far as the final production of the software product and release to the customer. Because the object-oriented paradigm is applied, the various parts of the system can be in different stages of development.

Some notations for the various development process constituents proposed in section 3.2.1 are:

Requirement analysis:

(b) "shopping list" with classification of the entities [RBP+91]

(c) Scenarios, Scripts [RG92], Use Cases [JCJO92]

(d) Glossaries [RG92])

(e) Object Model [RBP+91], Information Structure Diagram [SM92]

(f)
- State Model, Object Communication Model, Event List [SM92]
- Action Data Flow Diagram, Object Access Model [SM92]

Global System Design:

(b) architectural frameworks [Nat91, Obj90]

(c) referential system architectures [RBP+91, GS93]

Object Design:

(a) Class Diagram, Dependency Diagram [SM92]

(b) Object Interaction Graph, Visibility Graph, Class Description, Inheritance Graph [CAB+94]

3.3 The System Components Dimension

A key property of a (software) system is the quality of its **internal architecture**. A good architecture makes the system easy to understand, change, test and maintain. The architecture of the system has influence on the development process, because the selection and structure of the components of the system predetermine which components can be reused or adapted (from previous projects), which components have to be developed from scratch, and how these components have to be integrated into a homogeneous system.

3.3.1 Constituents of the System Components Dimension

As the 3D-model is suited to describe systems on different level of granularity, this has to be taken into account when identifying the generic components. The classification of the generic constituents of the system components dimension has to be valid for the description of the system on different levels of granularity.

The categorization used here follows the *object-oriented paradigm*, adopting the class concept to describe the constituents of a system in general. This implies that every system, subsystem, module, etc., can also be seen as object and part of a superior system (specification). As consequence, the specification of each "system" can be decomposed for all its parts, so that every part might be described by its own "3D-model" again (cf. sec. 2.5).

Interface is a constituent of the system components dimension for the purpose of communicating with other systems, or the outside world, corresponding to the public features

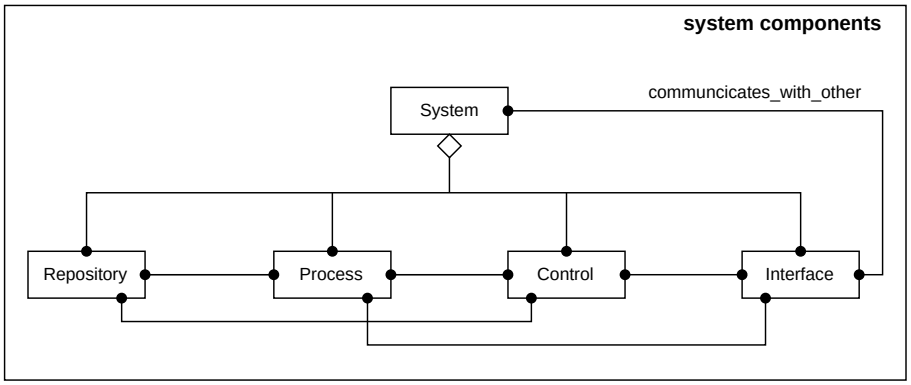

Figure 3.8: The constituents of the system components dimension. The semantics of the association links between the constituents is not specified in detail. At this stage, they just indicate communication between the components.

of a class. In the scope of the "system" abstraction, user interface and interfaces to other (sub)systems fall into the same system component category, because purpose of both is to delimit the communication with, and access to, the rest of the system. As consequence, a system description can have several, very different interface components.

Example: A software system can have a user interface which can be quite extensive to develop, with an own development life cycle - a complete software subsystem of its own. Such a user interface itself has two "interfaces": the one to the users (the windows, buttons, etc.), and an interface to the rest of the software system, which triggers the system's reaction.

Repository is the equivalent to the data structure part of a class. In the context of this classification, "repository" can comprise whole (sub-)systems like persistent data stores, e.g., database components (of software systems).

A *process* component realizes the functionality on top of the data structures, comparable to the body of a class. In many systems, this part is identical with what is called the *application specific part.*

Control is often encapsulated in the methods in class descriptions, but is realized by extra components (modules, etc.) in systems of a larger scale than a class.

Each of the system components types described above has its own purpose and provides one specific set of services within a system.

Example: The MVC-concept [KP88] is another example for a classification framework for system components according to their specific task within the system. MVC distinguishes the component types "model" (comparable to "repository" + "process" in the system components dimension of the 3D-model), "view" (comparable to "interface"), and "control" (comparable to "control").

The semantics of the associations in fig. 3.8, representing various types of relationships between the system components, as well as the actual number of system components within a system is up to now not further specified. This problem will be addressed further in sec. 3.3.2.

In the following table, the constituents of the system components dimension are matched to the specific terminology used on different granularity levels of granularity of system specification:

Concept:	software system:	component/library:	object class:
Interface	subsystem	API	public features
Control	e.g., ORB	-	e.g., pre/post conditions
Processing	application part	functions	body
Repository	database	repository	data structures, instance variables

3.3.2 Integration of the Constituents of the System Components Dimension

Integrating the different components of a (software) system means to specify the **(software) system architecture**. To formalize which components build up a system, and how these components work together, gains more and more importance with increasing size and complexity of the systems. This is reflected by the large amount of activities in this area of computer science. These activities have produced a number of approaches to describe (generic) system architectures, but most of them concentrate on the components, rather than on (the much more difficult) question of collaboration and integration.

[GS93] use existing program organizations to extract abstractions, the so-called "prototypical" architectures (cf. sec. 3.3.3).

[Nat91] gives a framework for the functionality of a software engineering environment (a special software system therefore).

[STO89] describe systems in terms of a "corporation model" to illuminate aspects of collaboration between software components.

Another approach to the integration of components via a formalized framework is the **Common Object Request Broker Architecture** (CORBA) [Obj91], a framework for controlling the collaboration and interaction of objects within an object-oriented software system. As CORBA is also based on the object-oriented paradigm, it seems to be a good starting point to specify formally the integration of components in the system components dimension.

3.3.3 Examples

System components and Use Cases:

A comparable, granularity level-independent classification of a system can be found in Jacobson's **Use Case** methodology [JCJO92]. In that methodology, the functionality of a system is described by means of "use cases", which model actors and their sequences of interactions with a potential system. Such use cases serve as models for the structure and the use of a (software) system or parts of it. The use cases are then analyzed and structured using different types of objects which represent the different component types of the system. Jacobson distinguishes between:

- *entity objects* to handle the information that a system will use over a longer time,
- *interface objects* to handle communication with the environment of the system,
- *control objects* which are needed in complex cases when control of the system can't be naturally placed into the other categories.

This categorization is very similar to the one proposed in sec. 3.3.1 (cf. fig. 3.9).

3D-model approach	Use case driven approach
interface component	interface objects
repository component	entity objects
control component	control objects
processing component	(the body of entity and control objects)

Figure 3.9: Use Cases and system components

System components and the ECMA/NIST reference model

The so-called **Reference Model for Frameworks of Software Engineering Environments** [Nat91] serves as reference for building standard components, together with the services they should provide. The ECMA/NIST reference model specifically addresses the development of software engineering environment, which are, in fact, a special group of software systems. In fig. 3.10, the standard components proposed in the ECMA/NIST reference model are related to the system components dimension of the 3D-model.

Object Management: logical modeling and physical storage of objects; includes definition, storage, maintenance, access of data entities or objects and the relationships between them	repository component
User Interface: provides presentation features between the system and the user of the system	interface component
Tools: facilities that are useful in many application domains; support for various "composite functional elements"	processing component, interface component (e.g., if it is a UI toolkit)
Communication: exchange of information between components & subsystem by mechanisms of messages, process invocation, and remote procedure call, or data sharing	control component, interface component (e.g., if the communication is with the environment of the system)
Process Management: unambiguous definition and performance of activities across the system	processing component, control component

Figure 3.10: ECMA/NIST and system components

The fact that the interface component can be found in three of the five ECMA/NIST service groups indicates that the categorization of system components as proposed in the 3D-model might not always be so strictly applicable to existing software components.

Integration - Prototypical Architectures

Examples of architectural integration can be found in [RBP+91] and in [GS93], where so-called **Prototypical Architectures** (**architectural idioms**) are proposed. They represent abstractions from heuristics, and capture some basic structural models and computational principles. Architectural idioms in [RBP+91] include:

1. batch transformation
2. pipes and filters
3. call-and-return
4. event systems
5. blackboard pattern
6. interactive interface

Chapter 4

A First Case Study

The two case studies presented in the following describe the development of systems developed at TNO in the recent past. The purpose of these case studies is to identify in existing projects the abstract elements of software engineering as introduced in the previous chapters. Furthermore, first examples of (intuitive) 3D-model graph representation are given.

4.1 Case 1: DEF

Goal of the **DEF**[1] project was the proposal of a standard exchange format for geo-scientific 3D subsurface model data. The standard exchange format had to be applicable for both tapes and computer networks, and is supposed to use as basis for software development in geo-scientific projects. Basis for the evaluation of the DEF project was TNO-report OS 92-83A [PRK92].

In the following two cases, reference to the nodes of the 3D-model are indicated by a `[...]`-notation, specifying the "coordinates" in terms of the three 3D-model dimensions. `[SC...]` stands for the system components dimension, `[V...]` for the views dimension, and `[DP...]` for the development process dimension. If the coordinates can't be given precisely, intuitive terms are given, as, e.g., in `[analysis]` which expresses that the related software engineering element is located in the analysis section of the development process dimension (more precisely: on a plane characterized by the value "analysis" for the development process dimension coordinate).

Contents:

[1] DEF stands for **D**ata **E**xchange **F**ormat

Overview

Concerning the **development process dimension** [DP], only the first two phases (analysis, design) are present, because the definition of the standard data format didn't involve any implementation/coding. DEF can be seen as software system-independent analysis and design activity to specify the data structure part for following projects (cf. sec. 4.2).

Only these two main phases are distinguished: *conceptual modeling* [analysis] and a translation to *relational/hierarchical structures* [design].

Development Phases:

[DP-1]	[analysis]	Standard logical model definition (= conceptual modeling)
[DP-2]	[design]	(Translation to) relational/hierarchical structures

The (standard) logical model relates only to one particular **system component** [SC] of the 3D-model: the *repository* [SC-1]. This repository has some subcomponents though. In the following table, the second column indicates again the component category concerning the system components dimension. All subcomponents of [SC-1] belong to the repository category again.

Subcomponents of [SC-1]*:*

[SC-1.1]	[repository]	Semantic Part of a 3D subsurface model
[SC-1.2]	[repository]	Structure for topology / geometry for irregular and regular (grid) representations
[SC-1.3]	[repository]	Extra info

DEF describes only one **view** [V] throughout its whole life cycle: the *structural view* [V-1].

Views:

[V-1]	[structural]	Data model

Altogether, the whole trajectory of this project can be described as a "flat" graph extending on the "structure plane" of the 3D-model grid, i.e., the plane that extends above the [V-1] coordinate of the view dimension (cf. fig. 4.1). The 3D-model graph of DEF has no view coordinate other than "structure" [V-1]), i.e., consists only of structural description.

Analysis

The conceptual modeling phase can be subdivided into four sub-phases: *identification of the objects* [DP-1.1], *NIAM schemes* [DP-1.2], *description of relationships and constraints* [DP-1.3], and *determining an object type catalogue* [DP-1.4]. To establish and stabilize the results of [DP-1.1] to [DP-1.3], **iteration** was used, but after stepping further to sub-phase [DP-1.4], no iteration back to the previous phase was done (cf. fig. 4.1). This particularity of the development process couldn't be retrieved from the documents. The course of the sub-phases was inquired by interviewing members of the project.

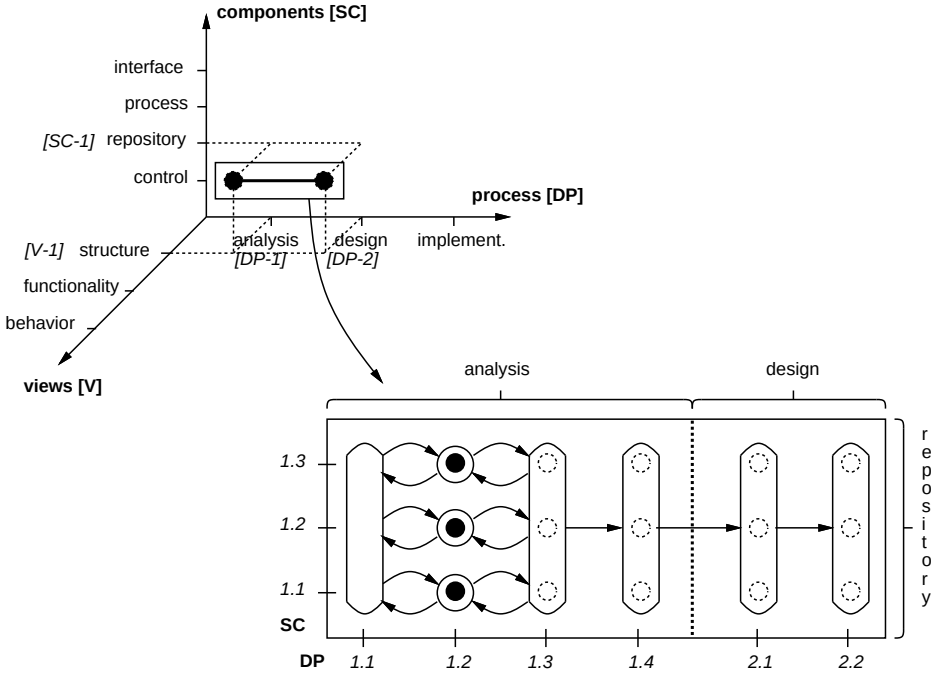

Figure 4.1: The plane representing the specification trajectory of DEF.

Sub-phases of the analysis `[DP-1]`*:*	
`[DP-1.1]`	list of all objects & their definitions
`[DP-1.2]`	NIAM schemes
`[DP-1.3]`	description of relationships and constraints
`[DP-1.4]`	object type catalogue

Details of the analysis

`[DP-1.1]` List of all objects & their definition:

The result of this sub-phase is a list of all (non-lexical) objects, including definition, synonyms, and attributes. The three subcomponents `[SC-1.1]` to `[SC-1.3]` are not yet distinguished.

Notation: structured text

```
NOLOT ...
Definition ...
Synonyms ...
LOT's ...
```

⟶ *3D-model node:* `[DP-1.1|SC-*|V-1]`[2]

[2] The '*' indicates that the node covers all the three subcomponents `[SC-1]` to `[SC-3]`, and therefore all specified nodes of the components dimension.

[DP-1.2] NIAM schemes:
Drawing the NIAM-schemes included decomposition of the application domain into the three subcomponents [SC-1] to [SC-3]. Relationships were added. Due to the NIAM concept, there is no straightforward hierarchization or aggregation of the entities, i.e., these two special types of relationships are not (easily) identifiable in the schemes.

Notation: NIAM schemes

⟶ *3D-model node:* [DP-1.2|SC-1|V-1]
[DP-1.2|SC-2|V-1]
[DP-1.2|SC-3|V-1]

[DP-1.3] Description of relationships and constraints:
The decomposition into three distinct subcomponents ([SC-1] to [SC-3]), performed in the phase before ([DP-1.2]) was not carried into this phase. The description of relationships and constraints is not subdivided into the subcomponents.

Notation: (formalized) text

⟶ *3D-model node:* [DP-1.3|SC-*|V-1]

[DP-1.4] Object type catalogue:
The object type catalogue meant defining an subtype hierarchy between the identified entities. The subtypes are distinguished by a "LOT" (lexical object type, i.e., an attribute) related to the super-type.

Notation: structured text

```
e.g., NOLOT: Model --- LOT: Model Type
      Geophysical types:  visco-elastic/rheologic
                          aroustic
                          mechanical
                          ...
      Geological types:   tectonostratigraphic
                          age
                          sedimentologic environments
                          ...
```

⟶ *3D-model node:* [DP-1.4|SC-*|V-1]

Design

The design phase consisted of two translations of the conceptual data model to 5th-order normal form relational tables [DP-2.1] and to a hierarchical description [DP-2.2]. The two translations don't represent two distinct sub-phases of the design phase, but rather two

documents in two different (design) notations that have been produced consecutively in two "sub-activities". Each of the two sub-activities results also in a description of file formats to organize the relational and hierarchical models (the results of this phase) into flat (ASCII) files.

`[DP-2.1]`: Translation to 5th-order normal form relations; no distinction of subcomponents.

Notation: "relations diagrams"

```
e.g.:    <-----------------> 
        ||SSM-Name||SSM-Type|CSY-Type|
```

$\longrightarrow$ *3D-model node:* `[DP-2.1.1|SC-*|V-1]`

The definition of the flat file format for the relational descriptions of the models can be seen as a *meta model description*, because the flat file format doesn't describe the geo-scientific models themselves, but in general the structure of the relational tables kept in flat files.

Notation: "structured text"

```
e.g.:  - file header
       - data dictionary ----- table/record id code string
       - table data records  -record length
                             - field start pos.
```

$\longrightarrow$ *3D-model node:* `[DP-2.1.2|SC-*|V-1]`

`[DP-2.2]`: Translation to hierarchical structures `[DP-2.2.1|SC-*|V-1]` took place using the relational description, adding and describing some extra constraints `[DP-2.2.2|SC-*|V-1]`. Again, there is no distinction of subcomponents. Here too, a flat file storage format is defined for storing the hierarchical models `[DP-2.2.3|SC-*|V-1]`.

Notation: "hierarchical diagrams" (cf. `[DP-2.1]`)

$\longrightarrow$ *3D-model node:* `[DP-2.2.1|SC-*|V-1]`
`[DP-2.2.2|SC-*|V-2]`
`[DP-2.2.3|SC-*|V-1]`

Decomposition of subcomponents

The subcomponents `[SC-1.1]` to `[SC-1.3]` could be decomposed into one more level of detail, i.e., into the level of **single entity types and relationship types** for the analysis phase (respectively single relations for the design phase).

Notation: (cf. the notations on the `[...|SC-1.*|V-1]`-level)

Subcomponents of [SC-1.1] *to* [SC-1.3]*:*

[DP-1.1|SC-1.*.x|V-1]	objects of the application domain object list
[DP-1.2|SC-1.1.x|V-1]	entity types and relationship types of the first subcomponent (semantic part ...)
[DP-1.2|SC-1.2.x|V-1]	entity types and relationship types of the second subcomponent (structure for topology ...)
[DP-1.2|SC-1.3.x|V-1]	entity types and relationship types of the third subcomponent (extra info)
[DP-1.3|SC-1.*.x|V-1]	relationships and constraints
[DP-1.4|SC-1.*.x|V-1]	objects in the object type catalogue
[DP-2.1.1|SC-1.*.x|V-1]	relations of the relational description
[DP-2.2.1|SC-1.*.x|V-1]	relations of the hierarchical description

Conclusion

This case is rather simple to classify, as it consists only of one view (the structural) of one subcomponent (the repository/data structure) of a system. As the final products are only specifications of formats and not a software system, the life cycle of this project is rather rudimentary, too. But it is exactly this simplicity that makes this case very suitable as introduction of the classification scheme of the 3D-model, and a useful pre-study for upcoming cases.

4.2 Case 2: GEOMOD

Based on the standard exchange format developed in the DEF project (cf. sec. 4.1), GEOMOD realizes a standard environment to support storage, import and export of 3D subsurface model data for third party applications (e.g., well correlation, seismic picking, 2D and 3D gridding/triangulation). The functionality of GEOMOD comprises exchange facilities for model data files of various formats, conversion functions to transform data in formats of the same geometrical type, or to a point geometry format of the same degree in topology, or a decreased topology degree. The system comprises a graphical user interface and an internal meta data structure implemented in a relational database management system.

Several documents of the project documentation were used as input for this case study: the Software Project Management Plan (SPMP) [Rit92], the Software Requirements Specification (SRS) [Flo92c], the Global Software Design Description (gSDD) [Flo92b], and the Detailed Software Design Document (dSDD) [Flo92a]. Project staff was interviewed, too.

Contents:

The Software Project Management Plan

In this case study, the software project management plan (SPMP) [Rit92] was mainly used as a first overview to determine the intention of the project. The projected system is outlined in terms of functionality and informally broken up into components that should realize the different parts of the functionality of the system.

Although the SPMP is one of the first documents of the project, already a number of "hard" constraints are given, as e.g., the use of an RDBMS for the storage of meta data, or the use of XFaceMaker as user interface builder.

From the SPMP, an overview of the (projected) system components can be derived, although the assignment of each component to a particular category of the system components dimension `[SC]` is sometimes ambiguous. The second column of the following table indicates the assignments of components to system component categories.

Notation: informal text

$\longrightarrow$ *3D-model node:* `[DP-1|SC-x|(functional/structural)]`[3]

Subsystems identified in the SPMP:

`[SC-1]`	`[process]`	exchange facilities
`[SC-2]`	`[interface]/[process]`	data loading subsystem
`[SC-3]`	`[interface]/[process]`	export subsystem
`[SC-4]`	`[interface]`	user interface
`[SC-5]`	`[repository]`	internal data structures

With *exchange facilities* `[SC-1]`, the conversion functions and related support functions are meant, so it is appropriate to put them into the component category `[process]`. The *loading subsystem* incorporates two sets of functionality: an interface to load data model definitions and a processing part to convert them to relational data structures. Therefore, this subsystem belongs to two system component categories. This holds analogically for the *export subsystem* which produces GEOMOD compliant or other external exchange files. The ambiguity of **multiple attribution** can be "resolved" in this case, if the *file system*, on which the software works, is integrated in the notion of the GEOMOD "system". This is a reasonable integration for two reasons:

1. The (NSF) file system is mentioned anyway as repository for the data files. In the gSDD, not only the file system itself, but also UNIX functions to access the file system

[3] As the description of the components in the SPMP is (very) informal, no real `[V]`-dimension is assigned to those nodes.

will be mentioned. According to the object-oriented notion, the file system plus UNIX functions can be seen as an individual, high-level object (or subsystem, in the 3D-model notion).

2. Ambiguities, such as the `[process]`/`[interface]` problem of the *loading* and *export subsystems* can be resolved, because now those two components don't have an interface to the outside of the system anymore, but only to another component of the system. The multiple assignment of component categories to components is thereby shifted to the decomposition of those ambiguous components, where this problem can be resolved by an extra 3D-model for each component.

Subsystems identified in the SPMP (revised):

`[SC-1]`	`[process]`	exchange facilities
`[SC-2]`	`[process]`	data loading subsystem
`[SC-3]`	`[process]`	export subsystem
`[SC-4]`	`[interface]`	user interface
`[SC-5]`	`[repository]`	internal data structures
`[SC-6]`	`[repository]`	NSF distributed environment

The *user interface* `[SC-4]` is supposed to have a graphical part (using XFaceMaker as construction tool and as widget resource) and also a character-based part. To store the *internal* (meta) *data structures* `[SC-5]`, the use of a relational DBMS (ORACLE) is planned.

Development Phases

In the SPMP [Rit92], a number of development phases `[DP]` are defined (the resulting documents/products are given in the third column of the following table):

Development phases of GEOMOD:

`[DP-1]`	preparation	proposal, SPMP, SQAP, contracts
`[DP-2]`	requirements specification	software requirements specification
`[DP-3]`	prototyping	prototype report[4]
`[DP-4]`	design	global/detailed software design description, test plan
`[DP-5]`	implementation	code, user documentation
`[DP-6]`	testing	test report
`[DP-7]`	case study (benchmark)	benchmark test report
`[DP-8]`	installation & checkout	installation manual

In some phases, a number of different documents had to be produced. Those different documents describe different views, or subcomponents of the system at that particular development phase. Hence, they result in different nodes in the 3D-model graph, although they could as well be considered as different documents to produce within one specific node.

[4] The prototype itself is a deliverable of the prototyping phase, too, but is not mentioned here, as it was not made available to the customers.

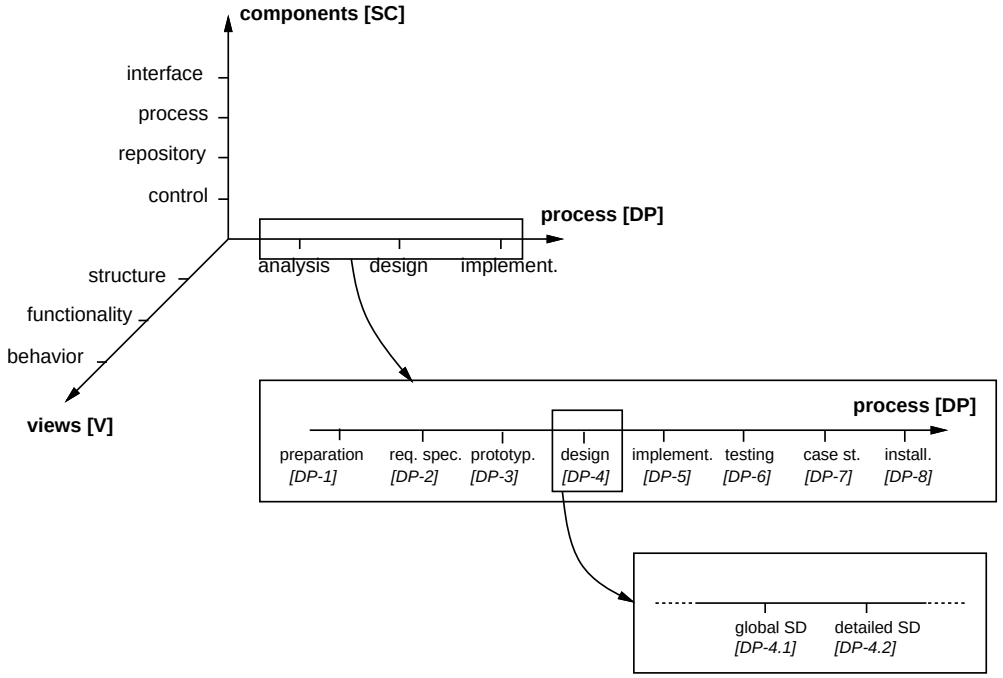

Figure 4.2: The development phases of GEOMOD

Further constraints are defined in the SPMP:

- Methods & Standards:
  - Yourdon Structured Design
  - NIAM / Extended ER
  - C (TNO Coding Standards)
  - SQL
  - Motif Style Guide (UI design)
  - ANSI/IEEE standards based testing
- Tools:
  - STP
  - X-NIAM
  - XFaceMaker (UI design)

These technical constraints have to be attributed as "constraints of realization" to the appropriate node of the 3D-model graph. "XFaceMaker" for example, can be attributed to two different nodes. It is a **tool constraint** for the *prototyping phase* `[DP-3]`, because XFaceMaker is used to prototype/construct the user interface. But it is also an **implementation constraint** for the *implementation phase* `[DP-5]` because the implementation of the user interface uses XFaceMaker classes and objects.

The node of the 3D-model representing the *preparation phase* [DP-1] can, in principle, be subdivided into 3 "sub-nodes". The *proposal* [DP-1.1] evolved into two distinct documents concerning two viewpoints of the project: the *SPMP* which describes strategic and technical aspects [DP-1.2.1], and the *contracts* which describe the legal and financial aspects of the project [DP-1.2.2].

The Software Requirements Specification

Apart from the functional requirements, the Software Requirements Specification (SRS) [Flo92c] also presents the application domain, refers to the logical data model (DEF, cf. sec. 4.1) and (informally) describes user interface characteristics and some more technical constraints. These sections (of the SRS) can be seen as distinct nodes of the 3D-model, too.

User interface requirements:

Two types of user interfaces are required: a textual user interface which works with input sentences (commands), and a graphical user interface which allows the user to specify commands and parameters via a menu structure. The way the UI works, i.e., its behavior (syntax checking + parsing), is described, and the foreseen commands are listed in a sketch of a pull-down menu bar. This list of commands can be interpreted as list of potential events (provided by the user) that trigger the behavior of the system. The behavior is further not specified formally (e.g., via state transition diagrams), but was validated later via the prototype and described implicitly, together with other aspects for the rest.

Notation: text, (informal) diagrams

$\longrightarrow$ *3D-model node:* [DP-2|SC-3/4|V-1] [6]
[DP-2|SC-3/4|V-3] [7]

Technical Constraints:

Additional technical constraints are given in the SRS:

- OSF-Motif Style Guide, POSC E& P Style Guide for the looks of the user interface
- single site, single user, single job
- Unix workstations, DEC-5000
- OSF-Motif Window Manager
- C
- ASCII files (as external data files)

[6] [SC-3/4] are the 'edit file' and 'exchange format definition' components, defined in the functional requirements of the SRS (cf. below). [V-1] is the structural view of the system.

[7] [V-3] is the behavioral view of the system.

- ORACLE (for the relational (meta) database), embedded SQL
- file exchange through NFS protocol

These are all constraints that have to be attributed to the appropriate nodes (once they are all established).

Functional requirements:

An important section of the SRS is the (formal) specification of the functional requirements. The main functions are decomposed (cf. the following table), and the functions on the lowest level of decomposition are described.

Notation: data flow diagrams, structured text,

Main functions of the system:

[SC-1]	[process]	Exchange
[SC-2]	[control]	File management action
[SC-3]	[interface]	Edit file
[SC-4]	[interface], ([process])	Exchange format definition
[SC-5]	[control]	Event handler

$\longrightarrow$ *3D-model node:*[DP-2|SC-x|V-2] [8]

It is notable that this list of 5 subcomponents differs essentially from the arrangement of subcomponents in the SPMP. This can be attributed to the different viewpoint that was taken here (functionality [...|...|V-2]) in comparison to the SPMP (structure [...|...|V-1]).

Logical model / database requirements:

The meta model, i.e., the definition of the standard exchange format was scheduled to be stored in a relational database (ORACLE). The basis/input for the relational tables of that relational database is the so-called *logical data model* which is defined in terms of *ER-diagrams* and *object definitions* (i.e., description their attributes and attribute values). The logical data model is based on the results of the DEF-project (cf. sec. 4.1), where the specification technique *NIAM* was used. Here, ER-diagrams were used because of the constraints imposed by one of the project partners.

Notation: ER-diagrams, structured text

$\longrightarrow$ *3D-model node:*[DP-2|SC-4|V-1]

The process *exchange format definition* [SC-4] is only one possibility to attribute the meta data structure to a system component of the *analysis phase* [DP-2]. It is not obvious from the description of the functional requirements that this data structure belongs to the exchange format definition subcomponent, because it is mentioned in the context of other subcomponents, too. The logical data model can also be seen as a further development of the *internal data structures* subcomponents mentioned in the SPMP [DP-1|SC-5|(V)] into

[8] [V-3] is the functional view of the system.

a more elaborated node [DP-2|SC-5.(repository)|V-1]. The reason for ambiguities like this is that the functional decomposition presented in the SRS does not lead to a definition of a system architecture for this system.

The Global Software Design Description

The Global Software Design Description [Flo92b] establishes a first concrete picture of a system architecture for the GEOMOD system. It explicitly defines "subsystems" and describes them (textually). Additionally, the database tables for the ORACLE meta database are defined.

Development sub-phases of [DP-4] (design)

Design sub-phases:		
[DP-4.1]	global design	gSDD [Flo92b]
[DP-4.2]	detailed design	dSDD [Flo92a]

The GEOMOD global system architecture

In this first part, the GEOMOD system is partitioned into nine subsystems. An overview of the subsystems together with their interactions is presented in a figure (sec. 3.2 of the gSDD), then textually explained (sec. 3.3 of the gSDD). While the figure sketches the structure of the system [V-1], the textual descriptions mainly address functionality and behavior of the components [V-2/3].

Notation: (informal) diagrams, text

Subcomponents of the GEOMOD system:		
[SC-1]	[interface]	User interface
[SC-2]	[control]	Command handler
[SC-3]	[control]	Message handler
[SC-4]	[control], [interface]	File management and edit functions
[SC-5]	[process]	Exchange functions
[SC-6]	[process]	Utilities
[SC-7]	[repository]	Meta map functions
[SC-8]	[repository]	Initialize functions
[SC-9]	[control]	File handler

$\longrightarrow$ *3D-model node:* [DP-4.1|SC-x|V-1]

The gSDD clearly distinguishes other system components than the SRS. So, an obvious question is what the mapping in terms of 3D-model nodes between the 5 subcomponents (in that case: subprocesses) identified in the SRS [DP-2|SC-x|V-2] and the 9 subcomponents listed here [DP-4.1|SC-x|V-1] is. The following table and fig. 4.3 give an overview:

SPMP [DP-1]	SRS [DP-2]	global SDD [DP-4.1]
Exchange facilities [SC-1]	Exchange [SC-1]	Exchange functions [SC-5] Utilities [SC-6]
Loading subsystem [SC-2]	(subprocess 1.4)	Initialize [SC-8] File handler [SC-9]
Export/import subsystem [SC-3]	File management action [SC-2] Edit file [SC-3]	File management and edit functions [SC-4]
Internal data structures [SC-5]	Exchange format definition [SC-4]	Meta map functions [SC-7]
User interface [SC-4]	Event handler [SC-5]	Command handler [SC-2] Message handler [SC-3] User interface [SC-1]

In figure 4.3, the main specification trajectory is indicated by the bold (vertical) specification "lines" (a, b, c). The SRS (b) receives additional input from the informal descriptions of functionality (a1) and behavior (a2) in the SPMP. Informal description of functionality and behavior is also included in the detailed design description (c1, c2).

The *user interface* and the *utility functions* have not been (formally) functionally specified during the *requirements specification* [DP-2], and they have not been specified in the scope of another subcomponent in that phase. The *initialize functions* can be traced back to the subprocess 1.4 of the *exchange* process ([DP-2|SC-1|V-2]).

Details of the subcomponents [SC-1] to [SC-9]

In the global Software Design Description [Flo92b], after the figure with the general system architecture, more details for each subcomponent are given. The components are subdivided into more specific subcomponents, and their behavior, functionality and interactions with other subcomponents are described in some more detail. So these descriptions are not only concerned with the fine structure of the nine system components, but incorporate also their functional and behavioral views, at least on an informal level (e.g., "[The command handler] will parse the sentence to come up with a command and a number of arguments."). This means that there is a change of "coordinate" (in terms of the 3D-model) in the views dimension [V], when going from the global system architecture figure to the descriptions of the subsystems, thereby providing a "complete" picture of the views dimension.

Notation: (informal) diagrams, text

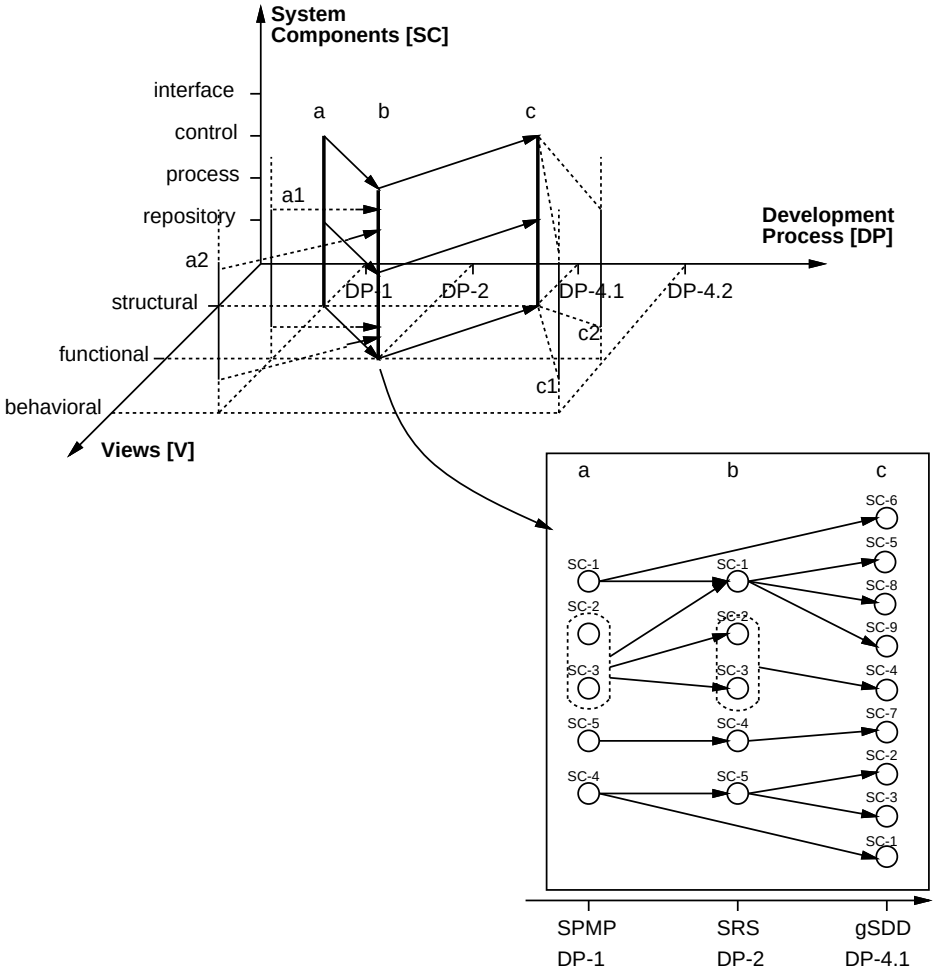

Figure 4.3: The subcomponents dimension along the phases

User interface `[DP-4.1|SC-1]`**:**

Subcomponents of the User interface component:		
`[SC-1.1]`	`[interface]`	Graphical user interface
`[SC-1.2]`	`[interface]`	Textual user interface
`[SC-1.3]`	`[process]`	Syntax check

From the textual descriptions, several relationships/interactions with other system components can be derived. These interactions are not always described as taking place between two components on the same level of decomposition. Often, relationships are identified between software components of different granularity levels, e.g. between components of the uppermost level (`[SC-1]` to `[SC-9]`), and the subcomponents of those components (`[SC-x.y]`).

Relationships involving UI subcomponents:

[SC-1.1]	triggers by sentences	Parser [SC-2.1]
[SC-1.2]	uses to check sentences	[SC-1.3]
[SC-1.1]	receives messages (text) from	Error handler [SC-3.1]
[SC-1.1]	receives messages (text) from	Status handler [SC-3.2]
[SC-1.2]	receives messages (text) from	Error handler [SC-3.1]
[SC-1.2]	receives messages (text) from	Status handler [SC-3.2]

Command Handler [DP-4.1|SC-2]:

Subcomponents of the Command handler component:

[SC-2.1]	[process]	Parser
[SC-2.2]	[control]	Executor

Relationships involving Command Handler subcomponents:

[SC-2.2]	evokes	File manager [SC-4], Meta map functions [SC-7], Exchange functions [SC-5], Utilities [SC-6]
[SC-2.2]	receives status from	File manager [SC-4], Meta map functions [SC-7], Exchange functions [SC-5], Utilities [SC-6]
[SC-2]	receives sentences from	Graphical user interface [SC-1.1]
[SC-2]	receives sentences from	Textual user interface [SC-1.2]
[SC-2]	triggers (error mess.)	Error handler [SC-3.1]

In the descriptions, the generic term "evoking the appropriate function" is used, but more details can be derived from sections describing the other components and from the architecture diagram.

Message Handler [DP-4.1|SC-3]:

Subcomponents of the Message handler component:

[SC-3.1]	[control]	Error handler
[SC-3.2]	[control]	Status handler

Relationships involving Message handler subcomponents:

[SC-3.1]	receives error code	Executor [SC-2.2]
[SC-3.2]	receives status code	Executor [SC-2.2]
[SC-3.1]	sends error message string to	User interface [SC-1]
[SC-3.2]	sends status message string to	User interface [SC-1]

The *error handler* receives error codes via the *executor* (as return codes of functions in other subsystems), but also directly from other subcomponents (for intermediate "pop-up warnings"). The same holds for the *status handler*, which receives status codes from the executor as well as directly from other subcomponents.

File management & edit functions `[DP-4.1|SC-4]`**:**

Subcomponents of the FM& E component:		
`[SC-4.1]`	`[interface]`	Edit functions
`[SC-4.2]`	`[control]`	File management

This system component is realized by using UNIX functions, so this component isn't really decomposed and described in this phase. Instead, details are postponed to the *implementation phase* `[DP-5]`.

Relationships involving FM& E subcomponents:		
`[SC-4]`	calls	(UNIX functions)
`[SC-4]`	receives status/error codes from	(UNIX functions)

Exchange functions `[DP-4.1|SC-5]`**:**

Subcomponents of the Exchange functions component:		
`[SC-5.1]`	`[interface]`	Load/save functions
`[SC-5.2]`	`[process]`	Translate functions
`[SC-5.3]`	`[interface]`	Item-level access functions
`[SC-5.4]`	`[repository]`	GEOMOD DB
`[SC-5.5]`	`[repository]`	UNIX file system

The *UNIX file system* `[SC-5.5]` is mentioned here as a component, because the system uses the file system as repository for data, i.e., external GEOMOD files and third party product files.

Utilities `[DP-4.1|SC-6]`**:**

The *utilities* are a subsystem that was not explicitly mentioned before. It can be attributed to the *exchange subsystem* of the *preparation phase* `[DP-1|SC-1]`, though. The following table gives an overview of the purpose of the utility functions.

Subcomponents of the Utilities component:		
`[SC-6.1]`	`[process]`	Retrieve modeled objects
`[SC-6.2]`	`[process]`	Size of database
`[SC-6.3]`	`[process]`	Delete corrupted database
`[SC-6.4]`	`[process]`	Regular to irregular transformation

Meta-map functions [DP-4.1|SC-7]:

Subcomponents of the Meta-map functions component:		
[SC-7.1]	[interface]	Delete format definition
[SC-7.2]	[interface]	Define format definition
[SC-7.3]	[interface]	Edit format definition
[SC-7.4]	[interface]	Load/save format definition
[SC-7.5]	[interface]	View format definition
[SC-7.6]	[repository]	Meta DB (tables)
[SC-7.7]	[repository]	Translation table
[SC-7.8]	[repository]	Format descriptions

The GEOMOD data model descriptions are stored in the (ORACLE) meta database (tables) and also in an external ASCII-file [SC-7.6].

Initialize [DP-4.1|SC-8]:

The *initialize function* could also be attributed to the *meta-map functions* [SC-7], but it was chosen to be put it apart, because it is not only used when manipulating the format definitions, and also has two different implementations (cf. dSDD [DP-4.2]).

Subcomponents of the Initialize component:		
[SC-8.1]	[interface]	Get meta-data

Relationships involving Initialize subcomponents:		
[SC-8.1]	retrieves from	Meta DB tables [SC-7.6]

File Handler [DP-4.1|SC-9]:

Subcomponents of the File handler component:		
[SC-9.1]	[interface]	Get data
[SC-9.2]	[interface]	Put data
[SC-9.3]	[repository]	Internal GEOMOD binary files

Again, the *internal GEOMOD binary files* [SC-9.3] are rather objects within the *UNIX file system* "subcomponent" [SC-5.5] than a subcomponent of its own.

Relationships involving File handler subcomponents:		
[SC-9.1]	handles request from	Load/save functions [SC-5.1], [5.3]
[SC-9.2]	handles request from	Load/save functions [SC-5.1], [5.3]

Detailed Software Design Description

In the Detailed Software Design Description [Flo92a] ([DP-4.2]), the subcomponents described in the previous phase are grouped together into executables. These executables are

specified, i.e., the functions they contain in terms of identification, description, input/output parameters, return values are defined, together with other files that go along with them. Also described are: the directory structure, environment variables, library modules, data entities, data model template files, functional flow.

Executables:

Notation: (informal) diagrams, (informal) text, structured text (e.g., module descriptions)

Components of [DP-4.2] *(= executables):*

		Relationship to components of [DP-4.1]:
[SC-1]	start_geomod	-
[SC-2]	geomod (kernel)	[SC-1], [SC-2], [SC-3], [SC-4], [SC-7.1-5], [SC-8.4]
[SC-3]	transfer	[SC-5.1], [SC-8.4], [SC-9.1-2]
[SC-4]	translator	[SC-5.2-3], [SC-9.1-2], [SC-8.4]
[SC-5]	u_functions	[SC-6]
[SC-6]	ge_meta_data	[SC-8.4]

Two versions of *get_meta_data* exist: one is a function incorporated in various executables to copy a meta-data file to the right directory ([SC-2], [SC-3], [SC-4]), the other is an independent executable [SC-6] to retrieves the meta-data from an ORACLE database.

The headers of each function in the executables are also specified, but they are not listed here to keep the size of the case study limited. They can be seen as refinement of the six components (executables) listed above ([DP-4.2|SC-x.y|V-1/V-2] - as both structure and functionality are described).

Additional files

Notation: (informal) text, structured text (e.g., module descriptions)

file name:	contents:	belongs to:
geomod.cfg	configuration parameter	start_geomod [SC-1]
geomod_error.txt	error texts	start_geomod [SC-1]
geomod_help.txt	help texts	start_geomod [SC-1]
geomod_setup	script for setting environment variables	start_geomod [SC-1]
gmdpw	encrypted password	start_geomod [SC-1]
geomod_dm.dat	ascii description of the GEOMOD data model	Meta database [DP-4.1|SC-7.6]
geomod_format.dat	list of formats supported	View format definition [DP-4.2|SC-7.5]
geomod_logo.bm	GEOMOD logo bitmap	start_geomod [SC-1]
GEOMOD_topAS	X-resource file	start_geomod [SC-1]
XFaceMaker2 library		geomod (kernel) [SC-2], or GUI [DP-4.1|SC-1.1]
libXfmExt.a	(on top of XFaceMaker)	geomod (kernel) [SC-2], or GUI [DP-4.1|SC-1.1]

libGeomod.a	library of kernel object modules	geomod (kernel) [SC-2]
libGeomodExt.a	for linking translators, u_functions and get_meta_data module [DP-4.1|SC-8.4] → translator [SC-4]	
libTrans.a	linking the loader/saver	translator [SC-4]
ORACLE DBMS	contains the GEOMOD data model	get_meta_data [SC-6]

Further remarks & Conclusions

bf Prototyping was applied in the GEOMOD project, too, but is not documented in detail here, because prototype and prototyping report were not available anymore. Furthermore, rather than providing essential structural information about the system, prototyping served to **validate two very specific aspects of the system**:

1. increase acceptance by getting feedback from users about a user interface prototype.
2. test the performance of flat file access vs. retrieval from a database for keeping the data files. On basis of the prototyping results, flat file access was chosen as storage format for the data, a database for keeping the model meta-data.

In order to limit the size of this case study to fit into the scope of this report, no further phases (and documents) have been included here. A more detailed case study is performed on SISTRE, another software system developed at TNO, and can be found in [Rav94]. There, also some of the aspects not treated here should be addressed.

Reflecting the concepts introduced in this report, this case study concentrates on **identifying the nodes of the 3D-model** which correspond to development activities and documents of the GEOMOD project. It can be seen that

1. only a small part of the possible nodes, i.e., only a part of the possible activities has been performed;
2. often, the descriptions of several constituents of a dimension (e.g., functionality and behavior of the views dimension) have been integrated into one specification, i.e., they are described by one and the same piece of document;
3. until the coding phase, only very few specification techniques or notations have been used. Most of the specification are given in (informal) text.

Furthermore, a considerable **switch in views**, from which the system is mainly described, can be observed in the transition from preparation to analysis to design (cf. fig. 4.3). *Software quality metrics* have to be defined to evaluate whether such a "zig-zag image" reflects a good and solid software development.

Chapter 5

Conclusions

This report introduced a model to structure software engineering in terms of *independent dimensions* and their *integration into a uniform framework*, the "3D-model". This approach is independent from any specific software engineering methodology, but allows to make use of their techniques and notations to a desired extend for the specification in the scope of an actual software development effort. In this sense, the 3D-model is a meta-model of software engineering which has to be adapted for real use. The advantages of using such a meta-model are

- the better and *clearer structuring* of the elements of software engineering, and therefore
- an *increased understanding* of the elements of software engineering and their mutual relationships, and
- *more freedom and adaptability* of customizing the software development for the actual development situation.

But the concepts presented so far in this report are only a first approach to an integral solution of meta-modeling software engineering. There are still a lot of open problems to tackle in the future, as e.g.:

Formalizing the relationships between the 3D-model nodes:

The nodes of the 3D-model graph representing the various elements of software engineering have been introduced. They can described as combination of constituents of the three dimensions (cf. chapt. 2). Now, the **possible types of relationships** (dependencies, etc.) between those elements, i.e., the edges between the corresponding 3D-model nodes, have yet to be captured. The relationships describe dependencies between the elements of software engineering, and the edges formalize them in the scope of the 3D-model graph. They constrain the possible way of working, because they order and relate the activities connected to the production of the specifications implied by the nodes.

A meta-meta-model:

The 3D-model self has to be specified formally, using the notion of a "three-dimensional", directed graph. As the 3D-model is a meta-model (of software engineering) itself, this formalism will be a **meta-meta-model**. A first attempt to adapt an already established meta-model

[SIWyS93] to formalize the specification of the 3D-model revealed that the concepts of the 3D-model first have to be understood and captured thoroughly before a "meta-meta-model" can be established. Nevertheless, finding and using a formal (graph) language for the 3D-model graph is an important prerequisite for capturing precisely the constituents, nodes, decomposition, relationships, etc. of the 3D-model, without relying on pre-defined, generic categories as done in this report.

Deriving a specific software engineering methodology:

The steps described in chapter 2 to come from a abstract framework of software engineering (the "3D-model) to a **practicable and tailored software engineering methodology** (cf. sec. 2.6 - *Applying the 3D-model*) have to be performed. Additional input for deriving the proper elements, dependencies, notations, etc. has to be provided by performing more case studies.

Bibliography

[Ald91] Albert Alderson. Meta-CASE Technology. In Albert Endres and Herbert Weber, editors, *Software Development Environments and CASE Technology, European Symposium*, volume 509 of *Lecture Notes in Computer Science*, pages 81–91, Berlin, 1991. Springer.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes, editors. *Object-Oriented Development: The FUSION Method.* Prentice-Hall, Inc., 1994.

[FkNO92] Christer Fernström, Kjell-Håkan Närfelt, and Lennart Ohlsson. Software Factory Principles, Architectures, and Experiments. *IEEE Software*, pages 36–44, March 1992.

[Flo92a] Frans J. T. Floris. GEOMOD: Data Exchange Software for 3D Subsurface Model Data, Detailed Software Design Document V1.0. Technical Report OS 92-104C, TNO Institute for Applied Geoscience, November 1992.

[Flo92b] Frans J. T. Floris. GEOMOD: Data Exchange Software for 3D Subsurface Model Data, Global Software Design Description V1.0. Technical Report OS 92-67C, TNO Institute for Applied Geoscience, July 1992.

[Flo92c] Frans J. T. Floris. GEOMOD: Data Exchange Software for 3D Subsurface Model Data, Software Requirement Specification V1.1. Technical Report OS 92-46C, TNO Institute for Applied Geoscience, November 1992.

[Ger93] Bart Gerritsen. Probe - Software Quality Assurance Plan. Technical Report OS 93-52-C, TNO Institute for Applied Geoscience, 1993.

[GS93] David Garlan and Mary Shaw. Architectures for Software Systems. Tutorial Notes, 15th International Conference on Software Engineering, Baltimore, June 1993.

[HS93] B. Henderson-Sellers. The O-O-O Methodology for the Onject-Oriented Life Cycle. *ACM Sigsoft, Software Engineering Notes*, 18(4):54–60, October 1993.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[KP88] Herb Krasner and Stephen Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August 1988.

[Nat91] National Institute of Standards and Technology. *Reference Model for Frameworks of Software Engineering Environments*, second edition, 1991. NIST Special Publication 500-201 (Technical Report ECMA TR/55.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tschritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.

[Obj90] Object Management Group, Inc. *Object Management Architecture Guide*, 1.0 edition, November 1990. OMG TC Document 90.9.1.

[Obj91] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, 1.1 edition, December 1991. OMG Document 91.12.1.

[PRK92] Simon Pen, Ipo Ritsema, and Theo Kemme. Data Exchange Format (DEF) for Geological and Geophysical 3D Subsurface Modelling Techniques. Technical Report OS 92-83A, TNO Institute for Applied Geoscience, September 1992. reprint.

[PW92] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM Sigsoft, Software Engineering Notes*, 17(4):40–52, October 1992.

[Rav94] Jan Ravensberg. Applying the 3D-model of software engineering: An extensive, evaluating case study (working title). Master's thesis, Leiden University (RUL), August 1994. In preparation.

[RBP+91] James Rumbaugh, Michael Blaha, William Premberlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[REE89] R. Rock-Evans and B. Engelien. *Analysis techniques for CASE: a Detailed Evaluation.* Ovum Ltd., 1989.

[RG92] Kenneth S. Rubin and Adele Goldberg. Object Behavior Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[Rit92] Ipo Ritsema. GEOMOD: Data Exchange Software for 3D Subsurface Model Data, Software Project Management Plan V1.0. Technical Report OS 92-31C, TNO Institute for Applied Geoscience, May 1992.

[SIWyS93] Motoshi Saeki, Kazuhisa Iguchi, Kuo Wen-yin, and Masanori Shinohara. A Meta-Model Representing Software Specification and Design Methods. Technical report, Tokyo Institute of Technology, Dept. of Electrical & Electronical Engineering, 1993.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented System Analysis: Modeling the World in Data.* Yourdon Press, 1988.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1992.

[STO89] Izhar Shy, Richard Taylor, and Leon Osterweil. A Metaphor and a Conceptual Architecture for Software Development Environments. Technical report, University of California, Dept. of Information and Comp. Science, July 1989.

[YC79] E. Yourdon and L.L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.

[You89] E. Yourdon. *Modern Structured Analysis*. Yourdon Press, 1989.