Chapter 2

SYSTEMS AND CONTEXT

There are numerous scientific disciplines: some are purely scientific, such as *mathematics*, *physics*, and *biology* while others are applied, such as *computer science* and *engineering* [80]. In considering any discipline nevertheless, the notion of **`system`** is an important one [19]; in *physics*, they study *physical systems*, in *biology*, they study *biosystems*, in *sociology*, they study *social systems*, and so on. Hence, the development of the *General Systems Theory* has been inspired [7,78], referred to as **`systemics`**. *Systemics* focuses on the characteristics of systems across the barriers between scientific disciplines. Such a perspective is considered important with regard to *EIS* since in approaching *EIS*, one would have to deal with *social systems* (because there are human entities, human behavior, and so on, in any enterprise), also with *technical systems* (because there are technical devices, software applications, and so on, in any information system). Hence, both social systems and technical systems would not only need to be studies in isolation but it is also necessary to understand their interrelationships.

For this reason, firstly in the current chapter, we will clarify what we mean by '*system*' and then we will touch upon *enterprise systems* and (*enterprise*) *information systems* – all considered essential with regard to the focus of this book. Secondly, we will explicitly discuss not only the *construction* of any system, by considering ontological systems, but also its *function*, emphasizing as well on the distinction between the two, reflected in two essential perspectives on system behavior: a. the *black-box perspective* considering what the system is delivering to its environment (functionally) and b. the *white-box perspective* considering how the system is delivering this. Thirdly, we will touch upon the *evolvability* of any technical (software-intensive) (sub-)system, part of an *EIS*, by considering *combinatorial effects*, in general and the *Normalized Systems Theory*, in particular. Finally in the chapter, we will consider context as an essential notion with regard to the system environment.

2.1 Systems

The *General Systems Theory* (already mentioned) proposes a unified approach in considering a system, based on the (justified) claims that there are some:

- concepts and structural principles that seem to hold for systems of many kinds;
- modeling strategies that seem to hold everywhere.

That has inspired Bunge [10] to consider theories that focus on the structural characteristics of systems and can therefore cross the 'largely artificial' barriers between disciplines. Such efforts have triggered interest to discover similarities among systems of many kinds despite their specific differences, such that studying current (complex) enterprises would become easier [54] – this often assumes de-emphasizing the aspects concerning the particular scientific discipline, focusing instead on the *structure* and the *behavior* of the system as such. This even goes beyond systemics and points to the broader notion of **`system analysis`**, as defined by Bunge: the essential goal behind system analysis is to enable one understand how a system operates.

Since those views are considered relevant to our focus on systems in general and enterprise systems (and *EIS*), in particular, we have adopted the *system definition* proposed by Bunge [10]:

DEFINITION 1 Let T be a nonempty set. Then the ordered triple $\sigma = \langle C, E, S \rangle$ is **`system`** over T if and only if C (standing for *Composition*) and E (standing for *Environment*) are mutually disjoint subsets of T (i.e. $C \cap E = \emptyset$), and S (standing for *Structure*) is a nonempty set of active relations on the union of C and E. The system is *conceptual* if T is a set of conceptual items, and *concrete* (or material) if $T \subseteq \Theta$ is a set of concrete entities, i.e. things.

The *system definition* of Dietz [19] is consistent with the above definition, acknowledging that among the *properties* of a system are:

- *composition*: a set of elements of some category (physical, social, biological, etc.);
- *environment*: a set of elements of the same category; the composition and the environment are disjoint;
- *structure*: a set of influence bonds among the elements in the composition, and between the elements in the environment.

Nevertheless, Dietz considers one more property, namely *production*, pointing that:

- the elements in the composition produce things, such as goods, services, and so on, that are delivered to elements in the environment.

For us, the *composition-environment-structure system view* is appropriate because even though production characterizes most systems, we claim that it is also possible that the composition elements of a system stay inactive (for a period of time or forever), still being part of the system.

Further, in line with the systemics views, we would consider further system categorization depending on the (research) area of interest; some examples of such categories are:

- *legislative system* – a system concerning legal norms and acts;
- *planet system* – a system concerning planets;
- *political system* – a system concerning political subjects.

Since our focus is on enterprises and information systems supporting enterprises, we are interested in two system categories, namely:

- *enterprise system*;
- *EIS*.

As for the *enterprise system concept*, it should correspond to a view on business, in general, and for this we refer to [54]: by '*business thing*', it is not meant only things concerning trade/commerce but also all things that refer to any *organized activity* which is driven by a particular *goal*. Next to that, businesses are envisioned as *human-driven* since humans are those through whom businesses operate. Hence, inspired by the views of Shishkov and Dietz [56], we propose the following definition:

DEFINITION 2 A system should be considered being an **`enterprise system`** if and only if it is composed of *human entities* collaborating among each other through *actions* which are driven by the *goal* of delivering *products* to entities belonging to the environment of the system.

By '*product*' we mean anything that is or can be delivered to a customer, no matter if it is a material thing (often called *product* or *goods*) or an immaterial thing (often called *service*), and this is referred to as a *production fact*.

In the same spirit and inspired by [54], we propose the following *EIS definition* where '*ICT*' stands for '*Information and Communication Technology*':

DEFINITION 3 A system should be considered being an **`EIS`** if and only if it is composed of *human entities* (often facilitated by ICT applications as well as by technical and technological facilities) collaborating among each other driven by the *goal* of *supporting informationally* a corresponding enterprise system.

Definition 2 and *Definition 3* both reflect the *ontological* (constructional) essence of the addressed system categories. This is claimed to be insufficient nevertheless with regard to EIS because an enterprise information system is not only about structurally bringing together different human and technical entities but it is also about enabling technical entities, such as devices, ICT applications, and so on, to support corresponding human entities accordingly. We argue that in order to achieve deep understanding on this, one would also need a *functional view* as well, such that one could step in the shoes of a particular human entity and understand the way this human entity is supported functionally by a device and/or ICT application. For this reason, we propose also another *EIS definition* that assumes a *functional perspective*, inspired by [54]:

DEFINITION 4 Concerning its *functional characteristics*, an **EIS** is a system which *manipulates data* and normally serves to *collect, store, process* and *exchange* (or *distribute*) *data* among users within or between enterprises, or among people within wider society.

In the following two sections, we will subsequently consider enterprise systems and EIS.

2.2 Enterprise Systems

In considering *enterprise systems*, we stick to *Definition 2*, according to which the goal of delivering products to the environment is essential and for this reason we take this as an important criterion for determining whether or not a particular entity belongs to an enterprise system. Only entities driven by the same *goal* would be considered belonging to the same enterprise system. If a consultancy company is also dealing with property rental, for example, then the human entities and activities about property management should not be considered belonging to the consultancy enterprise system since they are irrelevant with respect to the consultancy goal, and similarly, the human entities and activities about consultancy should not be considered belonging to the property renting enterprise system. Hence, this is all about the role and behavior that a particular human entity takes, not about the formal belonging of the entity to one organization or another. Further, this goal-driven criterion is not in conflict with our adopting a *composition-environment-structure system view* (as discussed already) since the goal itself (delivering consultancy, for example) may be existing and entities in relevant roles may be existing but this does not mean that those entities are active.

Hence, although it does not directly concern the composition and structure of an enterprise system, the goal driving it has to be taken into account when considering such a system.

Further, in identifying an enterprise system, it is important being aware of the *actions* and *human entities* (as well as the *roles* in which they appear) that are relevant to the system.

As for **actions** that may take place in an enterprise system, we distinguish between two action types, namely *production* and *communicative* (*coordination*) *ones*: *Production actions* (or *acts)* concern a particular output in the form of a material product or an immaterial product while *Communicative* (*coordination*) *actions* (or *acts*) concern the collaboration within the enterprise system; this collaboration is in support of the realization of (corresponding) production actions [54].

As for **human entities** and the **roles** in which they appear, we consider just the *actor-roles*: the roles being fulfilled by corresponding human entities; this we consider adequate for enterprise analyses because otherwise it would be confusing considering some entities who may appear in different roles, including non-typical ones (for example: a professor sending a fax, thus fulfilling the role *secretary*). Hence, we are interested in the role and not in the particular human entity fulfilling the role.

We thus view an enterprise system as a collection of actions and corresponding actor-roles: the actor-roles are the composition elements of the system while the actions concern its structure, as depicted in Figure 2.1 [54]:

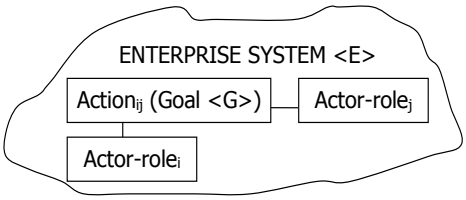


Fig. 2.1. Simplified view on an enterprise system.

As seen from the figure, within an enterprise system, one could identify *actions* whose realization relates to corresponding *actor-roles*.

In order to bring a deeper clarification regarding enterprise systems, we need to further elaborate on the notion of action (as mentioned already, we distinguish between *production actions* (or *acts*) and *communicative* (*coordination*) *actions* (or *acts*), and this also needs to be considered). We reflect this in the **`transaction`** concept [19] because of its capabilities to grasp those two aspects, namely production and coordination. Further, this concept is well aligned with the *actor-role* notion, assuming the possibility that not only a particular human entity could fulfill more than one actor-roles but that a particular actor-role could be fulfilled by more than one human entities. If nevertheless one particular actor-role is being fulfilled by one particular human entity, then the combination of the human entity and the actor-role is called **`actor`**. Hence, we consider the following definition [21,54]:

DEFINITION 5 A **`transaction`** is a finite *sequence of coordination acts* between two actors, concerning the same *production fact*. The actor who starts the transaction is called the *initiator*. The general objective of the initiator of a transaction is to have something done by the other actor, who therefore is called the *executor*.

Hence, *transactions* should be considered as the elementary building blocks of an enterprise system. As studied by Dietz [20], transactions are related to each other in a tree-structure. The top of the tree is called the *starting transaction* [57] - it is a transaction that is not caused directly by another transaction (from the particular tree) but triggers the execution of other transactions (within the tree).

Considering *transaction trees* (as a level of granularity) rather than transactions is more appropriate to be done in modeling enterprise systems because at the granularity level of transactions, the complexity is often rather big: even a simple enterprise system would contain a great number of transactions, making it difficult for modelers to grasp precisely and describe those transactions [54]. Thus, the consideration of transaction trees would help partitioning somehow the multitude of transactions, grouping them into segments. We hence introduce the **`business process`** concept in this regard [57]:

DEFINITION 6 A **business process** is a *structure of* (*connected*) *transactions* that are executed in order to fulfil a starting transaction.

Thus, in our view, the operation of enterprise systems concerns business processes (which are driven by the goal characterizing the system). Each business process consists of transactions, including a starting transaction – as exhibited in Figure 2.2. Transactions in turn relate to initiators and executors.

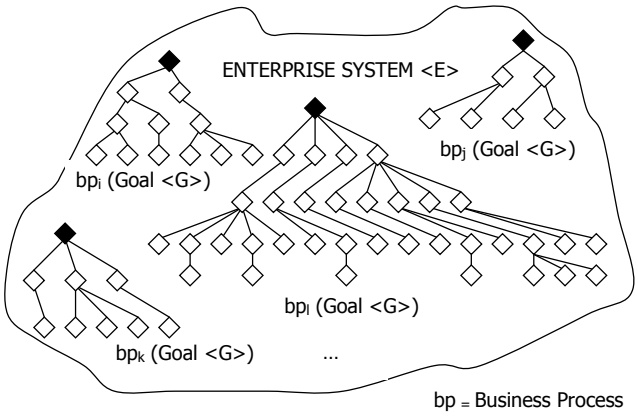


Fig. 2.2. Visualizing the operation of an enterprise system.

The figure shows a particular example of an enterprise system operation. It concerns many *business processes*; four of them are depicted in the figure, namely bp_i, bp_j, bp_k, and bp_l. As seen from the figure, each of the business processes (driven generally by the *goal* <G>) consists of transactions (with a starting transaction on top). The transactions are presented by white diamonds, the starting transactions are presented by black diamonds. A *starting transaction* could be activated in any of the following three ways: *outside cause* (activation from a customer), *periodic activation* (usually concerning payment activities), and *activation resulting from a waiting relation* (a transaction could start only after another one is completed) [22].

Summarizing so far: we have presented our viewing the operation of enterprise systems as concerning a number of business processes driven by a common general goal. We have also elaborated on our defining a business process.

A further consideration of enterprise systems should touch upon decomposition: firstly, because as it is well-known, decomposition reduces complexity in considering any system and secondly, because addressing particular parts of an enterprise system could allow for treating them separately and also for re-using them. Hence, we will consider the notion of **enterprise sub-system**, by putting forward the following definition [54]:

DEFINITION 7 An **enterprise sub-system** is a *system* which is a *part of an enterprise system*.

Based on *Definition 7*, it becomes clear that if W is the *set* containing all the *transactions* and *actors* included in an *enterprise system*, any *sub-set* $W^i \subseteq W$ which satisfies the system definition, would represent an *enterprise sub-system*.

Nevertheless, using the enterprise sub-system concept without any other restrictions makes little use because of the *non-determinism of the concept*: any combination of transactions and actors could be an enterprise sub-system. Hence, we argue that making use of the mentioned concept should assume the application of clear criteria when deciding what enterprise sub-systems to use and here the *re-use potential* is claimed to be of importance - this includes a clear *granularity positioning* of the enterprise sub-systems which one is to consider [54].

A possible and logical way of defining an enterprise sub-system is to consider corresponding *business processes*, because:

- the issues related to a particular business process are distinguishable from all other issues that belong to the corresponding enterprise system;
- business processes relate to a *useful granularity level* (between the *transaction level* and the *enterprise system level*).

Hence, we will consider such enterprise sub-systems that relate to a particular business processes. We will call such enterprise sub-systems **business components**, bringing forward the following definition [56]:

DEFINITION 8 A **business component** is an *enterprise sub-system* that *comprises exactly one business process.*

If more business processes are to be considered, for example three, then this would point to three corresponding business components. If it would then be necessary to bring two of them together (for example), this would mean just bringing together two business components, ending up in a *component of components*. This is certainly possible if: (i) the inter-relations concerning those components (two in our example) are well-defined; (ii) the relations with the environment are well-defined also, since this would not necessarily mean just 'putting together' the relations of one of the components with its environment and the relations of the other one with its environment – possible conflicts, redundancy, and so on should be avoided.

We have now introduced and clarified some basic *EIS*-relevant notions, paying special attention to the concept of *business component*. *Definition 8* positions this concept within the *enterprise engineering* area unlike other definitions according to which *business component* is a *software engineering* concept [1,4].

Still, the consideration of the notion of **component** vs the notion of **system** requires further discussion because in our view touching upon those issues is not only a matter of *granularity* but also a more general thing pointing to basic terminology currently used in *systems engineering*, *software engineering*, and so on. It would often be the case that our system of consideration is pointing to a particular enterprise but

this may also depend on the view point, as discussed already. *Business processes* are identified within the enterprise and on that basis we identify *business components*. Hence, it might be (although not necessarily) that an *enterprise system* is decomposed in terms of *business components* which are nevertheless not the *atomic* entities within the enterprise – *business components* could be **`decomposed`** themselves.

In *programming*, *components* are *decomposed* in terms of *objects* [1] but what is **`object`** in *enterprise engineering*? According to Dietz [19], an *object* is an *observable* and *identifiable* individual *thing*, for example a person or a car. Hence, we observe different ways of defining object in different disciplines – *software engineering* and *enterprise engineering* in this case. Since *EIS* relates to both of those disciplines, we need to go deeper in discussing that notion, such that we position it correctly among the other concepts we are considering in the current chapter. To do this, we note the word *observable* from the definition of Dietz and this bring us to *organizational semiotics* [43] where **`sign`** is defined as: *something that stands for something else in some respect or capacity*. *Organizational semiotics* brings useful value to *enterprise engineering*, by its theoretically relating the notions of *object* and *sign* through the so called *meaning triangle*, as depicted in Figure 2.3:

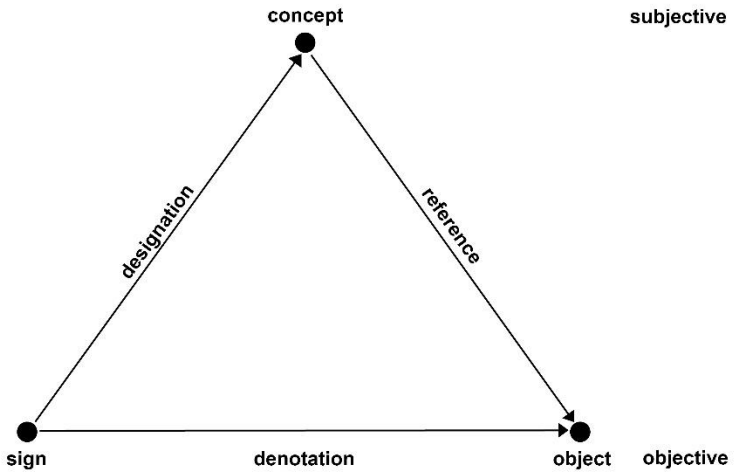


Fig. 2.3. The meaning triangle.

As the figure suggests, people use *signs* as *representations of objects* in order to be able to communicate about those objects and here the notion of *concept* is to be considered as well – this notion is **`subjective`** (unlike the notions of *object* and *sign* which are **`objective`**). Hence, a *sign* is an *object* that is used as a *representation of something else*. A well-known class of signs are the *symbolic signs*, as used in all natural languages, for example the name '*John Atkinson'* – we may write this name many times without the corresponding person named *John Atkinson* to be present, and we use this

sign in support of our communicating about the mentioned person. When it comes to the *object* '*John Atkinson*', this assumes our being *physically* able to perceive *John*, his face, and so on. This corresponds to the notion of *concrete object* - observable by human beings, unlike objects that are not observable by human beings, for example: *'number three*', called **`abstract objects`**. Further, the properties of an object collectively constitute the 'form' of the *object* [19]. *Objects* may be **`composite`**: *an aggregation of two or more objects* is also an *object*, for example: a car as a whole is an *object* but also the back seat of the car (or any other (*composite*) detail) is an *object* by itself.

What about *business components* and how does the notion of *business component* relate to the notion of *object*, as above presented? Let us take as an example a tourist enterprise, dealing with vacations' organization, accommodation bookings, flight bookings, and so on, and let us consider different *business processes* there, such as the accommodation booking business process and the flight booking business process. Hence, those two business processes would point to corresponding *business components*, namely Accommodation booking and Flight booking. As it is clearly seen from the example, we may consider those *business components* as:

- *abstract objects* since they are not observable by human beings;
- *composite objects* because we can go to *finer granularity*, for example, splitting the accommodation booking into the booking itself and the payment that goes as part of the booking.

Even though many examples one could think of point to *abstract composite objects*, it would not be justified claiming that all *business components* represent *abstract composite objects*. Still, being considered as an *object*, a *business component* represents a useful *enterprise modeling unit*, yet not the *atomic* modeling unit because, as discussed above, most *business components* could undergo further decomposition. This is logical because a *business component* points to a corresponding *business process* and the *business process* in turn represents a structure of *transactions*, as according to *Definition 6*. For this reason, we consider transactions as the *atomic enterprise modeling units*.

Still, at a *higher level* (with regard to elaboration), one could consider *business components* that give the right perspective for grasping the *enterprise* while at a *lower level*, where a more elaborated view is needed, considering *transactions* would be better.

Furthermore, when considering *actor-roles*, *transactions*, *business components*, and so on, it is necessary to establish what *governs* their (complex) *inter-relationships* and *behavior*. For this reason, we consider as well **`regulations`** in general, as important with regard to *behavior orchestration*, and in particular: (*aggregation*) **`rules`** that help introducing *behavior restrictions* [39]. We find *organizational semiotics* useful in this regard and particularly its *norm analysis method* reflected in the widely popular *rule (**norm**) pattern* [43]:

```
whenever <condition>
if <state>
then <agent>
is <deontic operator>
to <action>
```

We will not go discussing the *norm pattern* in more detail in this chapter – we only justify the need for *regulations* and *rules* in *analyzing* and/or modeling an *enterprise system*.

Finally, valid challenges in the context of what has been presented so far in the current chapter could hence be: (i) realizing an enterprise model that may help in better *understanding the enterprise under consideration* and/or *re-engineering* the enterprise, and/or *engineering a new enterprise*, and so on; (ii) delivering an enterprise model to be used as *basis for software specification*, that may help if automation is to be introduced within the enterprise, running software is to be updated, and so on. Thus, (ii) is especially relevant with regard to *EIS*. As studied by Shishkov [54], the **`enterprise-modeling-driven software specification`** is a complex task that could usefully be accomplished in a *component-based way*, such that *re-use*, *traceability*, and *evolvability* are possible.

Hence, an *enterprise-modeling-driven software specification* would assume using *business components* (and possibly *transactions* and corresponding *rules*) as *basis for specifying software*. This represents therefore a **`model-driven`** *enterprise-software alignment*, and elaborating on what we mean by **`model`** is necessary in this regard.

As considered by Shishkov [54] and Dietz [19], a *model* of *system A* is a *system* used to acquire knowledge about *system A*. Those views are consistent with the definition of Apostel [3], which we use:

DEFINITION 9 Any subject using a *system A* that is neither directly or indirectly interacting with a *system B,* to obtain information about the *system B* is using *A* as a **`model`** for *B.*

Moreover, realizing that a *model* of anything gives usually a 'partial picture', we need to define what should be considered as a *complete model*, and for this we firstly consider the notions *composition* and *structure* of an (*enterprise*) *system*, which notions are essential. They both concern two things, one of them is: how the entities belonging to the system are positioned among each other and the other one is: what are the (business) processes realized accordingly; the former is referred to as *structure* and the latter is referred to as *behavior* (or *dynamics*). We secondly consider *data* because any *system* (possibly an *enterprise system*, an *EIS*, or any other one), holds the need for storing, processing, and communicating data (it is always that things are counted, (statistical) data analysis is applied), and so on, no matter if this concerns biology, politics, or enterprises, to give just three examples of system domains. On that basis, we define **`complete model`** as follows:

DEFINITION 10 A **`complete model`** is a *model* that is elaborated at least in three perspectives, namely *structural perspective, dynamic perspective*, and *data perspective.*

We will also present (below) the **`business coMponent`** concept denoting a *complete model* of a *business component* where the word '*coMponent*' is with a capital '*M*' to indicate the relation to the word '*model*' [54]:

DEFINITION 11 A **business coMponent** is a *complete model* of a *business component*.

Hence, if we know the *structure* of an enterprise unit, the *processes* over this structure, and the related *data flows*, we claim to have a somehow '*complete*' perception of the enterprise unit, but is this always the case? What about situations in which complicated human-to-human communication goes beyond the mere business processes and data flows? We may consider two examples: (i) a holder of a debit card tries several times unsuccessfully to withdraw money from a cash machine, entering wrong personal identification number => we observe a business process and data flows but nothing actually happens between the bank and its customer; (ii) as a result of a simple conversation between a pizza restaurant waiter and a customer, a commitment appears for delivering a pizza to the customer => even though this is just a simple conversation, it brings in an obligation that has actual business sense. Thus, *EIS* as *systems* consisting of *human entities*, *technical entities*, and so on, are often characterized by **human-to-human communications** and those are to be considered as part of the *enterprise modeling* since such *communications* bring in *promises*, *commitments*, *negotiations*, and so on, and those issues may have impact on particular *business processes* and corresponding *enterprise* (*information*) *systems*. For this reason, in [54] this has especially been labelled as **communication perspective**. We have not considered such a perspective explicitly because according to *Definition 6*, *business processes* are considered as structures of *transactions* and transactions in turn are not only about the *production acts* but also about the *communicative* (*coordination*) *acts* - we believe that this already gives good reference to *human-to-human communication* and represents a guarantee that when considering *business processes* from a *dynamic* perspective, such *communications* would be adequately reflected.

And in the end, in Figure 2.4, we outline our view on how to use those concepts for *modeling*.

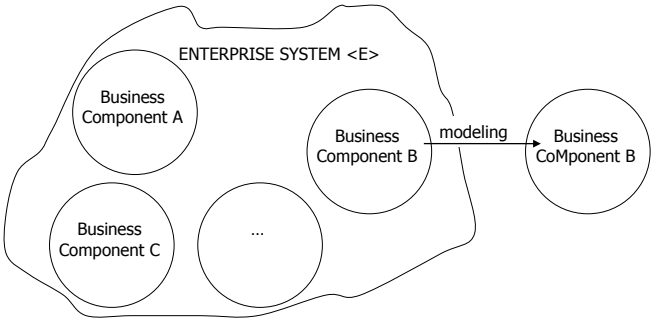


Fig. 2.4. The component-coMponent relation.

As seen from the figure, we view an *enterprise system* as composed of *business components*. We could represent such *components* in terms of *business coMponents* via *modeling*. Those *coMponents* could be used either as *enterprise modeling units* or as input for further *software specification* tasks.

2.3 Enterprise Information Systems

As mentioned at the beginning of the current chapter, after discussing *systems* and *enterprise systems*, we are addressing (in this section) particularly *EIS*, noting nevertheless that: (i) *enterprise systems are a well-known class of systems*; (ii) *enterprise information systems are a class of enterprise systems*. Thus, all characteristics of *systems* and *enterprise systems*, as discussed already, are to conform to *EIS* as well. For this reason, we will only focus on the distinctive features of *EIS* in this regard. Further, we make the assumption that ICT in general and **`ICT applications`**, in particular represent important part of any *EIS* – by '*ICT application*' we mean a *software application* that is nevertheless operating in a distributed networked environment and thus benefitting from current mobile and cloud technologies [12]. Still, no matter if we consider a *software application* or (more broadly) an *ICT application*, the **`software specification`** task is claimed to play a crucial role [54]. For this reason, we outline two important challenges, namely:

- the *software specification* task and its role in the creation of *EIS*;
- the *relation* between *business coMponents* and *software specification*.

Further, being an *enterprise system* itself, an *EIS* has the following properties:

- ✓ its compositional elements are *human entities*;
- ✓ human entities fulfill particular *actor-roles* in realizing activities within the EIS;
- ✓ the *EIS* structure concerns *inter-role relations* which are in turn driven by *goals*.

However, with regard to *enterprise systems*, the *goal* is the *delivery of business products and/or services* to entities belonging to the *system environment*, while with regard to *EIS*, the *goal* is the **`informational support to a corresponding enterprise system`**. As for *environments*: the *environment* of an *enterprise system* consists of *actor-roles* (those *actor-roles* may be fulfilled by human entities but they may also be fulfilled by technical entities) and *actions*, and those are external with regard to the enterprise of consideration; the *actor-roles* and *actions* that belong to the *environment* of an *EIS*, in contrast, are usually internal with regard to the *enterprise* of consideration, and the reason for this is the role of an *EIS* as *supporting* a corresponding *enterprise system* [54].

Thus, an *enterprise system* exploits an *EIS*, benefitting from corresponding *EIS services*. Said otherwise, an *EIS* supports a corresponding *enterprise system*, by providing *services* to it.

As mentioned already, such kind of support is usually realized by means of *ICT applications* which allow *enterprise systems* to utilize current possibilities that are related to ICT. With regard to this, we consider the following definition for '*ICT*

application', adapted from [54], that is consistent with the definitions and assumptions put forward in the current chapter:

DEFINITION 12 An **`ICT application`** is an *implemented software product* realizing a particular *functionality* for the benefit of entities that are part of the composition of an *enterprise system* and/or a (corresponding) *EIS*.

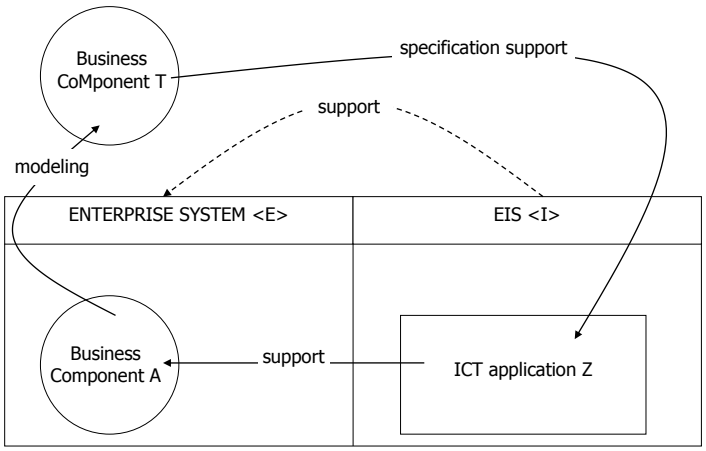


Fig. 2.5. Business coMponents' supporting the applications' specification.

Hence, *ICT applications* are largely instrumental with regard to the way in which *enterprise systems* are supported informationally, and in many cases, this is about the: (i) *automation of business processes belonging to an enterprise system* – for example, part of what human insurance brokers are doing is being automated, such that this same work is realized in an automated way, by means of software; (ii) *enrichment of existing business processes for the sake of utilizing new technological possibilities* – for example, moving storage to the Cloud would assume additional efforts on coping with information security, possible latency, and so on, to mention just two possible implications in such a context. Therefore, an ICT application is to be 'covering' either a whole enterprise system (this is obviously rare because as above suggested, the delivered ICT support is most often focused on a particular issue(s) within the enterprise under consideration) or part(s) of it corresponding to particular business processes – this makes *ICT applications straightforwardly aligned to business components* or components consisting of business components. Since this is matter of *granularity*, we would not distinguish between the cases when an ICT application points to one particular business component and the cases when an ICT application points to a group of (several) inter-related business components (which we called *component of business components*) – we will speak of an *ICT application* pointing to

a *business component* and mean both. Such a relationship should (logically) assume that the business component would have to be precisely reflected in the **`specification`** of the corresponding *ICT application*; otherwise, the *ICT application* support would be inconsistent with regard to the *enterprise* context. For this reason, we propose using *business coMponents* as source for the derivation of *ICT application's specification*; this is depicted in Figure 2.5.

As seen from the figure, the support (indicated by the dashed line) that an *EIS* realizes to an *enterprise system* is facilitated (actually driven) by *ICT applications*. As also seen from the figure, a *business coMponent* might support the *specification* of a corresponding *ICT application*. Hence, of particular interest are the relations:

business component – business coMponent – ICT application.

Said otherwise, we are interested to know how a (re-usable) *business coMponent* could be identified and also how it could be reflected in the *specification* of an *ICT application*.

Thus, we would need to discuss the role of *specification* in the design and development of an *ICT application* and also possibilities for decomposing the specification model.

We will hence firstly position the *specification task*, considering the *three-phase software creation process*, following Atkinson and Muthig [5]:

- *specification*, addressing the functionality of the software artefact-to-be;
- *realization*, addressing the specification's (further) refinement and also technological aspects;
- *implementation*, addressing the model-based coding bringing about the final software application output.

Hence, the *modeling support* that is provided by a *business coMponent* affects the *specification* phase as depicted in Figure 2.6 [54]:

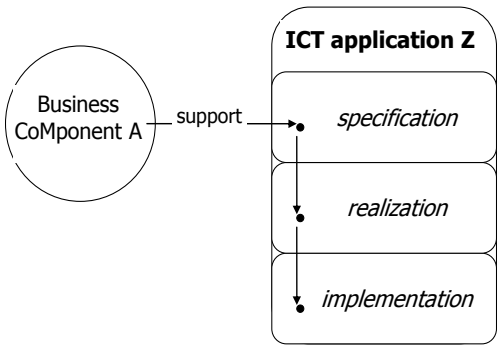


Fig. 2.6. A support of a business coMponent to the specification of an ICT application.

As far as *ICT applications* are concerned, we take also into account the current software development standards, as discussed at recent editions of the international

symposium on Business Modeling and Software Design – BMSD [8] according to which the **component-based software specification and development** are largely recognized.

Usually, within the software community, the term *software component* is associated with the *component-based development of ICT applications*, which is characterized by assembling re-usable software components [54]. They represent prefabricated, configurable, and independently evolving building blocks which provide some functionality that can be used separately or in composition with the functionality provided by other software components.

According to the *middleware* perspective [31], which does not necessarily envision a software component in the context of the development of an ICT application, software components are blocks of code ready to be deployed on top of a suitable execution environment (often called *container*) which provides a number of generic services for the execution of components, such as *event notification*, *authentication*, and so on.

We hence conclude about several essential characteristics of *software components*, also referring to MDA (see Chapter 4) [47], relevant to the *software engineering* domain:

- ✓ any *software component* is characterized by a particular *functionality* and is driven by the *goal of providing service(s) to its environment*;
- ✓ in its providing service(s), a *software component* could *collaborate* with other *software components*;
- ✓ the *environment* of a *software component* may consist of other *software components*, *ICT applications*, supporting platforms, and so on.

Hence, in addressing *software components*, we consider it necessary paying attention to the *interface specification*, *component dependencies*, *deployment*, and *granularity* – those issues are briefly discussed below. This is in tune with related studies reported in [31].

An *interface specification* can be seen as a contract which is established between a software component providing (implementing) a service and the component's environment using (invoking) it.

The *component dependencies* comprise the events that can be either produced or consumed by a software component, in its providing service(s).

Given its binary representation, a software component is a self-contained building block which could be *independently deployed* in a variety of environments.

Noting *composability*, a software component should not necessarily be a complete ICT application; it may be a part of the whole. It is well known, nevertheless, that there are examples of large software components that could be envisioned either as components or as applications. Thus, considering the *granularity* of a software component under development is of significant importance. In our view, in specifying the size of a software component, the modeler should take into account the fundamental requirement that a software component should be general enough to be re-usable in a number of ICT applications [55].

Hence, on the basis of the above analysis, we consider a relevant **software component** *ontological definition* [57]:

DEFINITION 13 **Software components** are *implemented pieces of software*, which represent *parts of an ICT application*, and which *collaborate among each other* driven by the *goal* of *realizing the functionality of the application*.

Since the *software component* concept concerns the implementation phase, we would need to propose also a *functional definition*, inspired by Szyperski [73]:

DEFINITION 14 A **software component** is *functionally* a *part of an ICT application*, which is *self-contained*, *customizable*, and *composable*, possessing a *clearly defined function and interfaces* to the other parts of the application, and which can also *be deployed independently*.

Thus, by creating an instance of a *software component*, we do actually deploy it. We could view, therefore, such a component instance as an *object*. However, there is little agreement on the differences between *software components* and *objects* [31]. For this reason, we will not enter this discussion within the current chapter.

Since any support from a *business coMponent* would concern the *specification* phase, we should consider another relevant concept referring to the logical building blocks of an ICT application (in contrast to software components representing the physical application building blocks, in the sense of *physical* component technologies, such as CORBA [11], **.**NET [4], EJB [26], and so on). We hence introduce the term **software coMponent** to reflect the above mentioned *logical* aspects:

DEFINITION 15 A ***software coMponent*** is a *conceptual specification model* of a *software component*.

Summarizing our views and referring to Shishkov [54]:

- an *enterprise system* consists of *business components*;
- an *ICT application* consists of *software components*;
- the creation of a *software component* is supported conceptually by a corresponding *software coMponent*;
- the identification of the *software coMponents* is supported conceptually by a corresponding *business coMponent*.

Figure 2.7 illustrates this:

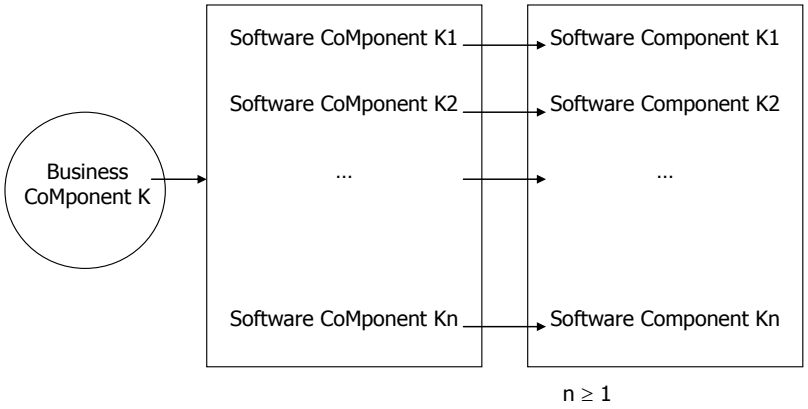


Fig. 2.7. Business coMponents, software coMponents, and software components.

As seen from the figure and as already stated, a *business coMponent* supports conceptually the identification of at least one *software coMponent*. A *software coMponent* in turn supports conceptually the creation of a corresponding *software component*.

We hence claim that the concepts introduced so far in the current chapter, allow for deriving a (component-based) *software specification model* on the basis of a corresponding *enterprise model*, realizing in this way a (component-based) *business-IT alignment*.

Construction is crosscutting with regard to all this – by *construction* we mean the ontological dependencies and relations among system elements, relevant to the question *How is the system realizing its functionality?* (as opposed to the question *What is the system realizing as functionality?*), and this will be considered in the following section. Still, it is to be noted that we have been consistent with such an ontological perspective in this chapter so far – what we will only do in the following section is to consider those issues more explicitly.

2.4 Ontological Systems and Function

Referring to the notions addressed in the previous sections, we consider a *system* and its *environment*, and we may like to also be explicit about the **`system boundary`** – the *system boundary* separates the *system* from its *environment*. Let us then consider together the *system*, the *system boundary*, and the *system environment*, calling this collectively **`Universe of Discourse`** or *UoD*, for short. Then, according to Dietz [19]: the *system composition*, the *system*, the *environment*, and the *structure* (spanning over them) are collectively called the *UoD construction*. The *UoD construction* can thus be described by enumerating the entities within the *system*, the entities of the

environment, as well as the relationships in the *structure* – this is illustrated in Figure 2.8:

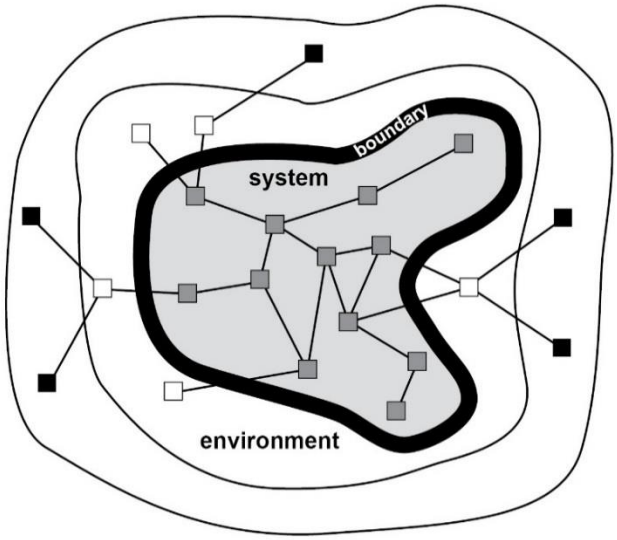


Fig. 2.8. The UoD construction.

On the figure: the *composition* of the *system* consists of the gray-colored elements; the *environment* consists of the white-colored elements; as for the black-colored elements – since they do not have influencing bonds with elements of the system, they are considered *UoD*-external; the black line separating the *system* elements and the *environment* elements represents the *boundary*; the lines represent the structural bonds between elements. Thus, only the bonds among the system-internal elements and the bonds between system elements and environment elements belong to the *UoD structure*. Finally, the *UoD composition*, together with the *UoD* structural bonds is called the *UoD kernel*.

An identical but more precise formal definition of the **`UoD construction`**, following Bunge [10], is presented below, using two special symbols, namely: (i) $\prec$ meaning *is part of* and (ii) $\triangleright$ meaning *acts upon*, and particularly: $\mathbf{x}$ acts upon $\mathbf{y}$ if and only if $\mathbf{x}$ influences the *behavior* of $\mathbf{y}$; if both $\mathbf{x} \triangleright \mathbf{y}$ and $\mathbf{y} \triangleright \mathbf{x}$ hold, we say that $\mathbf{x}$ and $\mathbf{y}$ *interact*:

Let $\boldsymbol{\sigma}$ represents our considered *UoD* and $\boldsymbol{\Gamma}$ a class of things, called the *category* of $\boldsymbol{\sigma}$. Then, the *composition* $\mathbf{C}$ of $\boldsymbol{\sigma}$ is defined as:

$$C(\sigma) = \{x \in \Gamma \mid x \prec \sigma\},$$

the *environment* **E** of **σ** is defined as:

$$E(\sigma) = \{x \in \Gamma \mid x \notin C(\sigma) \wedge \exists y : y \in C(\sigma) \wedge (x \triangleright y \vee y \triangleright x)\}$$

and the *structure* **S** of **σ** is defined as:

$$S(\sigma) = \{< x, y > \mid (x \triangleright y \vee y \triangleright x) \wedge (x, y \in C(\sigma) \vee (x \in C(\sigma) \wedge y \in E(\sigma)))\}$$

As for the notion of *sub-system* that has been already considered in this chapter, we are now re-visiting this notion, providing below a precise definition from an *ontological perspective* [19].

Let there be a *system* σ_1 with the *construction*:
$<C(\sigma_1), E(\sigma_1), S(\sigma_1)>$
and a *system* $\sigma_2$with the *construction*:
$<C(\sigma_2), E(\sigma_2), S(\sigma_2)>$.

Then *system* σ_2 is a *sub-system* of *system* σ_1 if and only if

$$C(\sigma_2) \subseteq C(\sigma_1)$$
$$E(\sigma_2) \subseteq (C(\sigma_1) \setminus C(\sigma_2) \cup E(\sigma_1))$$
$$S(\sigma_2) \subseteq S(\sigma_1)$$

Further, with regard to a *UoD*, the collective activity of the *system* elements and the *environment* elements is called **operation**. Even though this concerns not only the *system* but the whole *UoD*, the *operation* is essentially initiated and driven by the *system* (and possible contribution from elements that belong to the *system environment* is triggered by *system* elements). For this reason, we may say that the *operation* of a *system* is the manifestation of its *construction* in the course of time – this encompasses both the *production actions* and the related *coordination actions*, preformed accordingly [19].

And in the end, heterogeneous systems (for example, a car where one could identify: (i) a mechanical system; (ii) an electrical system, and so on) are more complex than homogenous systems (just the mechanical system, for instance, if we take the above example), and the above definitions and discussion apply straightforwardly to *homogenous systems*. When one would address a *heterogeneous system*, nonetheless,

one would have to reflect such a system in *a number of homogenous systems which are related to each other in a layered nesting* [10]. The way in which a collection of *homogenous systems* constitutes a *heterogeneous system* is not trivial and this holds particularly for *enterprises* since *enterprise systems* are *heterogeneous systems* [19].

However, we will not go deeper in this discussion in the current chapter. Instead, we will touch upon another important perspective over a system, namely the *functional perspective* (as opposed to the *constructional perspective* considered above). Below, we will explicitly discuss each of those two perspectives and will emphasize on the distinction between them.

2.4.1 Construction vs Function

When modeling a *system*, one could take a **`white-box`** perspective that is closest to the *ontological* view considered above – this is about capturing the *construction* and the *operation* of the *system*, while abstracting from implementation details which are assumed to be irrelevant; the *white-box model* is hence adequate for *building* or *changing* a *system*. Contrary to this, taking a **`black-box`** perspective is about capturing the *interactions between the system composition and the environment* – this conveys the *functional* perspective on a *system* and a *black-box model* hence has no direct relation with the *construction* and *operation* of the *system* under consideration [19].

In order to illustrate this, we consider for example a car, and we take a *white-box view* over the car as well as a *black-box view*, as shown in Figure 2.9.

As seen from the figure, the white-box view is close to the mechanic's perspective – the mechanic being interested in HOW the components of the engine, the components of the suspension, the components of the electric system, and so on work (each one and in combination among each other), such that the desired performance is realized. In contrast, the black-box view is close to the driver's perspective – the driver being interested WHAT the car can do for him/her in terms of *an input triggering corresponding output* – whether or not pressing the inside lamp button would lead to illumination inside the coupe, whether or not turning on the car key would lead to noise from the engine, whether or not pressing the brake pedal (while the car is moving) would stop the car, and so on.

Hence, taking a white-box perspective would lead to a constructional decomposition into engine, wheels, exhaust, and so on, while taking a black-box perspective would lead to a functional decomposition into the power system, the brake system, the audio system, and so on, as suggested by the figure.

After having discussed the *construction* and *function* of a *system*, we will turn to another important issue concerning *systems*, namely *evolvability*. In the following section, we will consider *combinatorial effects*, as strongly relevant to the mentioned concern, addressing this from the perspective of the *Normalized Systems Theory*.

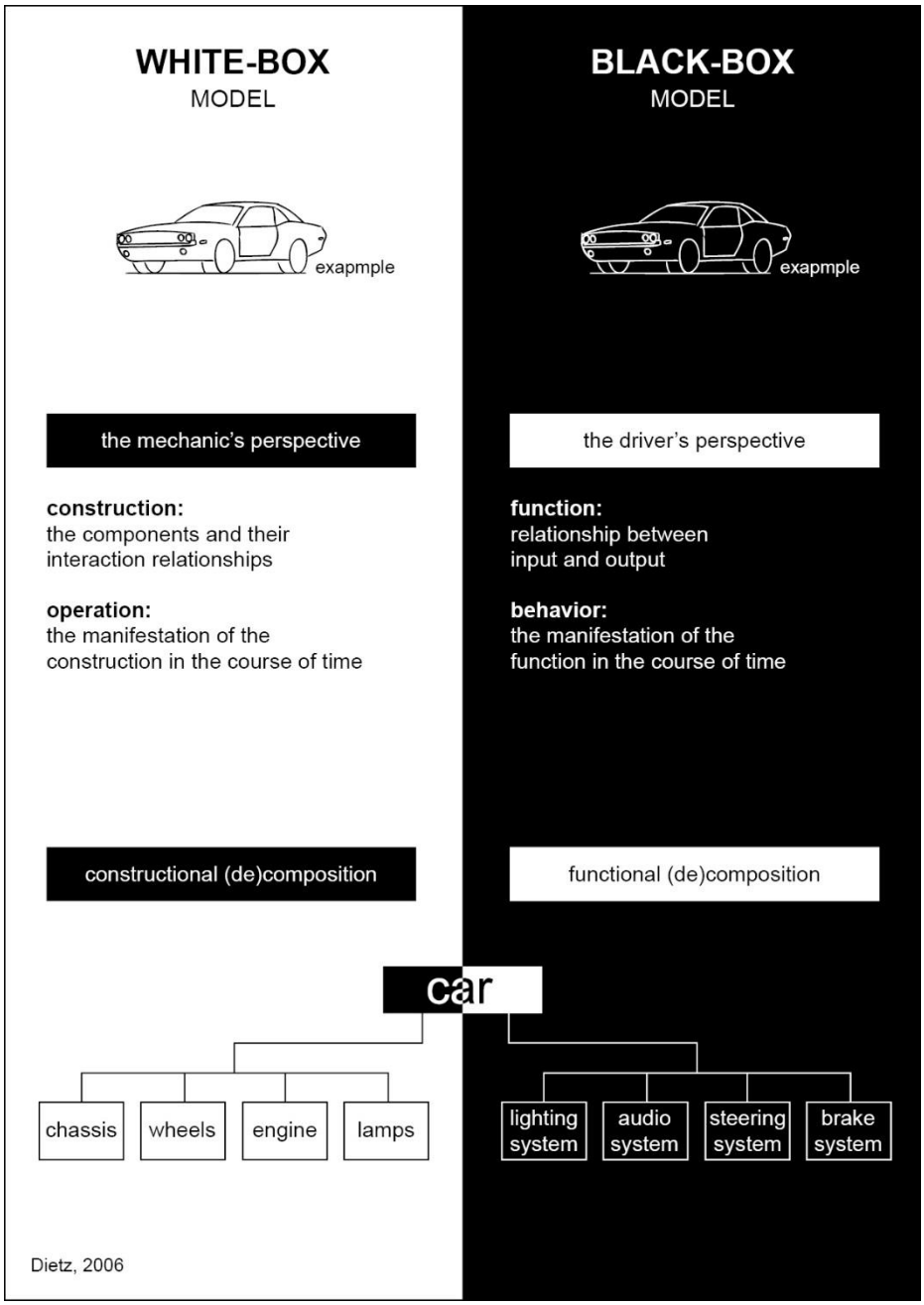


Fig. 2.9. White-box view vs black-box view

2.5 Normalized Systems

We consider the *Normalized Systems Theory*, referring to [35], acknowledging that *EIS* should be able to *evolve* over time; said otherwise, an *EIS* should be designed in such a way that it is *capable of accommodating change*. Hence, such kind of evolution concerns the maintenance of the software 'part' of an *EIS*. *Software maintenance* is not only expensive but it also leads to: (i) increased architectural complexity; (ii) decreased software quality [25]. This is also recognized by *Lehman's Law of Increasing Complexity*, indicating for a degradation of the structure of an EIS over time [40]. Thus, *the impact of a single change will increase over time*.

In order to avoid such quality degradation, it is suggested aiming at *theoretic stability* [44], referring to the fact that bounded input to a function results in bounded output values, even as $t \rightarrow \infty$. This means that a specific change to an *EIS* should require the same effort, irrespective of the size of the *EIS* or the point in time when being applied. Each change that is applied to an *EIS* requires a certain amount of effort. This effort can be measured in, for example, the amount of time or the lines of code needed to apply the change. This effort would nevertheless increase if on top of this intrinsic amount of effort, additional software components need to be adapted. How such effort increases over time is illustrated in Figure 2.10:

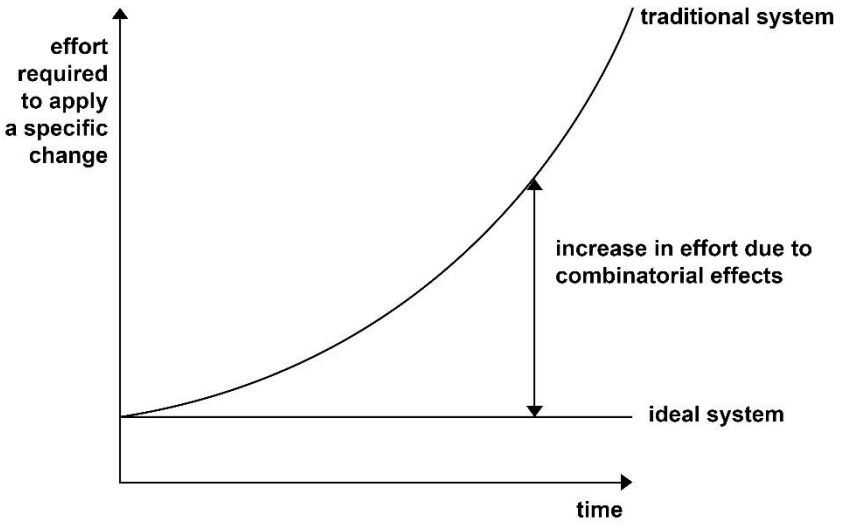


Fig. 2.10. Impact of combinatorial effects [45].

As it is seen from the figure, when an 'ideal' system is considered, the effort required to apply a specific change does not increase over time. However, this effort would actually increase over time (as mentioned above and as according to *Lehman's Law*) in a 'real' system, leading to deteriorating effects (over time) resulting from the applied changes - this is represented by the traditional system curve. Hence, there is a distance between the two curves, increasing over time. It is **`combinatorial effects`** that contribute to this distance [45].

Combinatorial effects occur when the impact of a change is dependent on the size of the *EIS* and avoiding *combinatorial effects* would lead to avoiding the software quality deterioration as explained already. The identification of such *combinatorial effects* assumes that software is considered as a *modular structure*.

Huysmans [35] considers the *inter-module EIS dependencies* as causing *combinatorial effects*, claiming that in such cases, realizing a change in a specific module would lead to *impact on other modules* that are (in principle) unrelated to the original change. Such dependencies can be *introduced at design time* while the vision on stability requires that *not a single dependency is introduced*, even when an unlimited amount of modules would be added - this is called *the assumption of unlimited systems evolution* [46]: thus, *only when no combinatorial effects occur while the EIS grows, it is considered to be evolvable*.

An *EIS* would be considerable as a **`normalized system`**, if exhibiting *stability* with respect to a defined set of changes and the *Normalized Systems Theory* deduces a set of *four design theorems* that act as *design rules* to identify and circumvent most *combinatorial effects* [46], claiming that any failure to adhere to one of those theorems would result in the introduction of *combinatorial effects*.

With regard to this, considering *modular structures*, taking a basic view, assumes the consideration of *action modules* and *data modules* only, called '*entities*'. Hence, our simplified view assumes considering *action entities* which perform certain operations on *data entities* (*action entities* also receive input in the form of *data entities*). A *data entity* thus contains *attributes* - concrete values or links to other *data entities*. An *action entity* in turn represents an *operation* at a given modular level and this would concern (several) tasks - a *task* is a *set of instructions* performing a certain functionality. Such a conceptualization is consistent with *Definition 10* where *structure*, *behavior*, and *data* are considered essential with regard to an *EIS*.

The first theorem, *separation of concerns*, implies that every change driver or *concern* should be separated from other *concerns*. The theorem allows for the isolation of the impact of each change driver; this means that each *module* can contain only one sub-modular *task* (which is defined as a change driver), but also that workflow should be separated from functional sub-modular *tasks*.

The second theorem, *data version transparency*, implies that data should be communicated in *version-transparent* ways between components. This requires that introducing the *data* change (for example, sending additional data between two components) should take place without having an impact on the components and their interfaces.

The third theorem, *action version transparency*, implies that a component can be upgraded without impacting the calling component(s).

The fourth theorem, *separation of states*, implies that actions or steps in a workflow should be separated from each other in time, by keeping *state* after every action or step. This suggests an *asynchronous and stateful way of calling other components*.

Hence, those theorems show at which point in the modular structure of an *EIS, combinatorial effects* occur and that the only *modular structures* free from *combinatorial effects* are the fine-grained structures. Especially the principles of *separation of concerns* and *separation of state* indicate that *modules* have to be *separated both functionally and in time*.

We are not going in more detail on discussing those four theorems and we are also not elaborating further on the *Normalized Systems Theory* because our goal in this section is to only consider the impact of *combinatorial effect* with regard to the *evolution* of an *EIS*.

In the following section, we will shift focus from the *system* to the *environment* – considering the challenge of adapting the behavior of an EIS to the surrounding *context*.

2.6 Context-Awareness

Let us consider again the constructional UoD view presented in Figure 2.8 and elaborate the view on the *environment*, such that those environment entities using the system products and/or services (called **`users`**), are made more explicit, as shown in Figure 2.11. What is depicted on the figure is:

- a *system* comprising entities and corresponding relationships;
- an *environment* comprising other entities and their corresponding relationships;
- a *boundary* separating the two (the *system* and its *environment*);
- a *user* comprising some entities belonging to the *environment* (in the figure they are, for example, two) and their corresponding relationships;
- the broader *universe* where the UoD (the *system* and its *environment*) belongs.

Following such a view nevertheless, one may establish that there are 'limits' of the *environment*, that is certainly not true because we engineer *enterprises* / *EIS* and in doing that, we are certainly limiting the *system*, establishing what is to belong to the *system* but we are not engineering the *environment* and thus we are not in a position to say what belongs to the *environment* and what does not belong to the *environment*. It is, for this reason, more straightforward to consider as *environment* anything that is outside the *system*. Still, this would be an obstacle to distinguish between those entities (outside the *system*) that are somehow interacting with the *system* and those entities (outside the *system*) that are not interacting with the *system*. Said otherwise, we position as belonging to the *environment* anything that is not only *system*-external but is also concerned with interaction(s) with the *system*, and this goes beyond our control – the designer cannot establish who and how may happen to be interacting with the *system*. For this reason, although necessary, the separation between what belongs and what does not belong to the *system environment* remains abstract.

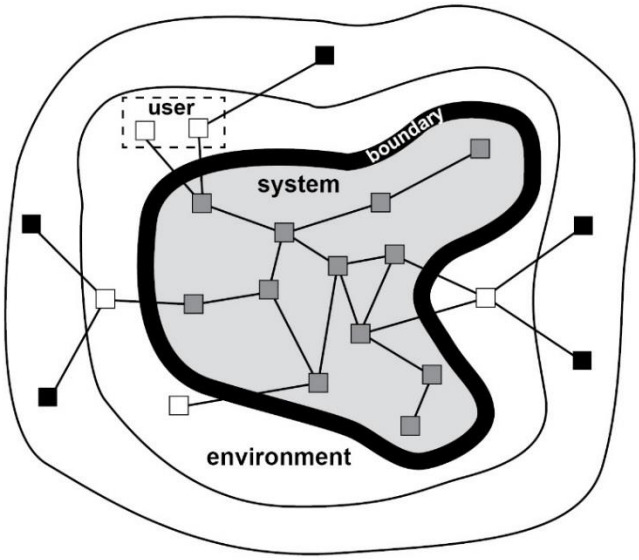


Fig. 2.11. Modeling the user as part of the environment.

As the figure suggests, there is always a *user* – no matter what a system delivers, it is delivered to a *user* (otherwise, the functioning of the system would be unjustified). We claim that the *user* is to be part of the *environment* because otherwise, it would mean that the *user* of what is delivered cannot be separated from the deliverer. Further, the system *user* may comprise one or more entities belonging to the system *environment* – each of them (or they both (if they are two, for example)) could consume different *services* (or one *service* together). Finally, not all entities belonging to the system *environment* should necessarily be parts of the *user* since it might be that the *system* needs to collaborate with other entities from the *environment* (different from the *user*), such that it is capable of delivering the requested products and/or services to the *user*.

Hence, a *user perspective* is needed in order to capture such a delivery of products and/or services (we call this *service*, for short). Further, it is often that the *service* delivered to the *user* is to be adapted to the situation of the *user*. For example, a person wearing a body-area network [6] through which body vital signs are captured, may appear to be at 'normal state' and then, for example, vital signs are captured and recorded as archival information, or the person may appear to be in an 'emergency state' and then help would need to be urgently arranged. Thus, one kind of *service* would be needed at *normal state* and another kind of *service* would be needed at *emergency state*. For this reason, the *system* (or a corresponding system-internal EIS or ICT *application*) should be able to: (i) *identify the situation of the user*; (ii) *deliver a service to the user, which is suited for the particular situation*. This is illustrated in Figure 2.12:

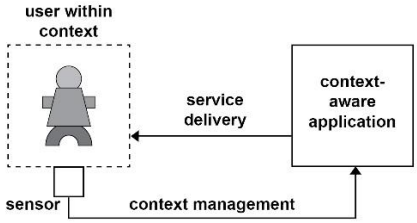


Fig. 2.12. Schematic representation of a context-aware application.

As it is seen from the figure a *service* is delivered to the *user* and the *user* is considered within his or her **`context`**, such that the *service* is adapted on the bases of the *context state* (or *situation*) the *user* finds himself / herself in. That state is to be somehow *sensed* and often technical devices, such as sensors, are used for this purpose. In the current chapter, we do not go into discussing *sensor technology* in detail and for this reason, by **`sensor`** we broadly mean the technical or other facility that helps establishing the *user situation*. As mentioned before, it might be an *EIS* delivering the *service* to the *user* or it might be that just one ICT *application* (for example) as part of the *EIS* is delivering the *service* – no matter whether the former or the latter, we call it **`context-aware application`** in the current section. Hence, a *context-aware application* adapts its behavior, in delivering *service*(*s*) to the *user*, based on the actual *context state* of the *user*, which *context state* is captured by sensors and corresponding information – sent to the *context-aware application* accordingly.

In the remaining of the current section, we will firstly consider context-aware applications and secondly – the *analysis of context situations* (*states*) related to this.

2.6.1 Context-aware Applications

Traditional ICT application development methods do not consider the *context* of individual users of the ICT application (or *application*, for short) under development, assuming instead that *end-users* would have common requirements independent of their context. This may be a valid assumption for applications running on and accessed at desktop computers, but would be less appropriate for applications whose *services are delivered via mobile devices*. Ignoring the dynamic *context* of users may lead to sub-optimal applications, at least for a subset of the *context situations* the *end user* may find himself / herself in. Hence, *context-aware applications* (mentioned already) have emerged, driven by the successful uptake of *mobile telephony* and *wireless telecommunications* [12]. Such *applications* are, to a greater or lesser extent, *aware of the end-user context situation* (for example, *user is at home*, *user is traveling*) and provide the desirable services corresponding to the situation at hand [61]. This quality points also to another related characteristic, namely that *context-aware applications* must be able to capture or be informed about information on the *context* of *end-users*, preferably *without effort and conscious acts from the user part* [62].

Developing *context-aware applications* is hence not a trivial task and as above suggested, the following related *challenges* have been identified: (i) Properly deciding what *physical context* to *sense* and what *high-level context information* to pass to an application, and also *bridging the gap between raw context data and high-level context information*; (ii) Deciding which potential *end-user context situations* to *consider* and which ones to *ignore*; (iii) Modeling *context-aware application behavior* including *switching* between *alternative behaviors* [61].

The basic *assumption* underlying the development of context-aware applications is that *end-user needs are not static*, however partially dependent on the particular *situation* the *end-user* finds himself / herself in, as already mentioned. For example, depending on his / her current *location*, *time*, *activity*, *social environment*, *environmental properties*, or *physiological properties*, the *end-user* may have different *interests*, *preferences*, or *needs* with respect to the *services* that can be provided by *applications*.

Context-aware applications are thus primarily motivated by their potential to increase user-perceived effectiveness, i.e. *to provide services that better suit the needs of the end-user*, by taking account of the *user situation*. We refer to the collection of parameters that determine the situation of an *end-user*, and which are relevant for the application in pursue of user-perceived effectiveness, as *end-user context*, or *context* for short, in accordance to definitions found in literature [18].

Context-awareness implies that information on the *end-user context* must be captured, and preferably so without conscious or active involvement of the *end-user*. Although in principle the *end-user* could also provide *context information* by directly interacting with the *application*, one can assume that in practice this would be too cumbersome if not impossible; it would require deep expertise to know the relevant *context parameters* and how those are correctly defined, and furthermore be very time consuming and error-prone to provide the parameter specifications as manual input [61].

Context-aware applications can be particularly effective if the *end-user* is *mobile* and uses a personal handheld device for the delivery of services. The mobile case is characterized by *dynamic context situations* often dominated by changing location (however not necessarily restricted to this). Different locations may imply different social environments and different network access options, which offer opportunities for the provision of *adaptive or value-added services* based on *context sensitivity*. Especially in the mobile case, context changes are continuous, and a *context-aware application* may exploit this by providing *near real-time context-based adaptation* during a *service* delivery session with its *end-user*. The adaptation is *near real-time* because *context information is an approximation* (not exact representation) of the real-life *context* and thus there may be a time delay [12].

Hence, through *context-awareness*, *applications* can be pro-active with respect to *service* delivery, in addition to being just re-active, by detecting certain *context situations* that require or invite the delivery of useful services which are then initiated by the application instead of by a *user request*. Otherwise said, traditional *applications* provide *service* in reaction to *user requests* (re-active), whereas *context-aware applications* have also the possibility of initiating a *service* when a particular *context situation* is detected, without *user input* (pro-active).

In summary, *context-awareness* concerns the possibility of delivering effective personalized *services* to the *end-user*, taking into account his / her particular *situation* or *context state*. Technological advances enable better and richer *context-awareness*, beyond mere *location-sensitivity*.

With regard to the design implications concerning *context-aware applications*, those *applications* require knowledge on *context* and exploit this knowledge to provide the best possible *service*, as mentioned above. This concerns the *end-user context*, i.e. the *situation of a person who is the potential user of services offered by an application*. Examples of end-user context are the *location of the user*, the *user's activity*, the *availability of the user*, and the *user's access to certain devices or facilities*. The *assumption* we make is that the *end-user is in different context situations over time*, and as a consequence, (s)he has *changing preferences or needs with regard to services*.

This corresponds to what is exhibited in Figure 2.12: the *application* is *informed by sensors* of the *context* (or of *context changes*), where the *sensing* is done as *unobtrusively* (and *invisibly*) for the *end-user* as possible. *Sensors sample the user's environment and produce (primitive) context information*, which is an *approximation* of the *actual context*, suitable for *computer interpretation* and processing. *Higher-level context information* may be derived through *inference* and *aggregation* (using input from *multiple sensors*) before it is presented to *applications* which in turn can decide on the current *context* of the *end-user* and the corresponding *service(s)* that must be offered. Further, according to Shishkov and Van Sinderen [61], *the design, implementation, deployment, and operation of context-aware applications have many interesting concerns*, including:

- ✓ *social / economical*: how to determine *useful context-aware services*, where *useful* can be defined in terms of *functional and monetary value*?
- ✓ *methodological*: how to determine and model the *context* of the *end-user* that is relevant to the *application*; how to relate the *context* to the *service* of the *application* and how to *model* this *service*; how to *design* the *application* such that the *service* is *correctly implemented*?
- ✓ *technical*: how to represent *context* in the *technical domain*; how to *manage context information* such that it is useful to the *application*; how to *use context information* in the *provisioning* of *context-aware services*?

Addressing the last two concerns (especially the last one) starts with considering possible *IT architectures* and according to Shishkov and Van Sinderen [61], *two principle architectures* could be appropriate, namely:

- **Context-aware Selection**: *end-user request(s)* and *end-user-related context information* are used to *discover a matching service* (or *service composition*). *Discovery is supported by a repository of context-enhanced service descriptions*. A *context-enhanced service description* not only specifies the *functional properties* (goals, interactions, input, output) and *non-functional properties* (performance, security, availability), but also the *context properties* of the *service*. *Context properties* indicate what *context situations* the *service* is targeting. For example, a service could provide information which is region-specific (such as a sightseeing tour), and therefore the context properties could indicate the relevance for a particular geographical area.

- **Context-aware Execution**: after the end-user request(s) has been processed and a matching service(s) has been found (possibly in the same way as described above), the *service delivery* itself would *adapt* to *changing context* during the *service session* with the *end-user*. When the *context* of the *end-user changes* in a relevant (to the application) way, the *service* provided is *adapted* to the *situation* at hand. For example, the user may move from one location to another while using a service that offers information on objects of interest, which are close-by (such as historic buildings within a radius of five kilometers, for example).

In both *context-aware selection* and *context-aware execution*, a new role is introduced, namely the role of **`context provider`**. A *context provider* is an information service provider where the information is context information. *A context provider captures raw context data and/or processes context information with the purpose of producing richer context information which is of (commercial) interest*. Interested parties could be other context providers or application providers. Further, a context-ware application obviously requires an *adaptive service provisioning component* and a *context information provisioning component*.

As far as the design of *context-aware applications* is concerned, we follow an approach that is a partial refinement of an existing one considered in [64], that concerns a general *design life cycle* comprising amongst other phases:

- **Enterprise Modeling**: during that phase, the *end-user* is considered in relation to *processes* that either support him / her directly or the goal(s) of related business(es). Those *processes* have to be *identified*, *modeled* and *analyzed* with respect to their ability to (collectively) achieve the stated goals. A model of the processes and their relationships is called an *enterprise model*.
- **Application Modeling**: during that phase, the attention is shifted from the business to the IT domain. The purpose is to derive a *model of the application*, which can be used as a blueprint for the *software implementation* based on a target *technological platform*. A model of the application, whether as an integrated whole or as a composition of application components, is called an *application model*. Enterprise models and application models should certainly be *aligned*, in order to achieve that the application properly contributes to the realization of the business/user goals. As a starting point for achieving proper alignment, one could delineate in the final enterprise model which (parts of) processes are *subject to automation* (i.e., are considered for replacement by software applications). The most abstract representation of the delineated behavior would be a *service specification of the application* (as an integrated whole), which can be considered as the initial application model.
- **Requirements Elicitation**: both the enterprise model and the application model have to meet certain *requirements*, which are captured and made explicit during the phase called *requirements elicitation*. Application requirements can be seen as a *refinement* of part of the business requirements, as a consequence of the proposition that the initial application model can be derived considering (parts of) the business processes (within the final enterprise model), especially those processes selected for automation.

- **Context Elicitation**: an important part of the design of a context-aware application is the process of finding out the *relevant end-user context from the application point of view*; we will refer to that phase as *context elicitation*. End-user context is relevant to the application *if a context change would also change the preferences or needs of the end-user, regarding the service of the application*. Context elicitation can therefore be seen also as the process of determining an *end-user context state space*, where each *context state* corresponds to an *alternative desirable service behavior*. Since relevant end-user context potentially has many attributes (location, activity, availability, and so on), a context state can relate to a complex end-user situation, composed of (statements on) several context attributes. Moreover, context elicitation relates to requirements elicitation in the sense that *each context state is associated with requirements* (i.e., preferences and needs of the end-user) on desirable application behavior. Context elicitation can best be done in the final phase of enterprise modeling and the initial phase of application modeling, when the role and responsibility of the end-user and the role and responsibility of the application in their respective environments are considered.

Figure 2.13 depicts those different phases and activities:

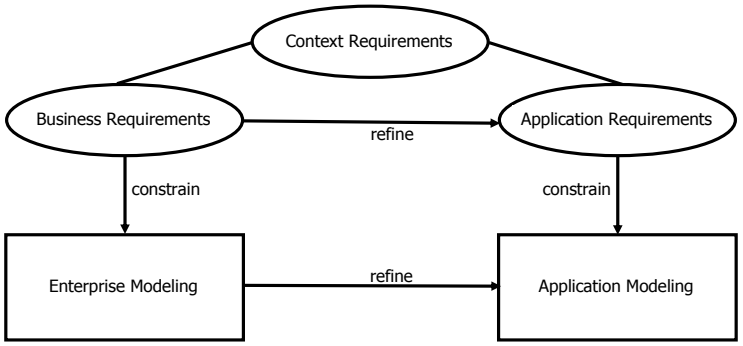


Fig. 2.13. Application design life cycle.

Following [62], *we assume that an end-user context space can be defined and that each context state within this space corresponds to an alternative application service behavior*. In other words, the application service consists of several sub-behaviors or variations of some basic behavior, each corresponding to a different context state. Any service behavior model would have to express the context state dependent *transitions* from one sub-behavior (or behavior variation) to another one.

With respect to those issues, the following *challenges* have been identified:

- Properly deciding what to *sense* and how to *interpret* it in adapting application behavior can be problematic since the interpreted sensed information must be a valid indication for a change in the situation of the end-user and it is not always trivial to know *how context information is to correspond to a user situation*.

- Deciding *which potential end-user context situations to consider and which ones to ignore* is challenging because there may be tens or even hundreds of possible end-user situations, with only several of them with high probability to occur, and therefore considering the others at design time is not sensible with respect to adequate resources expenditure.
- Modeling the application behavior including the *switching* between *alternative desirable application behaviors* can be complicated because alternative behaviors are behaviors themselves which also are to be considered in an integrated way, allowing for modeling the *switching* between them, driven possibly by *rules*.

Those challenges will be discussed below.

With regard to ***deriving context information***: an adequate decision about what should be *sensed* and how it is to be *interpreted*, concerns the extraction of context information from raw data, which relates broadly to *context reasoning* [6].

`Context reasoning` is concerned with *inferring context information from raw sensor data* and *deriving higher-level context information from lower-level context information*. As for the extraction of context information from raw data, related *algorithms* are needed to support it, and two main concerns are to be taken into account:

- the ability of specific target applications, e.g. in domains such as healthcare or finance (for example), to use the *output* of the algorithms;
- the availability of sensors providing *input* to the algorithms.

Current standard mobile devices can already operate as sensors, e.g. they can gather GPS info, Wi-Fi info, cellular network info, Bluetooth info, voice call info, and so on. In addition, dedicated sensors (that for example measure vital signs) can be integrated with existing mobile networked devices. Next to that, future standard mobile devices may even include other types of sensors, e.g. measuring temperature.

Hence, it is considered crucial developing *efficient context reasoning algorithms*, by investigating whether it is possible to derive certain specific *context information* from certain specific *sensor information*. In order to adequately refine such algorithms, additional restrictions would need to be taken into account: (i) restrictions concerning *the (specific) processing environments of mobile devices*; (ii) restrictions on *memory usage, processing power, battery consumption, wireless network usage*; (iii) restrictions that concern *real-time versus delayed availability of extracted context*.

In order to develop adequate algorithms that extract context from raw sensor data, it is therefore important to appropriately consider *gathering raw sensor data which is augmented with user input*. Concerning the sensor data, it should be *pre-processed and filtered*, in order to be properly structured as input for the context reasoning algorithms which in turn would be expected to *automatically yield the desired output*. The (delivered) context information must be of certain (minimal) quality in order to be useful; otherwise said, certain *quality-of-context* levels should be maintained.

Finally, some issues that have more indirect impact, need also to be taken into account:

(a) The delivered context information would have to be often applied in real-time environments where failures, performance requirements, available interfaces, and operational environments are to be taken into careful consideration;

(b) In order new applications to be enabled, it is important to investigate how the algorithms could be integrated in the infrastructure for context awareness.

With regard to ***context situations***, it is to be noted that it may be that there are many (tens, hundreds, and even more) possible end-user situations, for example: user at home, user driving, user busy, user out of battery, user on holiday, user in emergency, and so on. Situations are situations but *which situations are relevant*, *how many of them have high occurrence probability*, which situation corresponds to the so called *main success scenario*? Those questions points to the following claims:

- The application designers should only consider *relevant* context states. For example, if this is about arranging a phone call with John, then 'John is at home' or 'John is driving', or 'John is in a meeting', and so on are relevant context states but 'John is insured' is irrelevant.
- Out of all possible relevant context states, there should be several ones that are of *high occurrence probability* and thus all other ones are of lower occurrence probability (see the next sub-section).
  - The *high-probability context states* could be reflected at design time; this makes sense because the applications developers are preparing a 'solution-box', such that upon identifying a particular high probability context state, the application 'takes' a system behavior version out of the box – a behavior version that matches the context state; this would lead to adequate system behavior, carefully 'prepared' at *design time*.
  - The *low-probability context states*, in contrast, may be ignored at design time because spending time and resources for system behavior versions that are not expected to occur, is considered inappropriate. Still, it is possible (even though not very probable) that such context states occur. For this reason, we argue that even though not reflected at design time, such context states are to be addressable at *run time*, through intelligent algorithms.
- There should always be a *default behavior* because in our view, the application behavior modeling needs a *main success scenario* to serve as the regulation back for the system – then, any possible deviations from the main success scenario could be modeled as extensions [54].

This is illustrated in Figure 2.14:

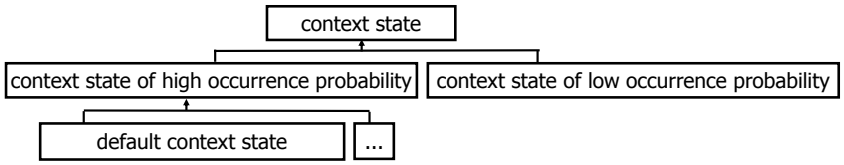


Fig. 2.14. Classification of context states.

As seen from the figure, from the perspective of developing a context-aware application, one is to distinguish between *context states of high occurrence probability* and *context states of low occurrence probability*, and the *default context state* is certainly one of the context states of high occurrence probability.

With regard to ***switching between application behaviors***, this is an important issue as well because even if context states are identified properly and also matched to corresponding desired behaviors (or addressed by intelligent algorithms), *it is a challenge to handle the mere switching between one (current) desired behavior of the application and another one (upcoming)*. Let us take for example the case of supporting a person wearing a *body-area network*, by means of a *context-aware e-health application* [6] and let us take for simplicity just two of the possible context states, namely: '*normal'* (the person is being just monitored, by transmitting data that concerns vital signs to a hospital) and '*emergency'* (the person urgently needs medical help and the goal is that the person sees a medical specialist as soon as possible, no matter who the medical specialist is or which is the hospital where the medical specialist stays, or if this would be arranged by an ambulance reaching the person). Then, if there is a context change, for example: from '*normal'* to '*emergency'*, how would this be realized? If the application would stop the vital signs data transmission and start searching for the closest medical specialist, would the vital sign info be recorded such that it is possibly used by the medical specialist? In the opposite case, if there is a context change, from '*emergency'* to '*normal'* (for example, if the person feels better and indicates that (s)he would not need emergency treatment any more) and the application would hence have to stop dealing with the emergency help arrangement and would have to go back to transmitting data concerning vital signs, then what would happen if for example an ambulance is traveling to the location of the person, should the application also take care of informing the approached medical specialist(s) that the emergency situation has been cancelled? Those examples show that *switching between application behaviors is not trivial and this challenge needs to be adequately addressed at design time*.

Summarizing the above, a *context-aware system* can be seen as concerning a sequence of *actions* that achieve: **S** (*sensing* and *capturing*), **I** (*interpretation* and *state derivation*), **W** (*switching*), and **P** (*provisioning*), as shown in Figure 2.15:

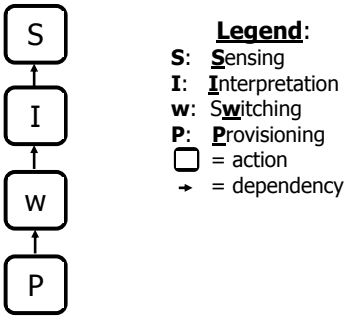


Fig. 2.15. Simplified view on a context-aware system.

With regard to **S**: The system should be able to *sense context* and *capture this context as context information*.

With regard to **I**: the system should be able to *interpret* the captured context information and *derive higher-level context information*, in particular – user context state changes, as triggers to alternative behaviors.

With regard to **w**: the system should be able to *handle the switching between its alternative behaviors*.

With regard to **P**: the system should be able to *provide services* covering all possible context states.

This is obviously a simplified model, since each of the actions represents a potentially complex process, and the dependencies between those normally involve multiple instances of information exchange and triggering. Still, the (*probabilities-driven*) *context analysis* challenge has crucial importance with regard to all above-discussed issues. For this reason, we consider this issue further in the next sub-section.

2.6.2 Context Analysis, Context States, Occurrence Probabilities, and Context Parameters

As studied by Shishkov & Van Sinderen [62], context analysis is to be about approaching the possible context states and corresponding desired behaviors, and this is to include not only studying the possible context states and their occurrence probabilities but also discovering useful context parameters whose values indicate the occurrence of particular states.

As far as **occurrence probabilities** are concerned, it is to be noted that in deciding about the states, the designer is sometimes inevitably driven by subjective judgements that are hardly supportable by rules: How a situation is perceived? What behaviors can be expected? Further, the designer must often make pragmatic decisions – ignoring, for example, states that usually do not occur (although they might occur). In our view, besides such subjective decisions, there are steps which in general help to

adequately approach the context analysis challenge. Those steps concern the consideration of **`random variables`**. Exploring their probabilities, allows us to apply **`statistical analysis`**, including *hypotheses testing* and *parameters estimation* [42].

Considering just possible outcomes is sometimes not enough in approaching a phenomenon; we might need *to refer to an outcome in general*. This is possible if we have a *random variable* and we study the *occurrence probability* of the outcomes.

Let us consider for example *land border security* and particularly the activities of border police officers on preventing illegal border crossings, supported by technical infrastructure and devices [59]. Further, let us consider particularly the case of distant monitoring: there is a camera transmitting in real-time and a border police officer is following the visual information being received; essential in this case is *whether the camera is transmitting or not* (if the camera is not transmitting, this would be alarming and there may be numerous reasons for that, such as illegal human intervention, outage, natural cause, and so on).

We can consider here the *random variable* **`Y`** with respect to those outcomes, namely: *camera transmitting* and *camera not transmitting*. **`Y`** would be a *discrete random variable* [42] since it may take on only a countable number of distinct values – two in our case. Provided the number of possible distinct values is exactly two, we have the case of **`a priori probabilities`** of each of the alternative outcomes (one of those probabilities can be calculated by deducting the other one from **`1`**).

Hence, if (for example) statistical information from the border authorities indicates that within a certain time frame, *in 80% of the time a particular camera was transmitting*, we would conclude that the *a priori* probability of the first of the mentioned possible outcomes (namely: *'camera transmitting'*) is **`0.8`**. The *a priori* probability of the second alternative outcome is thus **`0.2`**.

Hence, our *context states* represent the *'camera transmitting'* and *'camera not transmitting'* alternatives, with *a priori* probabilities **`0.8`** and **`0.2`**, respectively – Figure 2.16:

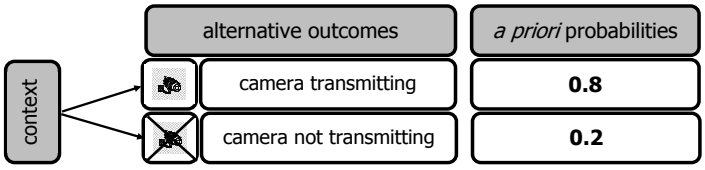


Fig. 2.16. Two context state alternatives.

It is to be noted, with regard to the current example, that even though we observe whether a camera is transmitting or not, it is not the camera that is the *end-user* of what a *context-aware application* is delivering because the *context-aware application* is not supporting the camera but the border police officer who is using the camera's output. Hence, those alternative outcomes point to two alternative situations for the border

police officer, namely: (i) *the border police officer is counting on the camera*; (ii) *the border police officer is not counting on the camera*. Depending on the situation of the border police officer, the *context-aware application* would deliver one kind of support or another.

Therefore, knowing the *occurrence probability* of each outcome helps in deciding about the *de fault* system behavior, about the optimal allocation of resources, about risks, and so on.

Further, in order to prescribe how to recognize each of the states (two in our case), we assume that the state at a particular moment is recognizable through observing the values of appropriate **`parameters`**. If we have **`n`** *parameters* appropriate to our scenario and if each of them has certain possible values, then each value combination would point to a particular state.

Then, by considering the *value combinations*, we can know the context state, by simply observing the values at any moment [62].

It is also necessary to analyze potential context states, such that the ones of high occurrence probability are identified. We argue that this may be done intuitively or on the basis of *statistical information*. We are hence interested in considering the latter in more detail.

With regard to this, we consider *statistics*, *data analysis*, and *probability theory*, and for this we refer to Freund [28].

Although *descriptive statistics* is an important branch of *statistics* and it continues to be widely used, statistical information usually arises from **`samples`** (from observations made on only part of a large set of items), and this means that its analysis requires *generalizations* which go beyond the data – this is an observed shift in emphasis from *descriptive statistics* to the methods of *statistical inference*. As for *probability theory*, it provides the basis for the methods which are used when *generalizations* are made from observed data, namely when the methods of *statistical inference* are used.

Let us take as an example a support delivered by a *context-aware application* to workers – the application supports a worker, by *informing the worker of the environmental conditions*, in general, and the concentration (in the air) of sulfur oxides, in particular, such that the worker knows if it is safe to be out or not. It is hence necessary knowing the concentration levels of sulfur oxides, which are of high occurrence probabilities. This would allow for better designing the application and to be able as well to establish a realistic work plan, knowing (approximately) how many working days to plan for the worker to work outside the factory.

In our example, we have made *80 observations* – one *sample* per one day, hence 80 days in total; let us consider the following example results (sulfur oxides in tons):

15.8	26.4	17.3	11.2	23.9	24.8	18.7	13.9	9.0	13.2
22.7	9.8	6.2	14.7	17.5	26.1	12.8	28.6	17.6	23.7
26.8	22.7	18.0	20.5	11.0	20.9	15.5	19.4	16.7	10.7
19.1	15.2	22.9	26.6	20.4	21.4	19.2	21.6	16.9	19.0
18.5	23.0	24.6	20.1	16.2	18.0	7.7	13.5	23.5	14.5
14.4	29.6	19.4	17.0	20.8	24.3	22.5	24.6	18.4	18.1
8.3	21.9	12.3	22.3	13.3	11.8	19.3	20.0	25.7	31.8
25.9	10.5	15.9	27.5	18.1	17.9	9.4	24.1	20.1	28.5

Since the smallest value is 6.2 (put on gray background) and the largest value is 31.8 (put on gray background as well), we make a choice for the following classification assuming 7 classes:

- 5.0 – 8.9 first class;
- 9.0 – 12.9 second class;
- 13.0 – 16.9 third class;
- 17.0 – 20.9 fourth class;
- 21.0 – 24.9 fifth class;
- 25.0 – 28.9 sixth class;
- 29.0 – 32.9 seventh class.

It is now necessary to establish how many items fall into each class (those are called '*class frequences*') and as well what is the corresponding percentage and the cumulative percentage:

- 3 items (out of 80) into first class 3.75% 3.75%;
- 10 items (out of 80) into second class 12.50% 16.25%;
- 14 items (out of 80) into third class 17.50% 33.75%;
- 25 items (out of 80) into fourth class 31.25% 65.00%;
- 17 items (out of 80) into fifth class 21.25% 86.25%;
- 9 items (out of 80) into sixth class 11.25% 97.50%;
- 2 items (out of 80) into seventh class 2.50% 100.00%.

This is graphically presented as histogram in Figure 2.17:

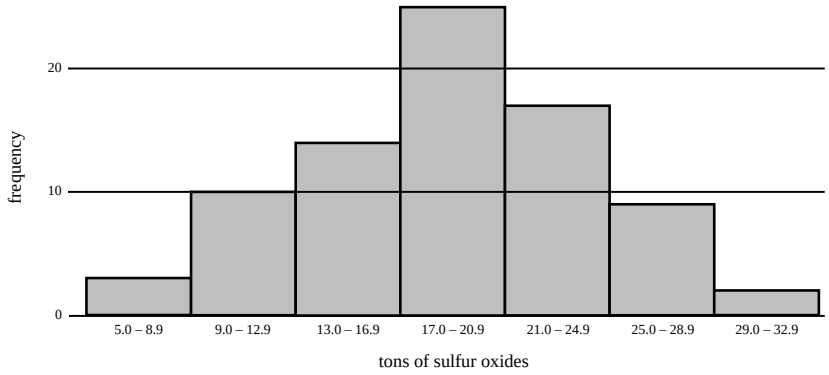


Fig. 2.17. Histogram of the distribution of the sulfur oxides emission data.

As it is seen from the figure, most items (25) fall into the fourth class:

- 31.25 percent of all sample items show between 17.0 and 20.9 tons of sulfur oxides;
- 33.75 percent of all sample items show less than 17.00 tons of sulfur oxides;
- therefore, 65.00 percent of all sample items show less than 21.00 tons of sulfur oxides.

Further, let us calculate the *mean*, by summing up the 80 numbers (15.8 + … + 28.5) and dividing the resulting number by 80: 1511,7 /80 = 18,9. In this case, we can trust that number because each of the 80 days has equal *importance weight* in contrast to cases when this is not the case, as for example observations summarized in big London against observations summarized in small Delft.

In order to avoid the possibility of getting misled using the mean (as above mentioned), it is recommended to consider the *median* – see Figure 2.18:

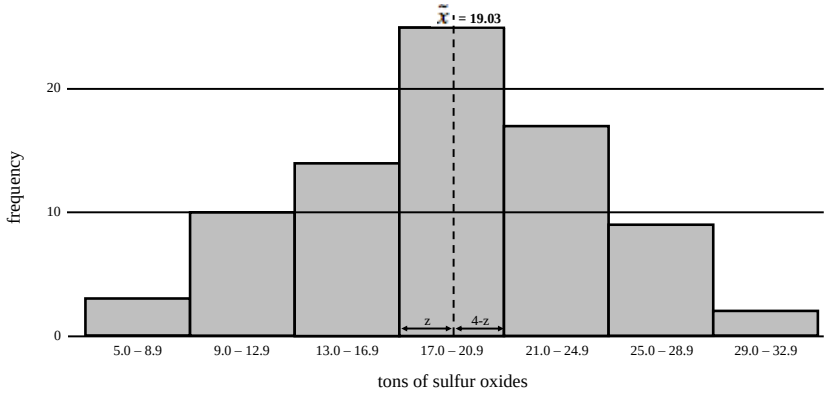


Fig. 2.18. The median of the distribution of the sulfur oxides emission data.

The median should '*split*' the sample items, such that 50% of them have values smaller than the value the median points to and hence 50% of them have values greater than the value the median points to. In the considered example, we need to find this number that fulfills the following: 50% of the sample sulfur oxides values are smaller than the number and 50% are greater. On the figure, the median is displayed in dashed line.

We find the median of the distribution of the sulfur oxides emission data in the following way:

1. We note that 33.75% of the sample items have values lower than 17.00 (this can be seen from the numbers presented above); we note also that 50.00% of the sample items have values lower than the so called '*median value*' (pointed by the *median* as shown in Figure 2.18). The difference between the two is: 50.00% – 33.75% = 16.25%. We note also that the mentioned 33.75% correspond to: first class + second class + third class while at the same time 31.25% correspond to the fourth class only (those are values greater than 17.0 and smaller than 20.9). Thus the median value corresponds to the fourth class (because 16.25% is smaller than 31.25%). For this reason, we state that the median value equals to 17 + z, which means that the '*distance*' between the median value and 21 (where the fifth class 'begins'), equals to: 4 – z, because we have class intervals of 4 (9.0 – 5.0 = 4, 13.0 – 9.0 = 4, and so on).

2. We then split the fourth class into two sub-classes, namely four-L and four-H, such that: (i) the items belonging to the fourth-L sub-class have values that are greater than 17 and smaller than the median value; (ii) the items belonging to the fourth-H sub-class have values that are greater than then median value and smaller than 21. Thus: (a) 31.25% of all sample items belong to fourth-L sub-class + fourth-H sub-class; (b) 16.25% of all sample items belong to the fourth-L sub-class (50.00% - 33.75% = 16,25%: see above); (c) thus, 15.00% of all sample items belong to the fourth-H sub-class (31.25% - 16.25% = 15.00%); (d) 52.00% of the sample items belonging to the fourth class, belong to the fourth-L class ((16.25/31,25)*100).
3. We assume that the values in each class are *evenly distributed* (spread evenly throughout the class); this would mean that if 52.00% of all values belonging to the fourth class belong to the fourth-L class, then 52.00% of the whole class interval (that is 4) correspond to z (Figure 2.18) which is the 'sub-class interval' corresponding to the fourth-L sub-class. This would mean: **z = 52% * 4 = 2.08**.
4. The way amounts have actually been grouped in the considered example is precise to the point of the nearest tenth of a ton (5.0, 8.9, 9.0, and so on) and this is to assume *refinement* to some extend – for example, considering that 5.0 includes everything from 4.95 to 5.05, the class 5.0 – 8.9 includes everything from 4.95 to 8.95, and so on. Such a desired *level of precision* points to the so called '*class boundaries*' – if we assume such *level of precision* for the example, this would mean that **the lower boundary of the fourth class is 16.95**.
5. Hence, in order to find the median value, we should add the corresponding sub-class interval (2.08) to the lower boundary of the class (16.95): **16.95 + 2.08 = 19.03**, as also seen from Figure 2.18.

Hence, half of the sample items have values that are smaller than 19.03 and the other half of the sample items have values that are greater than 19.03.

In summary, in the current example:

- the MEAN equals to 18.90;
- the MEDIAN equals to 19.03.

In the example, as explained already, both values are very close and we could *round this to* `19.00`, hence claiming that for the period in which the sample values were taken, it may be expected that **the amount of sulfur oxides (in tons) in the air would be around 19**. If this is acceptable, according to the regulations, then this would mean that it is to be planned than in most days, workers would be able to work out; otherwise, it is to be planned that in most days, workers are to be kept inside the factory, for example.

Let us *assume* that *19.00* points to *possibility to work out*. In this case, **the default application behavior would assume that the end-user is working out** and only if the situation of the end-user changes – the amount of sulfur oxides in the air goes above the norm, the application would switch to another behavior that assumes instructing the end-user to get inside, and so on.

Thus, in developing context-aware applications, it is helpful conducting *data analysis* as above-suggested, such that the default application behavior is adequately determined – with regard to this, the *data distribution* is to be considered, as well as the *mean* and/or the *median* values; still, with regard to those issues, we are not going in

more detail in the current section, noting nevertheless that what *statistics* and *probability theory* offer can be even more instrumental (through other concepts and approaches as well) with regard to *context analysis*.

IN SUMMARY, in the current chapter, we introduced our systemics views, touching upon systems and their composition, on one hand, and the system environment and context of users, on another hand, extending this to enterprise systems and EIS. In the following two chapters, we will present relevant social theories (Chapter 3) and computing paradigms (Chapter 4), elaborating on how the concepts and views considered in the current chapter, can be rooted both enterprise-wise and technology-wise.

Chapter 3

SOCIAL THEORIES

As mentioned already, we are considering in this chapter social theories, such that we ground our modeling views concerning enterprise systems and EIS, as presented and discussed in the previous chapter, and in particular – we are addressing those issues that are about the **human aspects** relevant to enterprise systems and EIS. This is because human behavior, human decisions, human communication, human failures, and so on are all about the functioning of any enterprise system or EIS. For this reason, we need to be explicit in considering such issues when modeling / designing such systems. Further, just referring to a theory would be insufficiently useful because aligning concepts and views to a particular theory is not trivial – we argue that this could only be achieved if concepts (and views) are bridged to theories, driven by particular **concerns** since only such an approach can justify the selection of particular theories. For this reason, we firstly present several concerns whose identification has been inspired by Shishkov [54] and in line with the views presented in the previous chapter:

- **Intuitive Behavior**: There are human entities in any enterprise system / EIS and often their behavior is driven by interpretations, knowledge, judgements, beliefs, values, and so on – those are not always easy to objectively observe and identify. For this reason, it is claimed that intuitive human behavior is an essential concern that needs to be addressed explicitly in the analysis and design of such systems.
- The **Human Element**: In line with the above paragraph, it is to be noted that human entities differ from any other non-human entities, such as devices, applications, and so on, because all processes in Society are human-driven – it is only humans who have rights, it is only humans who benefit from social prosperity, it is only humans who can be kept responsible, and so on. For this reason, no matter what is happening (for example, a drone is in the air, monitoring a land border), it is to be possible to 'trace' this to corresponding human authority and responsibility.

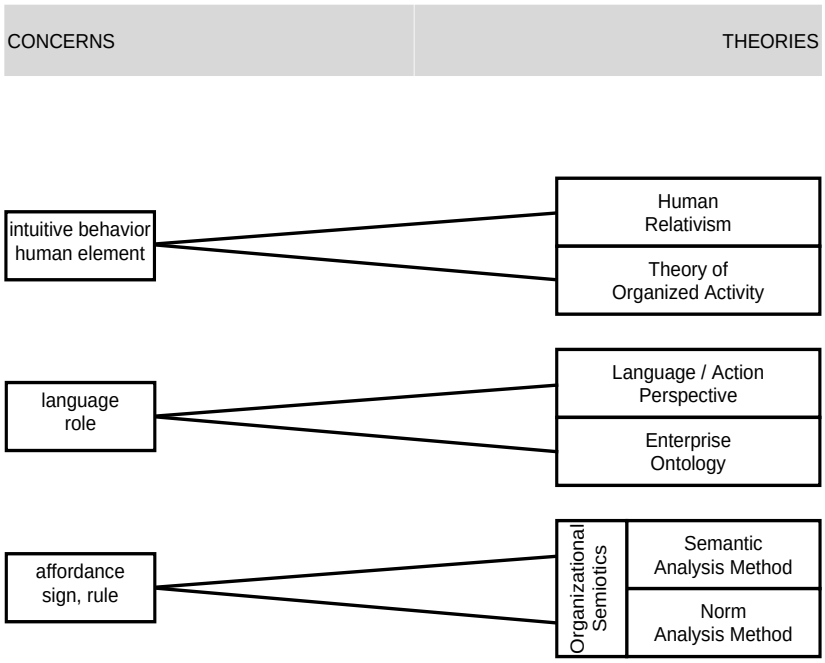


Fig. 3.1. Concerns and theories.

- **`Language`**: Human entities, being part of any enterprise system / EIS, communicate among each other, using language – this goes beyond what is just driven by rules, since through language, human entities give promises, express disagreement, lead negotiations, and so on. Such issues have impact on the functioning of a system and need to be adequately modeled.
- **`Role`**: Human entities are more sophisticated in their behavior than technical entities – unlike a technical entity which follows 'embedded' rules only, a person would often think, make decisions (especially in exceptional situations), and so on, and this may result in conflicts with the rules; hence, it may happen that a human entity realizes activities that are not part of his/her 'job profile' (a professor faxing for example, with this being part of the secretary's responsibilities); for this reason, we argue that it is appropriate modeling roles, not just the (human) entities fulfilling those roles, as already discussed in Chapter 2.
- **`Affordance`**: There are many different objects that need to be considered when modeling an enterprise system / EIS and what is important in this regard is reflecting their features and capabilities, for example: in a library, a book affords to be borrowed; thus, we consider the 'affordance' concept useful as it concerns the modeling of (enterprise) systems.

- **Sign**: In an enterprise context, often something stands for something else, for the sake of properly conveying semantics to corresponding entities, for example – in case of fire within a building, if a person is not sure which direction to follow in order to leave the building, in case a green light can be seen from somewhere, the person would take this direction because it is widely accepted that 'Exit' signs are colored green and illuminated; hence, the green light helps the person make the right decision how to proceed in a complex situation – this is thus a sign (we have discussed the 'sign' concept in the previous chapter, and we argue that this notion should be adequately reflected in the modeling of enterprise systems / EIS.
- **Rule**: Any (enterprise) system is essentially governed by regulations and rules (norms) and for this reason, it is essential to be capable of adequately reflecting rules in modeling enterprise systems / EIS.

Following Shishkov [54] and considering recent studies [8], we have established that: (i) **human relativism** and the **Theory of Organized Activity (TOA)** well cover the *human element* and *intuitive human behavior*; (ii) the **Language / Action Perspective (LAP)** and **enterprise ontology** well cover the (language-based) *human communication* and corresponding *roles* that corresponding human entities can fulfill; (iii) **organizational semiotics**, in general, and the **semantic analysis method** as well as the **norm analysis method**, in particular, well cover the concepts of *affordance*, *sign*, and *norm* (rule), as suggested by Figure 3.1.

For this reason, in the remaining of the current chapter, we will firstly consider human relativism and TOA, secondly – LAP and enterprise ontology, and thirdly – organizational semiotics.

We consider those social theories as underlying with regard to our concepts, views, and way of modeling, as it concerns enterprise systems and EIS, in general and the modeling of human aspects in this context, in particular. As mentioned before, social theories are insufficient when it also comes to ICT and software – for this we need as well computing paradigms, such that the social theories applied and the computing paradigms followed are complementary with regard to each other. We address the social theories in the current chapter and the computing paradigms – in the following chapter.

3.1 Human Relativism and TOA

In order to provide theoretical principles with respect to the necessity of taking properly into account the human element and its behavior, in [16], a new philosophical stance – **human relativism** – was proposed, together with an analysis of *human action* seen as the kernel element of any approach following that stance. The same perspective characterizes **TOA** where *organized activity* is the key concept. Those theories will be addressed further on in the current section.

3.1.1 Human Relativism

Human relativism, as considered in [16] takes a World perspective consistent with *functionalism*, *social relativism*, *radical structuralism*, and *neo-humanism*, as presented in [33], also establishing the possibility of complementary using formal methods and theories, for the sake of overcoming the limitations of most objectivist stances, related mainly to cases of *unpredictable behavior* usually related to the human element – this including most inter-subjective experiences, such as interpretation, knowledge, beliefs, intentions, value, and so on, which often remain hidden from our senses. It is claimed that scientific methods and objectivism are unable to deal with human behavior in general since it is impossible (from such a perspective) to reproduce or predict things like interpretation or understanding, or to regulate mechanically human actions [16].

To tackle this from the perspective of human relativism assumes acknowledging the human centeredness and the unpredictable behavior of human entities. Human relativism recognizes this human centrality in all human activities, by acknowledging an objective reality as human relative. There are many evidences of this human relative view even in objectivism. The visible images transformed from infrareds into the visible spectrum, for example, allow us to experience a different reality where human bodies cannot be easily separated from the environment, because there are no clear boundaries. However, this reality is in fact seen and experienced by some animal species as science proofs. In this sense we may question ourselves, which is the real reality, the reality we observe with our vision or the reality observed using, for instance, the infrared spectrum? Or, are they different views of the same reality? There is no claim in human relativism that the reality we see is the real reality, neither an explanation nor sense of what a real reality is. The solution is more a practical solution – this is the reality we have, we experience and we share. By assuming the human at the center we also assume and accept his/her view as bounded, focused and particular.

Further, information is human-related as well – information is extracted by humans from the reality using perception and interpretation processes. The distinction between perceptions, the process of acknowledging the external reality through our senses, and interpretation, the meaning making process, is a useful way to help understanding the nature of information and its acquisition process. Only information goes through an interpretation process, the other elements of the (human) reality are just perceived. In fact, perception filters part of the human reality accessible to a particular individual.

To perceive does not mean to interpret and this separation allows us to understand what observable is. Usually, ***observability*** concerns what we think a human being is able to percept or acquire through his/her senses. This excludes the interpretation process and information as well. Usually information is not observable but it can be extracted from observable things. Observable things can be viewed as material or physical things from the objectivist view, for example - a smile (that is an observable thing) may express happiness (that is not an observable thing). At the same time nonetheless, observing a smile on the face of a person does not guarantee happiness – this is a matter of interpretation and also, different persons may express themselves differently. To solve this ambiguity or meaning problem, the above-mentioned *observability* concept is the first step and with regard to this, human relativism has the following assumption:

ASSUMPTION Anything that is **observable** will be more *consensual*, *precise*, and therefore more *appropriate* to be used by *scientific methods*.

Further, in considering the notion of '*observability*' it is necessary to consider a related notion as well, namely, the notion of **precision**.

According to [15], to have a high degree of *precision* means to have a reduced level of ambiguity and different meanings in some term or element making it generally accepted, recognized, and shared. One way of achieving *precision*, for example, is the use of physical measurement.

This leads to stating an important *human relativistic* hypothesis:

HYPOTHESIS By adopting **observable** elements or **high precision** elements under a *human relativistic* view, it is possible to derive a scientific and theoretical well-founded approach to EIS.

Those basic *human relativistic* ideas are claimed to be aligned with social *constructivism* and *objectivism*, making a proper connection between them.

Since most enterprise systems / EIS are 'challenged' by issues related to the human element, such as unpredictability (and this prevents the use of scientific and objective methods), *human relativism* identifies and highlights this point, by recognizing **human behavior** as an essential challenge with respect to those issues.

Those thoughts point to another important *human relativistic* hypothesis:

HYPOTHESIS We may freely apply *technical approaches* if there is *no unpredictable behavior* present, specifically *human behavior*.

Hence, *human relativism* points a way to overcome the difficulty in dealing with unpredictable behavior, in particular *human behavior*. When approaching *human behavior*, one would realize that what is 'seen' is just the *observable* part of the behavior – the *observable human actions*. One should then acknowledge importance of the unpredictable aspects of *human behavior*, for building adequate models of enterprise systems /EIS. Still, besides just acknowledging those issues, *human relativism* proposes ways to cope with *ambiguities* resulting from unpredictable *human behavior*:

- to reduce the dependability of the enterprise system / EIS on *human behavior*;
- to better use the power of *human behavior*, through support coming from tools that are not only facilitating humans but are also stimulating them to generate feedback that in turn could help to better capture the different aspects of *human behavior*.

Thus, building upon other philosophical stances, human relativism is essentially focusing on human behavior with recognition of the fact that even though precision can be achieved, observable behavior is just a part of the complex human behavior, and in order to cope with this complexity, one could either make systems less dependent on human behavior or introduce tools that not only support humans with regard to their actions but also help the system better capture the different aspects of human behavior.

As already mentioned, in the following sub-section, we will further the discussion on *human behavior*, by addressing the *Theory of Organized Activity – TOA*.

3.1.2 TOA

The *Theory of Organized Activity – TOA*, proposed by Anatol Holt [34] considers a concept relevant to *human behavior*, namely the concept of **`Organized Activity`**, or **`OA`** for short, and Anatol Holt states the following with regard to that concept:

> *"I intend the expression 'organized activity' to mean a human universal. Like language, organized activity exists wherever and whenever people exist. It will be found in social groups of a dozen, or in social groups of millions - in the jungle and in New York City, in every culture, and at every stage of cultural/technological history. It is manifest in every form of enterprise, whether catching big game, coping with a fire, or running a modem corporation – even acquiring and communicating by language."*

This is how Anatol Holt positions the *OA* concept acknowledging the *TOA* emphasizing the following issues that concern any *OA*:

- A common communication language – expressed not only by words, but by actions and things as well, known as *units* and recognized by people sharing or involved in the same activity. Behind this idea there is an essential and associated meta-theory called the *theory of units*.
- *Actions* – which directly affect, involve or act on things or materials. *Actions* are related to a *temporal dimension*.
- *Bodies* – representing things or materials, related to a *material dimension*.
- *Action Performers* – always *persons* and/or *organizational entities*.

TOA is thus mainly considering *actions*, *bodies*, and *action performers* as well as their inter-relationships.

As far as *actions* are concerned, *TOA* emphasizes especially on *human actions*, acknowledging that *responsibility can only be attributed to humans*, which would mean that *computers and other tools cannot perform actions*.

As for *action performers*, *human actions* are motivated and driven by them (in the interest of the *action performers*).

Figure 3.2 [34] defines the OA kernel. The figure is presenting two dichotomies, namely *persons* <-> *organizational entities* and *actions* <-> *bodies*, suggesting that (as according to Anatol Holt) any OA, no matter how complex and subtle, can be usefully represented in those terms.

Besides the *action* and *body* concepts, *TOA* also defines the concepts of *state* and *information*. A *state* in *TOA* only applies to *bodies* and is only understood within specific domains of *action*. This notion makes a *TOA state* different from the usual technical description of a *state*. Regarding *information*, in *TOA* it has the exclusive end use of making decisions, which determines the following course of *actions*. *Information* in *TOA* is carried in lumps by *bodies*, being those lumps exclusive properties of those *bodies*. *Information* contents of a *body* depend on the context of its use and on the particular actors performing the *actions*. The same *information* can be used differently by different actors or in different contexts.

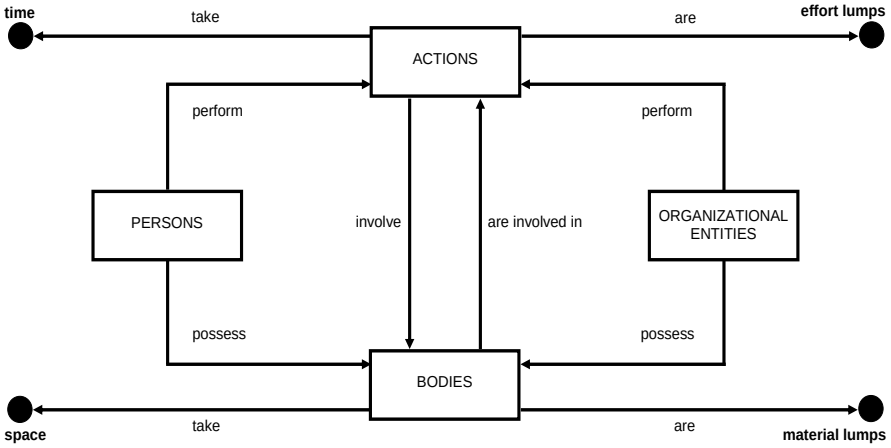


Fig. 3.2. The OA kernel.

Anatol Holt claims that itt is only TOA that:

- relates *information* to human decision;
- has potential to define *measures* consistent with those of Claude Shannon;
- makes explicit all real-world operations performed on real-world *information*.

Thus, we claim that both *human relativism* and *TOA* provide a useful perspective on enterprise systems / EIS, emphasizing on *human behavior*. In the following section, we are going to consider the (*language-driven*) *communication* among (human) entities.

3.2 LAP and Enterprise Ontology

According to *Definition 10*: "*A complete model is a model that is elaborated at least in three perspectives, namely structural perspective, dynamic perspective, and data perspective*", and as suggested in the previous chapter, if one would be considering an *enterprise system* or an *EIS*, one would be interested in capturing the *structure* of the system, the system's *behavior*, and the corresponding *data flows*. As also suggested in the mentioned chapter nevertheless, is that the *human-to-human communication* (characterizing *enterprise systems* and *EIS*) needs to be considered as well – actually, the *communicative actions* (related to *human-to-human communication*) are related to the *transaction* concept (Definition 5) and *transactions* are considered as the elementary building blocks of *enterprise systems*. For this reason, besides addressing *structural* issues, *behavioral* issues, and *data* (or *factual*) issues, we need to take as well a **`communicative perspective`** concerning *human-to-human communication*. This perspective is addressed in the current section, and in particular – *LAP* and *enterprise ontology*.

3.2.1 LAP

Taking a *communicative perspective* in approaching an *enterprise system*, is motivated by the importance of grasping not only the *structural*, *factual*, and *behavioral* (*dynamic*) *enterprise system* aspects but also the *communicative* aspect [52], and one of the most sound and popular theories behind that issue is the *Language / Action Perspective – LAP* [82]. *LAP* is a theoretical orientation towards approaching the modeling of *business processes*, by emphasizing the importance of *interaction* and *communication*. The theory recognizes that *language is not only used for exchanging information, as in reports (for example), but that language is used also to perform actions, as in promises or orders (for example)*. Such *actions* are claimed to represent the foundation of communities and organizations, and must be understood to create effective *EIS*. For this reason, it is claimed that adequately capturing the *communicative aspects*, characterizing the considered *enterprise system(s)*, would contribute to the creation of sound and complete *business process models* [54]. Further, referring to the *white-box* vs *black-box* enterprise systems modeling, reflecting *construction* vs *function* (Figure 2.9), it is to be noted that applying LAP allows for revealing the *construction* and *operation* of an enterprise, not just capturing the enterprise dynamics. Such a direction corresponds to the consideration of *transactions* as enterprise modeling elements (Definition 5).

Hence, taking a **`white-box perspective`** and considering the notions of *actor*, *production*, and *coordination* (as explained in the previous chapter), *LAP* suggests that the *functional behavior* of an enterprise is brought about by the *collective working of its constructional components* [54]. The *construction* and the working of a system are most near to what a system really is, to its ontological description [10]. Acknowledging Bunge's vision, Dietz takes a *LAP* perspective in considering enterprises, claiming that an *enterprise is a discrete dynamic system in the category of social systems, having social individuals or actors, each of them having a particular authority to perform production acts* (*P-acts*) and a corresponding responsibility to do that in an appropriate and accountable way; *the structure of an enterprise consists of coordination acts* (*C-acts*), i.e. the *actors* enter into and comply with commitments regarding the performance of *P-acts* [23] – that all points to the *generic white-box model* of an enterprise, consisting of the actors, the *P-world*, and the *C-world* [54], as presented in Figure 3.3:

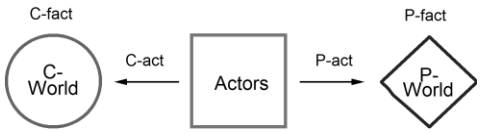


Fig. 3.3. The white-box model of an enterprise.

C-acts concern *human-to-human communications*. An instance of such kind of communication consists of *two human processes*:

- *a sender (role 1) expressing something (a message)* and
- *a receiver (role 2) interpreting the message.*

What can be communicated between a sender and a receiver? *Elementary communicative acts*, such as **request**, **promise**, **state**, **accept**, and so on, are considered from the *LAP* perspective. This is consistent with *Definition 5* according to which "*a transaction is a finite sequence of coordination (communicative) acts between two actors concerning the same production fact*". Hence, **production acts** and **coordination (communicative) acts** appear to be performed in particular sequences or chains that can be viewed as paths through a **generic pattern** pointing to a *transaction* [23], and also, in the enterprise context *Role 1* (see above) would correspond to **customer** while *Role 2* – to **producer**.

Hence, a more elaborated (and *LAP*-driven) view on *transactions* suggests that a *transaction* is a **finite sequence of C-acts between two actor-roles**, the *customer* and the *producer*. It takes place in *three phases*: the **order phase** (*O-phase*), the **execution phase** (*E-phase*), and the **result phase** (*R-phase*). *O-phase* is a conversation that starts with a request by the *customer* and that, if successful, ends with a promise by the *producer*. *E-phase* basically consists of the performance of the *P-act* by the *producer*. *R-phase* starts with a statement by the *producer* that the requested *act* is performed and ends, if successful, with an acceptance by the *customer*. All this is reflected in the generic transaction pattern depicted in Figure 3.4:

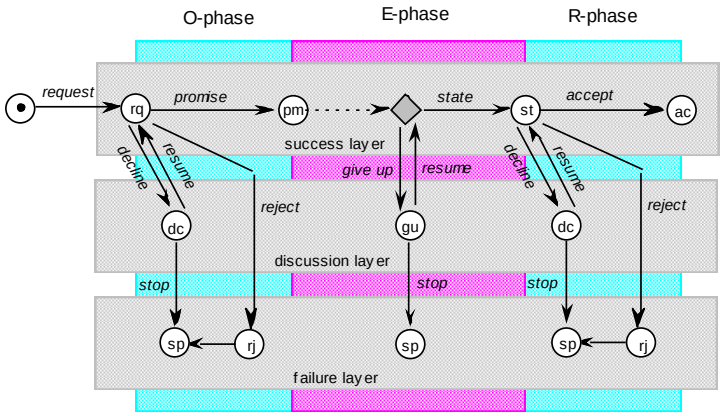


Fig. 3.4. The transaction pattern.

As it is seen on the figure, besides the three phases – *O-phase*, *E-phase*, and *R-phase*, there are as well three layers, namely: *success layer*, *discussion layer*, and *failure layer*; further, the *elementary communicative acts* considered and their abbreviations are as follows:

- **rq** *request*
- **pm** *promise*
- **st** *state*
- **ac** *accept*
- **dc** *decline*
- **rj** *reject*,

and as well two 'factual' acts are considered, namely: **sp** (*stop*) and **gu** (*give up*). The generic transaction pattern needs to be further explained and for this we will use a toy example reflecting the situation in which a customer enters a small pizza shop.

Let us firstly consider the *success layer*:

The customer (John) enters the shop and requests a pizza to be delivered to him, assuming to pay for this according to the announced prices. The person at the desk (Tim), realizing that the ingredients for the requested pizza (such as cheese, tomato paste, and so on) are available, promises to deliver a pizza to John. Then Tim goes to the kitchenette and prepares the pizza for John. Up to this point, we have two *elementary communicative acts*, namely: *request* and *promise* (as it can be seen from the figure, communicative acts are presented as *disks*) and one *production act*: the pizza preparation (as it can be seen from the figure, production act is presented as *diamond*). Further, after having prepared the pizza, Tim comes back to John, bringing the pizza to him, stating that the request was fulfilled. John takes the pizza and pays, implicitly meaning that he is satisfied with the result and accepts what was delivered (a pizza in this case). **It is only the acceptance that makes the transaction completed**. Said otherwise, if such an acceptance is not reached (and for example, John refuses to pay and goes out), then there is no transaction, no matter how many *communicative* / *production acts* have taken place.

Let us secondly consider the *discussion layer*:

If after John asks for a pizza, Tim, realizing that not all pizza ingredients are available, declines the request, this puts John and Tim into some kind of *negotiations*. As part of such *negotiations*, Tim may announce that even though he cannot deliver a pizza, he can deliver a sandwich instead, for example. Then, there are two possibilities – John either agrees to have a sandwich or not. If John agrees to have a sandwich, this means that John introduces a new request (instead of requesting a pizza, John is already requesting a sandwich). To this Tim promises to deliver a sandwich to John (new promise) and all goes back to the *success layer*. If nevertheless, John would decide that he would not go for a sandwich, then all goes to the *failure layer*, as shown on the figure. Considering further the *discussion layer*: if after having started the pizza / sandwich preparation, Tim unexpectedly experiences an electricity outage, this would result in his impossibility to adequately finalize the delivery – this puts John and Tim into *negotiations*. As part of such negotiations, Tim may announce that due to an electricity outage, he cannot deliver the pizza / sandwich within reasonable time but, based on information from the electricity supplier, he expects all to be back to normal within one hour, for example, and hence, he can deliver the pizza / sandwich with a one-hour delay (for example), because he had to temporarily give up the pizza / sandwich preparation, causing in this way inconvenience to John, but this could be

compensated by a lower price, for example. If John would agree, this would mean that implicitly John has made a request assuming the 'new' conditions (new request) and Tim has promised to deliver according to the 'new' conditions (new promise), and Tim is in the process of preparing the pizza / sandwich according to the 'new' conditions (new production). Then all goes back to the success layer. If nevertheless, John has no time to wait, then all goes to the failure layer, as shown on the figure. And in the end, if the pizza / sandwich is ready and Tim delivers it to John, *stating* that what was *requested* was fulfilled, it is possible that instead of *accepting* the pizza / sandwich, John declines accepting the delivered result, for example – if John finds the way the pizza / sandwich looks inadequate. This puts John and Tim into *negotiations*. Tim may offer a lower price as compensation for the inadequate look of the pizza / sandwich, for example. If John would agree, then all goes back to the *success layer* and this would mean that implicitly John has made a request (new request) and Tim has made a promise (new promise) as according to the 'new' conditions, the pizza / sandwich was delivered (new production and new statement) and paid according to the new conditions, the result is accepted and this means that the **transaction is completed**. If nevertheless John would not like to accept the delivered pizza / sandwich even at a new (lower) price, then all goes to the *failure layer*.

Let us finally consider the failure layer:

No matter if the transaction has reached the failure layer because John would not like to have a sandwich instead of a pizza (O-phase) or because John would not like to wait more (E-phase), or because John would not like to accept a pizza / sandwich that according to John has a look that is inadequate, even at a lower price (R-phase), as in the considered example, the **transaction is incomplete**; this means that *nothing essential has objectively happened in reality*.

Thus, by modeling an *enterprise system* / *EIS* in terms of *actor-roles* and *transactions*, we assume the **`potential`** for anything to take place among *actor-roles*, which is nevertheless not necessarily to happen.

Further, as mentioned in the previous chapter, we consider *transactions* as the *atomic enterprise modeling units* and this does not contradict with the fact that *transactions* in turn represent a sequence of *C-acts* (as mentioned above). What matters with regard to the *business processes* is whether there is a completed *transaction* or not – the *C-acts* alone are not enough to justify a *business process*. For example, if a person would use a cash machine just to enter his/her personal identification number and would then stop, this would leave no 'business trace', or if a person would just ask (within a pizza shop) what the price of a pizza is and would then leave. In those examples, we observe *C-acts* but no completed *transactions*.

Finally, our *systemics* concepts and views are claimed to be consistent with *LAP* and for this reason, we especially emphasize the **`transaction`** concept that is considered to have essential importance in this regard.

In the following sub-section, we consider the theory of *enterprise ontology* as proposed by Dietz [19] not only because this theory is partially based on *LAP* but also because some views of Dietz have influenced our previous work [54].

3.2.2 Enterprise Ontology

The *DEMO methodology* [17] has been developed on the basis of *LAP* and reflected in the *SDBC approach* [54]. This has inspired Dietz [19] to consider *LAP* in combination with *philosophical ontology* [10] and *organizational semiotics* [43] to propose the **Ψ**-*theory*, underlying **Enterprise Ontology** (**EO**). The overall goal of the **Ψ**-*theory* / *EO* is **to extract the essence of an enterprise from its actual appearance**, such that corresponding *white-box* models could be adequately derived – this is the enterprise ontological modeling. The **organization theorem** has crucial importance with regard to the above-mentioned goal and the *theorem* in turn is essentially backed by four axioms, namely: the **operation axiom**, the **transaction axiom**, the **composition axiom**, and the **distinction axiom**, as shown in Figure 3.5:

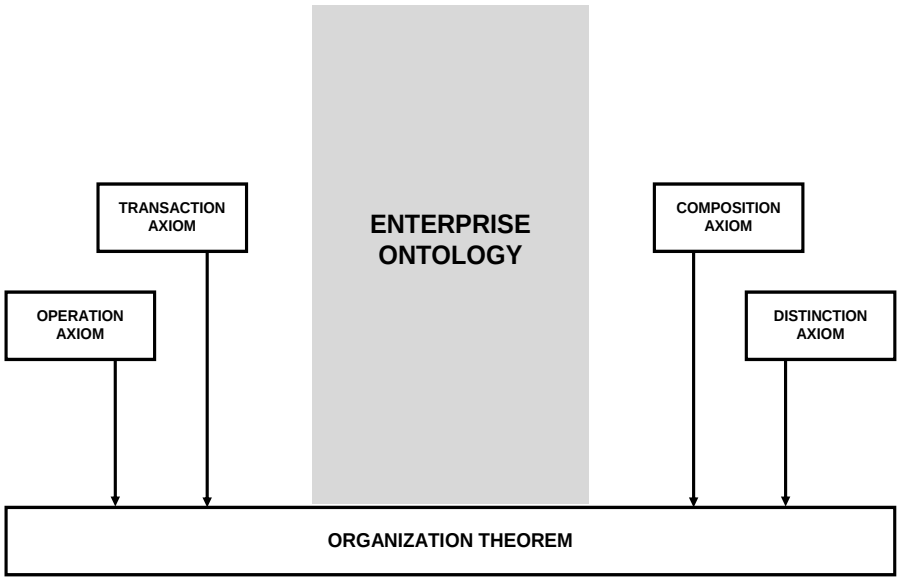


Fig. 3.5. EO – background.

Hence, in the remaining of this sub-section, we will firstly consider the *operation axiom*, secondly – the *transaction axiom*, thirdly – the *composition axiom*, fourthly – the *distinction axiom*, and finally – the *organization theorem*.

Operation Axiom

The *operation axiom* states that the **`operation`** of an *enterprise* (see Figure 2.9) is constituted by ***actors*** (see Section 2.2) who perform two kinds of *acts*, namely ***P-acts*** and ***C-acts*** (see Figure 3.3), as according to *LAP*.

By performing **`P-acts`**, *actors* contribute to bringing about the *goods* and/or *services* that are delivered to the *environment* of the *enterprise* under consideration (assuming that in the current sub-section we consider *enterprise systems* – see *Definition 2*).

Further, in line with the discussion on *material things* and *immaterial things* (see Section 2.1) related to the notion of '*product*' (see *Definition 2*), we note that *P-acts* could be either *material* or *immaterial*:

- examples of *material P-acts* are manufacturing acts, storage acts, transportation acts, and so on;
- examples of *immaterial P-acts* are judgement acts of a court (to sentence someone, for example), decision acts of an insurer (to grant an insurance claim, for example), appointment acts (bringing someone to the presidency of a company, for example), and so on.

By performing **`C-acts`**, *actors* enter into and comply with *commitments* towards each other regarding the performance of *P-acts*.

Transaction Axiom

Referring to the *LAP*-driven *transaction pattern* (see Figure 3.4) and to the *operation axiom*, we establish that

- a *C-act* is performed by one *actor* (called '*producer*') and directed to another *actor* (called '*customer'*);
- *C-acts* are always, either directly or indirectly, about *P-acts*.

Thus, the notion of *transaction* refers to the question how *P-acts* and *C-acts* are related to each other, and the *transaction pattern* is referred to as a *generic coordination pattern* in the above context.

Hence, the *transaction axiom* recognizes *the LAP-driven transaction pattern* according to which **`transactions`** always involve two *actor roles* and are aimed at achieving a particular *result*.

Further, taking the perspective of EO, a **`conversation`** is defined *as a sequence of C-acts between two actor roles that are aimed at achieving a well-defined result concerning a P-act*.

Thus, a *transaction* actually consists of two *conversations*, namely:

- an *actagenic* conversation (it is about the *order*) and
- a *factagenic* conversation (it is about the *result*).

If we consider the *transaction pattern* (Figure 3.4), we see that the *actagenic conversation* points to the *order phas*e and the *factagenic conversation* points to the *result phase*, while between them is the *execution* of the *P-act*, which both *conversations* are about.

What can also be seen from the pattern is that the *INITIATOR* of the *transaction* is the *customer* while the *EXECUTOR* of the *transaction* is the *producer* (it is the *customer* who would *request* a pizza, for example and this would *initiate* the

transaction, and also with respect to the same example, it would be the *producer* who would prepare and deliver the pizza, in this way *executing* what has been requested). Hence:

- in the *order phase*, the *initiator* and the *executor* work to reach an agreement about the intended *result* of the *transaction*, i.e., the *production fact* that the *executor* is going to create as well as the intended time of creation;
- in the *execution phase*, this *production fact* is actually brought about by the *executor*;
- in the *result phase*, the *initiator* and the *executor* work to reach an agreement about the *production fact* that has actually been produced, as well as the actual time of creation (both of which may differ from what was originally requested. Only if that agreement is reached will the *production fact* come into existence, as discussed in the previous sub-section.

Composition Axiom

The *composition axiom* concerns the **`business process`** notion (see *Definition 6*), considering a *business process* to be a structure of causally related *transaction* types. All causally related *transactions* are executed in order to fulfill a *starting transaction* – such a *starting transaction* is either *activated* from the *enterprise environment* or is *self-activated* on the basis of some kind of *self-activation condition*.

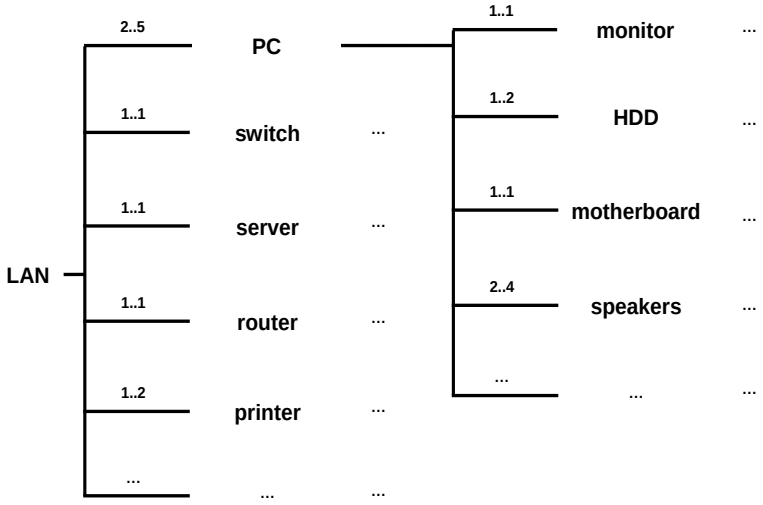


Fig. 3.6. A component structure of a LAN.

Said otherwise, something is requested to be delivered but in order for it to be delivered, the result of something else would be needed, and so on – we will illustrate this by means of a hardware example:

- a Local Area Network – LAN [12] is requested to be installed in an office;
- before configuring the LAN, the following is needed: a server, Personal Computers – PCs, a switch, a router, printer(s), and so on
- before a PC is delivered, the following is needed for its assembly and configuration: a motherboard, HDD(s), a monitor, speakers, and so on.

This is illustrated in Figure 3.6 to be read from left to right, suggesting that in order to configure a LAN (in the particular case), one would need 2 to 5 PCs, one switch, one server, one router, 1 to 2 printer(s), and so on, for example, and in turn for a PC to be configured, one would need one monitor, 1 to 2 hard drives (HDD), one motherboard, 2 to 4 speakers, and so on. Hence, one should firstly get the monitors, HDDs, motherboards, speakers, and so on, such that the PCs are configured, and then the same for the switch, the server, the router, and so on, and only after all of this has been realized, the LAN would be ready to be installed – this is a good example for **`causal relationships`** discussed above. The same is with the *transaction* belonging to a *business process* in an *enterprise* context: similarly to the need to configure a LAN (as in the above example), some kind of *starting transaction* needs to be *executed* and in order for it to be executed, it is necessary that (*before it gets executed*) other *transactions* get executed, and they may need in turn still other *transactions* to be executed, and so on. For this reason, the way we have presented such causal relationships in Figure 3.6 is considered helpful and we will apply the same way of representation (one entity type depends on the entity types to the right of it and the possible number of instances for each entity is given to the left of its label as interval).

Let us consider a simple example from the enterprise domain: A student (John) visits a property agency asking ADVICE in the form of recommendation – which is the best available for rent property, matching his demands. The consultant (Steve) from the agency is capable of delivering such kind of advice to John, assuming that John would pay for the delivered consultancy. Nonetheless, in order to deliver the advice, Steve would need (before delivering the advice) to realize some kind of MATCH-MAKING 'between' the demands of John and the characteristics of the available properties. And in turn, in order for Steve to realize such kind of match-making, he would have to do (before realizing the match-making) two things, namely: (i) REQUEST PROCESSING, such that the demands of John are appropriately reflected in standardized forms such that their effective use is possible and (ii) DATA SEARCH, such that there is an actual list of all currently available properties. Thus, Steve should firstly do the request processing and the data search, and only on that base he would be able to realize the match-making, and it is the match-making that is needed by Steve, such that he is able to deliver the requested advice to John. This is illustrated in Figure 3.7:

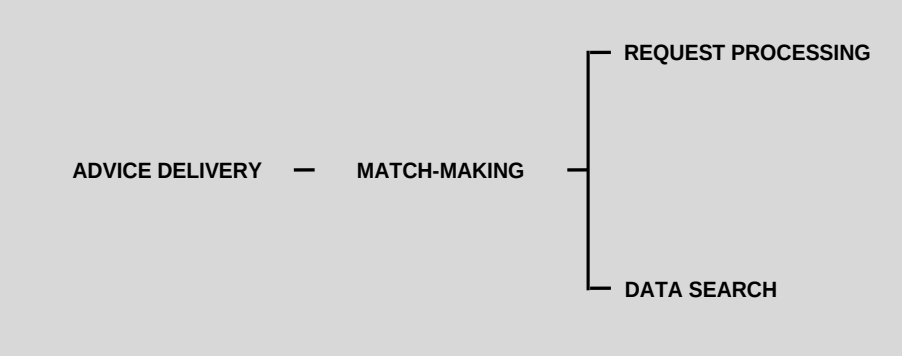


Fig. 3.7. Illustrating a causal relationship.

Thus, reading the figure from left to right suggests that ADVICE DELIVERY can be realized but under the condition that MATCH-MAKING is realized first. What the figure suggest as well is that MATCH-MAKING can be realized but under the condition that REQUEST PROCESSING and DATA SEARCH are realized first.

This represents a *business process* that is *driven by the goal of fulfilling the ADVICE DELIVERY starting transaction*.

Hence, after considering *elementary acts* (see the *operation axiom*) and *transactions* (see the *transaction axiom*), we are considering the *composition axiom* that addresses **`business processes`**.

Distinction Axiom

The *distinction axiom* serves to separate the distinct human abilities playing a role with regard to **`communication`** and in order to give useful background (claimed to be helpful in understanding the axiom), we refer to the so called *semiotic ladder* [43] that presents the (human-to-human) *communication* in terms of *layers*, in the following way:

- PHYSICAL WORLD: If two persons would like to communicate, they need physical conditions – this could be their closeness in terms of space, such that they can hear each other or a telecommunications channel, such as telephone connection, and so on.
- EMPIRICS: Even if the persons have physical conditions to communicate, the communication channel itself is to also be adequate – for example, if the persons are close to each other but there is too much noise, they would not be able to hear each other or if they have established a telephone connection but the quality of service is too low for them to hear each other well and without delays.
- SYNTAX: If the persons have adequate physical conditions and communication channel, this is still not enough for a full value communication to take place because they need to speak the same language or use the same communication patterns.

- SEMANTICS: If the persons are adequately exchanging information using the same language, for example, this is still not enough if they do not get the correct meaning. For instance, if John is at a garage, needing his car to be repaired urgently and he sees a queue of 10 cars, and it looks obvious that the garage would not be able to serve all those cars within the day, and if John asks the receptionist whether it would be possible his car to be treated urgently, and the receptionist answers 'Yes, as long as the cars from the queue get served', this actually means 'No', because it is obvious that the car of John would not be served the same day. If the receptionist had wished to mean 'Yes', he would have answered, for example: 'There are many cars in the queue but we will make an exception and treat you with priority'. This example, shows that the syntactic 'Yes, as long as the cars from the queue get served' has the meaning of 'No'. Hence, getting correctly the semantics is necessary in order to communicate of full value.
- PRAGMATICS: Even if the persons are adequately handling the communication both physically and also empirically, syntactically, and semantically, they also need to adequately handle the context in which they are communicating – for example, if John's colleague says to John 'I am freezing' and John is close to the widely open window during winter time, it is not enough that John gets the right meaning of what his colleague is saying; what goes beyond the meaning is that John should realize that by saying this, his colleague is trying to convince John to close the window, and John is expected to 'participate' in this negotiation (about whether to close the window or not) and not discuss with his colleague the way he is feeling.
- SOCIAL WORLD: Even if pragmatics, semantics, and so on are all handled adequately, there are societal norms of behavior that need to be respected. In the above example, it is expected that John would close the window even if John is not feeling cold because it is societally adequate to respect (when possible) the needs of the persons around. In this case, the colleague of John is not feeling good and John may like to help because closing the window would not immediately hurt John's comfort – still, this would help another person feel better and this is to be considered good behavior from a societal point of view.

In order to align the above semiotic perspective to communication, we consider the corresponding views of Habermas [32] who has identified three spheres of human existence, that play a role with respect to communication, namely: (i) objective world those are the things that are outside the subject and to a large extend exist on their own; (ii) subjective world – unique for every distinct subject; (iii) social world – what the subjects build and maintain in interaction. Then:

- With regard to (i), the (human-to-human) communication is aligning the concept of TRUTH => Here we have the class of acts for which the dominant validity claim is the claim to truth, for example *assertions* (John asks Betty what time it is, for instance, and then Betty would assert the current time).
  This is labelled as **`constativa`**.

- With regard to (iii), the (human-to-human) communication is aligning the concept of JUSTICE. => Here we have the class of acts for which the dominant validity claim is the claim to justice, for example *requests* and *promises* (If I request a loaf from the baker, for instance, I primarily claim that I am in the social position to do so and that the baker is in the social position to be addressed with the request; I hence accept the authority and responsibility of the baker to respond to the request, and the baker accepts my authority and responsibility to make a request, as exemplified by Dietz [19]).
  This is labelled as **regulativa**.
- With regard to (ii), the (human-to-human) communication is aligning the concept of SINCERITY. => Here we have the class of acts for which the dominant validity claim is the claim to sincerity, for example *praises* and *apologies* (If I bump into somebody, for instance, my apologizing is to convey to the person information that I am sincere, otherwise, and apology would not make sense).
  This is labelled as **expressiva**.

Next to that, 'non dominant' claims are possible as well, mixing up the above issues and several examples considered by Dietz [19] are brought forward in this regard:

- If I appear to be near a head of state and I ask him/her what time it is, things about *truth* and *justice* are mixed up because it is not considered just that one ask the time to the head of state.
- If I ask from a baker 100 loaves of the same time, things about *justice* and *truth* are mixed up because objectively, it is impossible for him to deliver at one 100 loaves.
- If John asks Richard what time it is and after hearing the answer, he asks Betty the same question, things about *truth* and *sincerity* are mixed up because if John knows the time already, is he sincere saying to Betty that he wants to know what time it is?

In this respect, **EO is primarily about regulativa** since: (a) It is assumed that the *constativa issues are taken indirectly*; (b) The *expressiva issues are disregarded* and this is not because emotions are considered unimportant but because they fall outside the ontological view on enterprises, as according to Dietz [19].

Hence, in the pizza example from the previous sub-section, just one elementary communicative (coordination) act (for example: 'the person at the desk promises to deliver a pizza'), as we label it 'C-act' for short, assumes communication conforming to the semiotic ladder (see above) and in the regulativa perspective, and in this we bring together the pragmatic and social considerations (as according to the semiotics ladder) claiming that the following three layers bring together the above views, taking a LAP perspective:

- ✓ **PERFORMA**: This is the actual act of evoking an attitude (for example, the customer had the person at the desk PROMISE to deliver a pizza or a conversation in a library and the context of the conversation had a person REQUEST membership, and so on). => This brings together the behavioral pragmatics and the societal relevance, as according to the semiotics ladder.

- ✓ **INFORMA**: This is about conveying semantics – for example, John may well explain in a library that he would like to have them deliver a pizza to him and they may get this correctly semantically but still this would not lead to a promise from their side because the situational context and social relevance are inappropriate with regard to what John is suggesting. => This corresponds to the semantics layer of the semiotics ladder.
- ✓ **FORMA**: This is about conveying information of full value and using the same language or communication pattern – for example, John may utter many sentences at a pizza desk and what John is saying may be adequately heard and syntactically understood but still, it may not be the case that they understand that John is asking a pizza to be delivered to him. => This brings together empirics and syntactics, as according to the semiotics ladder.
- ✓ Finally, the physical conditions necessary for such kind of communication, are acknowledged by EO but not explicitly considered since they are claimed to fall outside the ontological view on enterprises.

This is illustrated in Figure 3.8, summarizing the *distinction axiom*:

COORDINATION performer's perspective	HUMAN ABILITY	COORDINATION addressee's perspective
exposing commitment	PERFORMA	evoking commitment
expressing thought (formulating)	INFORMA	educing thought (interpreting)
uttering information (speaking, writing)	FORMA	perceiving information (listening, reading)

Fig. 3.8. Summary of the distinction axiom.

As it is seen on the figure: (i) The *forma ability* (bringing together *empirics* and *syntactics*) is about conveying information, as above mentioned, for example – *uttering* and *perceiving* of sentences in some language. (ii) The *informa ability* (building upon

the *forma* layer) is about conveying *semantics*, as above mentioned, for example – *interpreting* what was said or written, *getting the correct meaning*. (iii) The *performa ability* (building upon the *forma* layer and the *informa* layer) is about *bringing in new original things*, rightfully considering the context (pragmatics) and the societal relevance, as above mentioned, for example – *engaging into commitments*.

Hence, the *distinction axiom* states that there are three distinct human abilities playing a role in the operation of actors, namely: **`performa`**, **`informa`**, and **`forma`**, as explained and discussed already.

We consider the performa ability as the essential human ability for doing business of any kind.

Organization Theorem

We have already introduced, explained, and discussed **`four EO axioms`**, namely: the **`operation axiom`**, the **`transaction axiom`**, the **`composition axiom`**, and the **`distinction axiom`** – this brought focus on the:

- ***actor roles*** as *composition elements* of *enterprise systems* as well as their potential to realize ***production acts*** and ***coordination acts***;
- three basic ***human communicative abilities*** (*performa*, *informa*, and *forma*) with regard to the performance of *production* / *coordination acts*;
- ***transactions*** as the *atomic enterprise modeling units*;
- *causal relationships among transactions*, justifying ***business processes*** as *structures of transactions*.

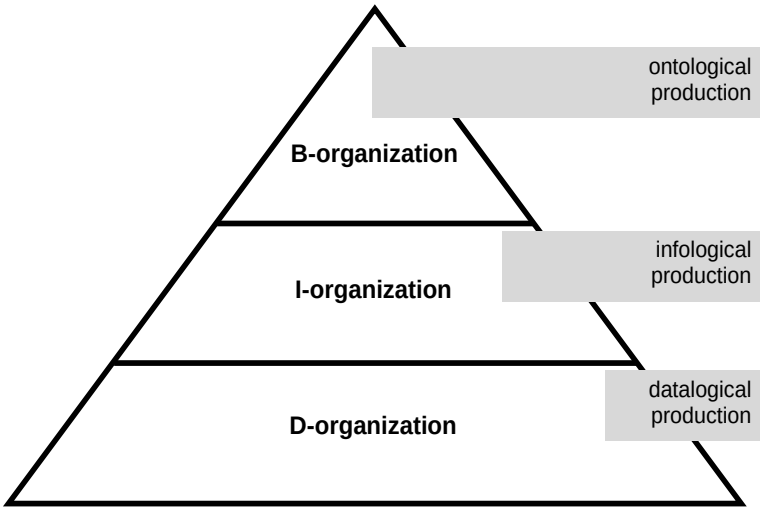


Fig. 3.9. Representation of the organization theorem.

Hence, the *goal of the* **organization theorem** *is to establish, based on the mentioned axioms*, a **concise, comprehensive, coherent, and consistent enterprise notion** *corresponding to a white-box (constructional) perspective*.

The **organization theorem states that an enterprise is a heterogeneous system that is constituted as the layered integration of three homogeneous systems**: the **B-organization** (from BUSINESS), the **I-organization** (from INTELLECT), and the **D-organization** (from DOMUMENT), related among each other in the following way (as shown in Figure 3.9):

- ✓ *the D-organization supports the I-organization*;
- ✓ *the I-organization supports the B-organization*.

All three *homogeneous systems*, as represented in the figure, are in the category of *social systems*, which means that they are similar as far as *coordination* is concerned: the elements are subjects that enter and comply with *commitments* to each other regarding *production acts* (in line with *LAP*). They differ only in the kind of *production*:

- the *production* in the *B-organization* is ONTOLOGICAL;
- the *production* in the *I-organization* is INFOLOGICAL;
- the *production* in the *D-organization* is DATALOGICAL.

This is the reason for considering an *enterprise* to be a *heterogeneous system* and hence the *B-organization*, the *I-organization*, and the *D-organization* represent *aspect systems* of the (total) *enterprise*.

As acknowledged by Dietz [19], an *enterprise* is more than just a well-established integration of those three aspect organizations. Firstly, human beings as *biological* beings need a particular environment to live in, as well as specific facilities to make their biological lives comfortable. Being a biological individual includes being a physical thing. Hence, *physical requirements* must be met, like the need for work space and mobility services. Moreover, a human being is an *emotional* being, a *psychological* being, and so on. While it is recognized that those additional aspects must be considered, they are *irrelevant as far as EO is concerned* since they do not directly relate to the notion of *enterprise*. Still, we consider as precondition dealing with those human aspects in a satisfactory way.

Thus, we argue that by considering LAP-EO, one could build enterprise models that are adequately rooted in corresponding real-life processes. In the following section, we are going to consider *semiotics*, emphasizing on *semantics* and (business) *rules*.

3.3 Organizational Semiotics

It is considered useful applying the *Semiotics Theory* [13], regarding issues connected with the analysis and modeling of *business processes* and *enterprise systems*. Actually, a branch of *semiotics* is considered, namely – **Organizational Semiotics** (**OS**), and in particular two *OS* methods: the *Semantic Analysis Method* and the *Norm*

Analysis Method [67,69,43]. *OS* focuses on the *nature*, *characteristics* and *behavior* of **signs**. The term '*organizational semiotics*' was officially coined in 1995 at an international workshop in Enschede, The Netherlands, after a long time of research on organizational studies and information systems. This section considers briefly some essential issues related to the *OS* theory.

Peirce founded *semiotics* as the *'formal doctrine of signs'* [51]. A *sign* is defined as *something that stands to someone for something else in some respect or capacity*. *OS* and the analytical methods [67,69,43] offer a theory to understand *enterprises*, with or without the computerized *information systems*. *Enterprises* are deemed as *systems* where *signs* are *created*, *transmitted*, and *consumed* for *business purposes*.

Stamper and his *school of OS* argue that in contrast to the concept of *information*, *signs offer a more rigorous and solid foundation to understand information systems*. For example, within a business context, a bank note is much more than a piece of colored paper with digits on it. It stands for the bank note holder's wealth and ability to pay, as well as the issuing bank's authority and credibility, and much more. Large quantity of underlying social relationships and behavior possibilities are attached to those business concepts; oversimplifying them into pure digits would be dangerous. On one hand, computers can only process and manipulate such digits; on the other hand, the underlying meanings and possibilities must be exposed to enable the correct processing. Adopting the concept of *sign* enables us to study the *enterprise* in a more balanced way, taking account of both *the technological issues, and the human and social aspects of information resources, products, and functions*.

OS adopts a subjectivist philosophical stance and an agent-in-action ontology. This philosophical position states that, for all practical purposes, nothing exists without a perceiving agent and the agent engaging in actions.

Stamper adopts the concept of **affordance** from the perceptual psychologist James Gibson, who defined *the affordances of the environment* as 'what it offers the animal, what it provides or furnishes, either for good or ill…' [29]. Based on the theory, since a person perceives things by recognizing what he can do with them or to them, a *thing* can be defined as an *invariant repertoires of behaviors, either substantive affordances or social norms that are available to the responsible person* [67]. For example, in the context of a university library, a book affords to be borrowed by a library user.

Borrowing a book is a *potential ability*, which may or may not be implemented in the reality. Nevertheless, once it is implemented, *new possibilities may emerge*. For example, a borrowed book may be retained or returned to the library by the user. Under certain circumstances, the library may also call it back. This shows that *affordances have dependency relationships among them*. In *OS* such a relationship is called *ontological dependency*.

We may schematically show this relationship as following, with the **antecedents** on the left side and the **dependencies** on the right, and the solid line denotes the *ontological dependency*:

book – borrow – return

Ontological dependency does not only show the logic relationship between the concepts. What's more important is that it shows the *dependencies get their meaning from the existence of the antecedents*. Since the existence of *dependencies* would not be possible without the existence of the *antecedents*, the lifecycle of the *dependencies* is always included by that of the *antecedents*. The existence of the *antecedents* thus forms a context for the *dependencies*.

For example, talking about returning a book without referring to the fact that the book was previously borrowed from the library would be off the topic.

Further, two essential *OS methods* considered (as it was mentioned already) are the **`Semantic Analysis Method`** and the **`Norm Analysis Method`**. Those methods are briefly discussed below.

3.3.1 Semantic Analysis

The *Semantic Analysis Method* is fundamentally based on the *Semiotic Theory* that has been discussed above. This method is a method for *elicitation and specification of user requirements*. It considers the *signs* created by members of an *enterprise*. *Semantic analysis* is theoretically founded in *OS* [68] and the *semiotic* framework. The method has been applied in many fields such as *user requirements* for *enterprise systems*, *organizational analysis*, *legal documents design*, and *analysis and design of Computer Systems* [43,54]. The *semantic analysis* is conducted usually in *four steps*, outlined below, and the final result is a *semantic schema*, called **`ontology chart`**:

- Taking into account that *semantic analysis* deals with analysis of documents and conversations, the first step that is to be realized, is to *gather relevant data and understand the problem*. This can be called *problem statement*.
- The second step is to *produce a list of semantic units such as verbs, nouns, adjectives and adverbs*. Those semantic units may be used to *describe human agents and their respective patterns of behavior*.
- The third step is to *further analyze the semantic units by linking them together according to their relationship in terms of generic-specific positioning*. This is shown graphically from the left to right on an *ontology chart*.
- The fourth step should *bring together all the linked semantic units into a coherent whole*, which produces a *complete semantic model*. The model is represented graphically through an ontology chart.

3.3.2 Norm Analysis

When studying *enterprises* from the *perspective of entities' behavior* it is necessary to specify the **`norms`** *based on which this behavior is realized*. *Norms* [70] are *the rules and patterns of behavior, either formal or informal, explicit or implicit, existing within a society, an enterprise, or even a small group of people working together to achieve a common goal*.

Norms are determined by *Society* or collective groups, and *serve as a standard for the members to coordinate their actions*. An individual member uses the knowledge of

norms to guide his or her actions. If the *norms* can be identified, the *behaviors* of the individuals, hence their collective *behaviors*, are mostly predictable. From this perspective, *to specify an organization can be done by specifying the norms* [71] and this holds also for enterprises.

Four types of norms exist, namely *evaluative norms*, *perceptual norms*, *cognitive norms* and *behavioral norms*. Each type of norms *governs human behavior from different aspects*. In *business process modeling*, most *rules* and *regulations* fall into the category of **behavioral norms**. Those *norms* prescribe *what people must, may, and must not do*, which are equivalent to *three deontic operators*: '*is obliged*', '*is permitted*', and '*is prohibited*'. Hence, the following format is considered suitable for specification of *behavioral norms*.

```
whenever <condition>
if <state>
then <agent>
is <deontic operator>
to <action>
```

It is essential to recognize that *norms* are not as rigid as logical conditions. If a person does not drink water for certain duration of time he cannot survive. But an individual who breaks the working pattern of a group does not have to be punished in any way. For those actions that are *permitted*, whether the agent will take an action or not is seldom deterministic. This *elasticity* characterizes *business processes*, therefore is of particularly value to understand the corresponding *enterprise*(*s*).

A *norm analysis* is normally carried out on the basis of the results of a *semantic analysis* (for information on *semantic analysis* interested readers are referred to [43]). The *semantic* model delineates the area of concern of an *enterprise*. The *patterns of behavior* specified in the *semantic* model are part of the fundamental *norms* that retain the ontologically determined relationships between agents and actions without imposing any further constraints. Nevertheless, *norm analysis* could be successfully related also to other modeling tools, as studied by Shishkov [54].

In general, a complete *norm analysis* can be performed in *four steps*:

- First step: *Responsibility Analysis*;
- Second step: *Proto-norm Analysis*;
- Third step: *Trigger Analysis*;
- Fourth step: *Detailed Norm Specification*.

Responsibility analysis enables one to identify and assign responsible *entities* (or '*agents*' as according to the *OS* terminology) to each *action*. The analysis focuses on the *types of agents* and *types of actions*. In an *enterprise*, responsibilities may be determined by the organizational constitution or by common agreements in the *enterprise*.

Proto-norm analysis helps one to identify relevant types of information for making *decisions concerning a certain type of behavior*. After the relevant types of information are identified, they can be used as a checklist by the responsible agent to take necessary factors into account when a decision is to be made. The objective of this analysis is to *facilitate the human decisions without overlooking any necessary factors or types of information*.

Trigger analysis is to consider the actions to be taken in relation to the absolute and relative time. The absolute time means the calendar time, while the relative time makes use of references to other events. The results of trigger analysis are *specifications of the schedule of the actions*.

The *detailed norm specification* concerns the specification of *norms* in two versions, a natural language and a formal language. The purposes of that are (1) to capture the *norms* as references for human decision, and (2) to perform actions in the automated system by executing the *norms* in the formal language.

For those *norms* identified in the *business processes*, some refers to the major authorities and responsibilities of the major figures in the *enterprises*. Those *norms* govern some trivial, relatively less important norms or those of lower priorities, from the perspective of organizational functionalities [54]. This strongly suggests the possible *hierarchies* exist not only in the *enterprise structure*, but also in the *norms*. The terms *'framing norm'* and *'contractual norm'* are used to express such kinds of *hierarchies* [69].

Hence, among the *EIS*-relevant *strengths* of *OS* are the following:

- *Semantic analysis* is powerful in situations in which it is necessary to put some unstructured information in order. This is an unavoidable task in any software project.
- *Norm analysis* is powerful in situations in which it is necessary to specify *rules* and also to relate a number of *rules* to each other. Hence, *semiotic norms* could be much useful in both *business process modeling* and *software specification* – both tasks include consideration of *rules*.
- *Semantic analysis* and *norm analysis* are founded in the *OS* theory; it is a well-established theory relevant to both *business process modeling* and *software specification*.

Nevertheless, as studied by Shishkov [54], those semiotic methods alone are not capable of soundly and completely aligning enterprise modeling and software specification; those methods need to be incorporated in an approach that would not only combine them adequately with other relevant social theories (besides OS) but would also relate them to appropriate computing paradigms.

IN SUMMARY, in the current chapter, we presented and discussed *social theories*, including *human relativism*, the *theory of organized activity*, the *language/action perspective*, *enterprise ontology*, and *organizational semiotics*, justifying their relevance to different aspects concerning *enterprise systems* and *EIS*. In the following chapter, we will consider in turn *computing paradigms* that are currently actual and also well-combinable with the mentioned *social theories* and consistent with the *concepts* and *views* introduced in Chapter 2.

Chapter 4

COMPUTING PARADIGMS

As a starting point with regard to what will be presented in the current chapter, we take the distinction between *procedure-oriented programming* and *object-oriented programming* [54,80]:

- *Procedure-oriented programming* (or *procedural programming*) uses a list of instructions to tell the computer what to do step-by-step. *Procedural programming* relies on PROCEDURES - a *procedure* contains *a series of computational steps* to be carried out. *Procedural programming* is *intuitive* in the sense that it is very similar to how a person would expect a program to work: if one wants a computer to do something, one should provide step-by-step instructions on how this is to be done. Examples of *procedural languages* include the early programming languages, such as Fortran and COBOL, and later on – Pascal and C, which have been around in the 1960s, 70s, 80s, and 90s.
- *Object-oriented programming* is an approach to problem-solving where all computations are carried out using **`objects`**. An *object* is a *component of a program* that 'knows' how to perform certain actions and how to interact with other elements of the program. *Objects* are the basic units of *object-oriented programming*. A simple example of an *object* would be a person. Logically, one would expect a person to have a name. This would be considered a *property* of the person. One would also expect a person to be able to do something, such as walking, for example. This would be considered a *method* of the person. A *method* in *object-oriented programming* is like a *procedure* in *procedural programming* (the key difference is that the *method* is part of an *object*). Hence, in *object-oriented programming*, the code is to be organized by *creating objects*, giving those objects *properties*, and so on. A key aspect of *object-oriented programming* is the use of **`classes`**. A *class* is a blueprint of an *object*: a *class* can be considered as a concept and an *object* - as an embodiment of that concept. For example, if a person is to be considered in a program, then one should be able to describe the person and have the person do something. A class called 'person' would provide a blueprint for what a person looks like and what a person can do. Examples of object-oriented languages include C++, Java, and so on.

A key difference between the two is that in procedural programming, procedures operate on data and those two concepts, namely 'procedure' and 'data', are two separate concepts while in object-oriented programming those two concepts are bundled into objects. This makes it possible to create complicated behavior with less code. The use of objects also makes it possible to re-use code. Once one has created an object with more complex behavior, one could use it anywhere in the code.

A further move to *component-oriented programming* has been inspired by those advantages [73]: With *object-oriented programming* focusing on the relationships between *classes* that are combined into one large binary executable, *component-oriented programming* focuses on interchangeable code modules that work independently and don't require you to be familiar with their inner workings to use them.

Thus, we observe an evolution

```
from procedure-oriented programming
through object-oriented programming
to component-oriented programming.
```

That evolution in *programming* has not only been useful as a stimulus to more effectively and efficiently producing code but it has also influenced the broader process of *software engineering* comprising requirements analysis, system analysis, system design, coding, testing, and implementation, with justifying an evolution from *monolithic software engineering* through *component-based software engineering* to *service-oriented software engineering* [54,72]:

Developing a **`monolithic`** *application* assumes result that is monolithic binary code. It may be that one even applies *object-oriented programming* and still the bottom-line is *monolithic* development – one may factor the business logic into many fine-grained *classes*, once those *classes* are compiled, if the final *application* is viewed that way (to be *monolithic*), then the result is *monolithic* binary code: all the *classes* share the same physical deployment unit (typically an EXE), process, address space, security privileges, and so on. Hence, if multiple developers work on the same code base, they have to share source *files*. Thus, in such an *application*, a change made to one *class* can trigger a massive re-linking of the entire *application* and necessitate retesting and redeployment of all the other *classes*.

In contrast, a **`component-based`** *application* comprises a collection of interacting binary application modules —that is, its *components* and the calls that bind them. The motivation for breaking down a *monolithic application* into multiple binary *components* is analogous to that for placing the code for different *classes* into different *files*. By placing the code for each *class* in an *application* into its own *file*, one would loosen the coupling between the *classes* and the developers responsible for them. If one would make a change to one *class*, although one would have to re-link the entire *application*, one would only need to recompile the source *file* for that *class*. Further, because a *component-based application* is a collection of binary building blocks, one can treat its *components* like LEGO bricks – simply 'adding' and 'removing' them. If one would need to modify a *component* implementation, changes are contained to that *component* only. No existing *client* of the *component* requires recompilation or redeployment. *Components* can even be updated while a *client application* is running,

as long as the *components* are not currently being used. Improvements, enhancements, and fixes made to a *component* would immediately be available to all *applications* that use that *component*, whether on the same machine or across a *network*. Finally, when one has new requirements to implement, one can provide them in new *components*, without having to touch existing *components* not affected by the new requirements. All those advantages have contributed to the increasing popularity of *component-based* applications, compared to *monolithic* applications.

The next step in those developments was marked by the appearance of **service-oriented** *software*: *component-based* software is about how one would build and implement a *system* – taking the whole *system* and dividing it into smaller better manageable *components*, and so on, while *service-orientation* is about how different *systems* communicate with each other, based on defined various standards for message formats, transport security, and so on. Hence, that is about allowing users to compose *services* at high-level, which *services* are realized by underlying *software components*. The advantages here are two-fold: (i) the technical complexity, characterizing *software components*, remains 'hidden' from the user who is composing *services* at 'higher level'; (ii) a user can bring together *services* whose underlying *software components* may be created by different developers, running on different servers, and so on.

Thus, we observe an evolution

from monolithic software engineering
through component-based software engineering
to service-oriented software engineering.

That *software engineering* evolution has not only been useful as a stimulus to more effectively and efficiently producing and utilizing software but it has also influenced in a broader perspective the *way of developing*, justifying an evolution from code-centric development through model-driven development to agile development [66]:

The **code-centric** development (considered in the past) would not support the analysis and design activities by *modeling* while the idea to use *models* for improving software development practices was gaining increasing popularity.

That led to the emergence of **model-driven** development that is not only about helping developers to reason at 'higher level' supported by *models* but is also about distinguishing between *computation-independent* and *technology-specific* issues being reflected in corresponding *model* types. This is considered as a viable 'bridge' between the '*Software World*' and the '*Real-life World*' in a sense that firstly, domain-related specifications are defined and secondly, those domain-related specifications are reflected, by means of model transformations, in corresponding platform-specific models, envisioning platforms, such as CORBA, J2EE, .Net, and so on. Model-driven development is hence attractive for its capability of bringing together domain-specific issues and technology-specific issues, by allowing for model transformations, as above mentioned. Nevertheless, the lack of sufficient development flexibility and collaborativeness as well as the insufficient capability to conveniently adapt modeling to changes, has justified the need for new development paradigms.

That has inspired the emergence of **agile** development that is based on iterative development, where requirements and solutions evolve via collaboration between self-

organizing cross-functional teams. *Agile* processes fundamentally incorporate iteration and the continuous feedback that it provides to successively refine and deliver a software *system*. Hence, *agile* development is people-centric, in contrast to model-driven development that is model-centric and also in contrast to code-centric development.

Thus, we observe an evolution

```
from code-centric development
through model-driven development
to agile development.
```

With regard to what was stated in the above paragraphs, it is to be noted that some of the paradigms discussed assume *distributed* computing environments (for example: *service-oriented* software engineering would envision the composition of *services* realized by *components* running on different computing environments) while others implicitly assume *mobility* (for example: *agile* development would often envision dynamic user feedback, possibly generated through *applications* running on *mobile* devices). This has justified an evolution from *mainframe* infrastructures, through *client/server* infrastructures, to *cloud* infrastructures [6,12]:

A **`mainframe`** infrastructure is based on a mainframe and terminals. A mainframe can be looked upon as a 'giant server' since only it serves 'dumb' terminals. Such a terminal has no drives, no independent operating system, and so on – it has just a screen and a keyboard. All data of any type is contained in the mainframe. Any info changed or added from a terminal would change the data in the mainframe.

In contrast, a **`client/server`** infrastructure assumes the partitioning of tasks or workloads between the providers of a resource or service, called *servers*, and service requesters, called *clients*. Hence, those principles are underlying with regard to current distributed computing environments. What such distributed computing environments lack as capability nevertheless is enabling 'outside' stakeholders to be served, possibly through their portable devices connected to the Internet.

This has inspired the emergence of **`cloud`** infrastructures assuming the provision of shared computer processing resources and data to computers and other devices on demand. *Cloud* infrastructures have hence become underlying with regard to current mobile computing environments.

Thus, we observe an evolution

```
from mainframe infrastructures
through client/server infrastructures
to cloud infrastructures.
```

With respect to the paradigms considered above, most challenges mainly relate to functional issues. Nevertheless, there are non-functional **`crosscutting concerns`**, such as security, privacy, recoverability, logging, performance monitoring, and so on. In the past, this was considered as part of the requirements elicitation, then the label '*crosscutting concerns*' was dominant, and currently we speak of *aspect-oriented software development* considering *crosscutting concerns* (called '*aspects*') at all stages of the software development life cycle [8].

The computing paradigms discussed above (except for aspect-oriented software development) are presented in Figure 4.1, reflecting their evolution over time.

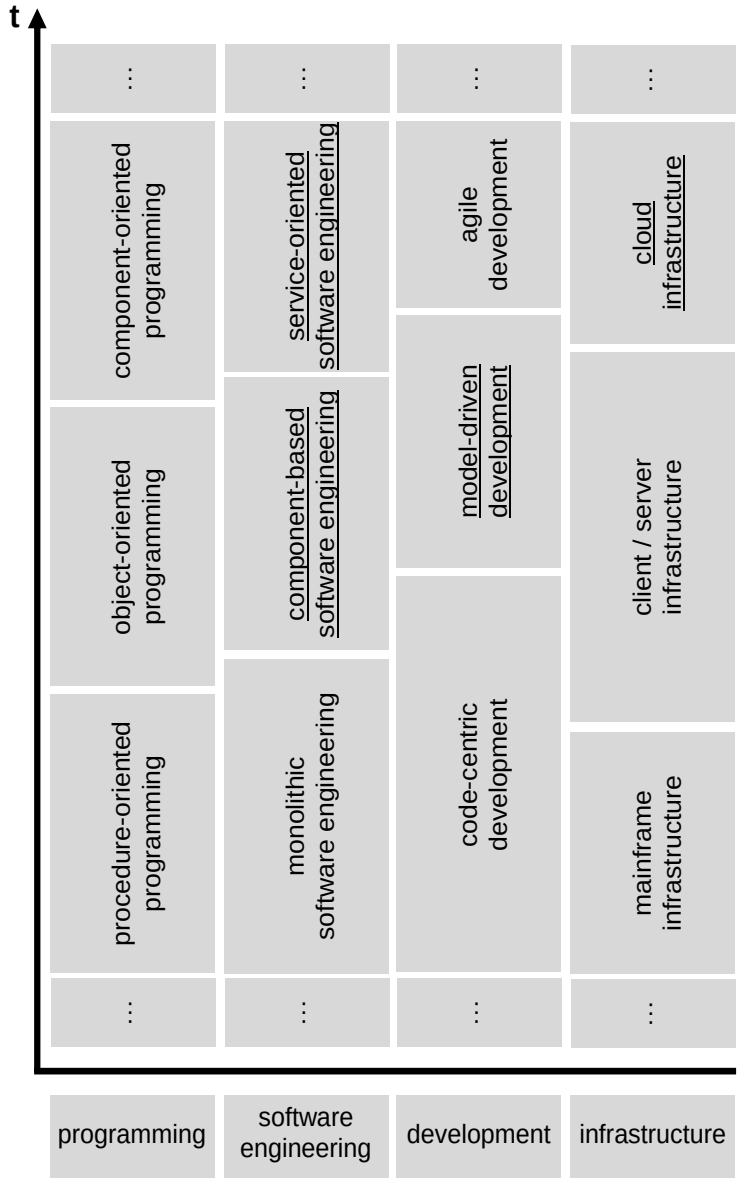


Fig. 4.1. Computing paradigms – evolution over time.

As it is seen on the figure and as discussed already, over time: programming's evolution comes through procedure-orientation, object-orientation, and component-orientation; software engineering's evolution comes through monolithicity, component-centricity, and service-orientation; development's evolution comes through code-centricity, model-centricity, and agility; infrastructure's evolution comes through mainframe solutions, client/server solutions, and cloud solutions. As it is seen as well on the figure: time-wise, the 'evolution patterns' differ from category to category, for example, the step forward from monolithic software engineering to component-based software engineering is preceded by the step forward from procedure-oriented programming to object-oriented programming. Nonetheless, those representations in Figure 4.1 are schematic and not numerically precise. Further, those 'transitions' are claimed to be viewed differently by different members of the Software Community and hence, there is no wide agreement on when exactly object-oriented programming has become 'predominant' compared to procedure-oriented programming, when exactly service-oriented engineering 'replaced' component-based software engineering as the preferred software engineering paradigm in the Software Community, and so on. Finally, we claim that most often one would observe overlaps and/or mixtures among paradigms, for example: why not claiming that both component-based and service-oriented solutions were predominant in a particular period, or why not claiming that some software applications have modules implemented using object-oriented languages and also modules implemented procedure-oriented languages? Hence, that representation mainly reflects the subjective views of the authors and is not claimed to be exhaustive.

Next to that, due to the limited scope of the current chapter, we are unable to consider all mentioned paradigms in more detail. Still, we have selected several of them for further consideration – the ones whose labels are underlined in the figure: component-based software engineering, service-oriented software engineering, and cloud infrastructures, and we will use more 'popular' labels for them, respectively:

- **`component-based development`** (meaning '*software development*');
- **`service-oriented architecture`** (meaning reference to '*software engineering*');
- **`model-driven engineering`** (meaning '*development*');
- **`mobility`** (meaning based on a *cloud infrastructure*),

plus the one not reflected in the figure, namely:

- **`aspect-oriented software development`**.

Thus, all those terms: engineering, development, architecture, are de facto largely overlapping, and we are not entering such a terminology discussion in the current paper. The terms used in Figure 4.1 reflect our desire to be maximum clear in mentioning different paradigms that belong to the same category. The *corresponding* terms to be used in the sections below reflect the popular labels that would be recognized by the wide audience.

And in the end, why exactly those paradigms and not other ones reflected in the figure will be elaborated? The bottom line is the relevance to *EIS* in general and the *enterprise-modeling-driven software generation*, in particular. **`Business coMponents`** have been considered in the previous chapters as a desired basis for

specifying software. For this reason, in our computing paradigms consideration, we would emphasize those paradigms that are relevant to the **component-based enterprise-software alignment**. This brings us to **components** (*component-based development*) and **services** (*service-oriented architecture*) that are claimed to be useful relevant units of re-use. Further, we would emphasize on **model-driven** *engineering* because we believe that only way to bring those two Worlds together (*enterprises* and *software*) is through corresponding *models*. Finally, we would emphasize on **mobility** and **non-functional crosscutting concerns** because we claim that they have essential importance for any current EIS and thus have to be explicitly considered and reflected in the specification of software.

For this reason, in the sections that follow we will consider: *component-based development*, *service-oriented architecture*, *model-driven engineering*, *mobility (emphasizing on cloud computing)*, and *aspect-oriented software development*.

4.1 Component-Based Development

The **Component-Based Development** (**CBD**) is considered to be a promising paradigm that addresses the design and development of ICT *applications*, and is founded on the principles of *object-orientation* [54] – *object orientation* (characterized by the fundamental concepts of *encapsulation*, *classification*, *inheritance*, and *polymorphism*) that was briefly discussed already, is widely recognized as a special approach to the construction of models of complex *systems*, in which a *system* consists of a large number of *objects*. Hence, **components** are essential with regard to *CBD* – if re-usable *components* are identified, they can be used many times for designing different *applications*. Next to that, *CBD* seems beneficial for the *application* design itself. By basing *application* development on encapsulated, individually definable, re-usable, replaceable, interoperable and testable (*software*) *components*, developers can build *applications* which possess durable configuration and a high degree of flexibility and maintainability. The process of *application* development would also be improved because building new *applications* would include using already developed *components*. This reduces development time and improves reliability. The performance and maintenance of developed *applications* would be enhanced because changes could occur in the implementation of any *component* without affecting the entire *application*. All this makes *CBD* reliable and effective.

All this justifies further the claim that *business coMponents* can be useful as basis for specifying *component-based applications* (see Chapter 2). By basing the design of *applications* on *software components* derived in turn from *business coMponents*, the *application* support to *business processes* can be improved considerably [54].

Hence, CBD has strengths reaching beyond the *application* development itself – the *component-based application development* can as well usefully support the *enterprise-modeling*-driven generation of *software*.

The idea of constructing modular software *systems* dates back to 1968, as according to Stojanovic [72], and referring to two complexity-avoidance approaches of that time is important, they are: '*buy before build*' and '*re-use before buy*'. This way of thinking

is considered to be an essential bottom line with regard to current *CBD* and this was even before the ideas of *object-orientation* (see above) appeared. Hence, during the 1990's, *CBD* has established itself as a natural extension and an evolution of *object-orientation*. *Components* have first been introduced at the implementation level for fast building a graphical interface using *visual basic eXtensions* controls and then there have been the *Component Object Model* of *Microsoft*, the *CORBA* components, and *Enterprise Java Beans components* – all of them proposed as standard *component-based* implementation solutions. This has contributed to a shift of emphasis from developing small, centralized, monolithic *systems* to developing complex *systems* consisting of functional units deployed over nodes of the Web and two key concepts have emerged, namely: (i) *components as large-grain building blocks of a system* and (ii) *architectures and frameworks as blueprints of the system describing its main building blocks and the way of composing them into a coherent whole* [72]. That conceptual evolution has been reflected in several widely popular *component* **definitions**:

- According to Szyperski [73], *a software component is a unit of composition with contractually specified interfaces and explicit context dependencies; a software component can be deployed independently and is subject to composition by a third party*.
- According to Lewandowski [41], *a component is defined as the smallest self-managing, independent, and useful part of a system that works in multiple environments*.
- According to Stahl et al. [66], *a component is a self-contained piece of software with clearly defined interfaces and explicitly declared context dependencies*.

We argue that those definitions further justify *Definition 13* and *Definition 14* (see Chapter 2), and also the way of looking at a *software component* from two perspectives, namely taking a *constructional* view and taking a *functional* view:

- CONSTRUCTIONALLY, software components are *implemented pieces of software*, which represent *parts of an ICT application*, and which *collaborate among each other* driven by the *goal* of *realizing the functionality of the application*.
- FUNCTIONALLY, a software component is a *part of an ICT application*, which is *self-contained*, *customizable*, and *composable*, possessing a *clearly defined function and interfaces* to the other parts of the application, and which also *can be deployed independently*.

It is to be noted however that even though all above definitions suggest essentially the same view on *software components*, they differ with regard to the perspective taken. What is to be taken into account in the current chapter is the explicit *EIS* focus we are following, and this assumes that: (i) software is specified based on *business coMponents* (see Chapter 2); (ii) software is delivered mainly in terms of ICT *applications*.

Hence, we summarize what we consider essential with regard to *software components*, taking into account the above-stated perspective:

- ✓ **a software component is an implemented piece of software**;
- ✓ **a software component is a part of an ICT application**;
- ✓ **a software component is self-contained**;
- ✓ **a software component possesses a clearly defined function and goal (in context)**;
- ✓ **a software component possesses clearly defined interfaces to the other parts of the ICT application**;
- ✓ **a software component can be deployed independently**;
- ✓ **a software component can work in multiple ICT applications and in multiple environments**.

Hence, establishing the way the *component* notion and the *object* notion relate to each other is important, and for that we refer to the studies of Stojanovic [72] where *components* are considered as *larger-grained objects* that are deployed and as such they would 'reveal' one or more *classes* 'inside'. It is thus concluded that **granularity** is the main issue in distinguishing *components* and *objects*. Further, if *objects* are identifiable *instances* of *classes*, then *component instances* (representing programming language *objects*) are *instances* of *component types*. Hence, *components* have much in common with *classes*. Nevertheless, there are some significant differences:

- *classes* represent logical abstractions while *components* represent physical things;
- *components* represent the physical packaging of otherwise logical elements and are at a different level of abstraction than *classes*;
- *classes* may have *attributes* and *operations* accessible directly, in general, *components* have operations that are reachable only through *component* interfaces.

Therefore, a *component* is a physical thing that conforms to and realizes a set of interfaces. Internally, a *component* may be implemented by a single class, by multiple classes, or even by traditional procedures in a *procedure-oriented programming language*.

For this reason, an explicit discussion is necessary on *component* **interfaces**:

As already suggested, a *component* is an encapsulated unit with a completely hidden behavior behind an *interface*. As studied by Stojanovic [72], the *interface* provides and explicit separation between the *outside* and the *inside* of a *component*, by:

- answering the question *WHAT* – What useful services are provided by the *component* to the *context* of its existence?
- not answering the question *HOW* – How are those service actually realized?

We relate that to the *black-box* and *white-box* perspectives, respectively, as discussed already (see Figure 2.9). A precisely defined *interface* allows for using the behavior (services) delivered by the *component* without knowing how that behavior is actually realized. Said otherwise, the *component* 'interior' remains hidden (and not important) for the *component's environment* as long as the *component* provides services, following the constraints defined by its contractual *interface* – it is often that the *interface* reflects the only information that shows the *component's* 'user' that the *component* actually does.

An *interface* is defined by Szyperski [73] as *a named collection of operations that are used to specify a service of a class or a component*, hence defining a *component interface* as *a specification of the component's access point*.

Thus, if a *component* has multiple access points, each of which represents a different service offered by the *component*, then the *component* would be expected to have multiple *interfaces*.

Further, an *interface* offers no implementation of any of its operations; instead, it merely names a collection of operations and provides their descriptions – it is hence possible to replace the implementation part without changing the *interface* [72]. Following Stojanovic further:

- a PROVIDED *interface* points to the services and operations that the *component* provides to its *environment*, in realizing its *function*;
- a REQUIRED *interface* specifies the services and operations that the *component* requires from its *environment*, in order to realize its *function*.

According to [80], any *interface* would have four *attributes*:

- ***name*** (each *component interface* is to have a unique name);
- ***keys*** (they are based on the search record definition of the *component*);
- ***properties*** (they relate to the record fields of the *component*);
- ***methods*** (a method is like a function that can perform a specific task according to corresponding requirements).

Finally, we claim the following: FIRSTLY, in order to make an interoperable *component* feasible, it is necessary to consider a corresponding **`component implementation model`** and in Sub-section 4.1.1, we present three popular and widely accepted *component implementation models*, namely the *Microsoft Component Model*, the *Enterprise Java Beans Model*, and the *CORBA Component Model*, as according to Stojanovic [72]. SECONDLY, with implementation technology not being sufficient by itself for adequately developing *component-based applications*, methods and approaches are needed for establishing how to reflect business requirements in the design and development of such *applications* – this we refer to as **`component-based development methods`** and in Sub-section 4.1.2, we present three popular and widely considered *component-based development methods*, namely the *Rational Unified Process*, *KobrA*, and *Catalysis*, as according to Shishkov [54].

4.1.1 Component Implementation Models

In the current sub-section, we will consider firstly the *Microsoft Component Model*, secondly – the *Enterprise Java Beans Model*, and thirdly – the *CORBA Component Model*.

Microsoft Component Model

The **`Component Object Model`** or **`COM`** for short, is a *language-independent*, *binary component standard* [81] whose core concepts include:

- a binary standard for function calling between *components*;
- the typed grouping of functions into *interfaces*;
- a base *interface* providing mechanism for (i) other *components* to dynamically discover the *interfaces* implemented by a *component* and (ii) a reference counter, allowing *components* to track their own 'lifetime' and delete themselves when appropriate;
- a globally unique identifier mechanism for *components* and their *interfaces*;
- a *component* loader to set up and manage *component* interactions.

COM provides as well mechanisms for shared memory management between *components* and also error and status reporting. In *COM*, an *interface* is represented as a pointer to an *interface node* and in turn, the *interface node* contains a pointer to a table of operation variables and those variables in turn point to the actual implementation of the operations.

Enterprise Java Beans Component Model

The **Enterprise Java Beans Component Model** or **EJB** for short, is a *server-side component* model for the development of *applications* in the programing language *Java* [26], where a *component* is called an *enterprise bean*. Further, there are two kinds of *enterprise beans*:

- *session enterprise beans* (those are transient *components* that exist only during a single client/server session);
- *entity enterprise beans* (those are persistent *components* that control permanent data kept in permanent data stores, such as databases).

Moreover, an *enterprise bean* resides inside a *container* with a *container* in turn consisting of a deployment environment for *enterprise beans*. Next to that, the *container* provides a number of services for each *enterprise bean*, such as lifecycle management, state management, transaction management, and so on. Finally, an *EJB* server provides a runtime environment for one or more *containers*.

Finally, the client *application* interacts with the *enterprise bean*, by using two *interface types* that are generated by the *container*, namely: (i) *home interface* (it can be used by clients to create, destroy or find an existing *enterprise bean instance*); (ii) *object interface* (it provides access to the application methods of the *enterprise bean*).

CORBA Component Model

The **CORBA Component Model** or **CCM** for short, is a *server-side component* model extending the CORBA core *object* model with a deployment model; *CCM* is as well providing a higher level of abstraction for CORBA and *object* services; the two major advances introduced by the *CCM* are a *component* model and a runtime *environment* model; a *component* is an extension and specialization of a CORBA *object* [11]. As for the model of a CORBA component type:

- Any CORBA *component* is denoted by a *component* reference.
- CORBA *components* support a variety of surface features, called ports, through which clients and other elements of an *application environment* may interact with those *components*.

This is presented on Figure 4.2:

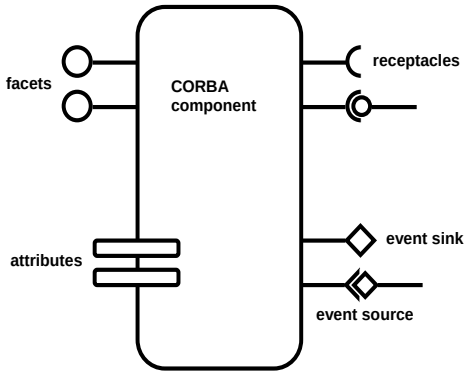


Fig. 4.2. CORBA component.

As seen from the figure, there are five different kinds of *ports*:

- *facets* – they are *interfaces* provided by the *component* for client interaction;
- *receptacles* – they are connection points that describe the *interfaces* used by the *component*;
- *event sources* – they are connection points that emit events of a specified type to interested event consumers;
- *event sinks* – they are connection points into which events of a specified type are announced;
- *attributes* – they are named values primarily used for *component* configuration.

Further, a *component* may have multiple *facets*, *receptacles*, *event sources*, *event sinks*, and *attributes*.

Finally, there are four categories of *components*, as studied by Stojanovic [72]:

- *service components* – they are stateless, have no identity, and support a single invocation per instance;
- *session components* – they have a transient state, have no persistent identity, and support more than one invocations per instance;
- *process components* – they have an explicitly declared state that is managed by the runtime environment, have an identity managed by the client, and have a behavior that may be transactional;
- *entity components* – they are similar to process components, except for their identity which is visible to the client but managed by the runtime environment.

In summary, in the current sub-section we have briefly presented three popular *component implementation models*; in the following sub-section, we will consider three popular *component-based development methods*, as already mentioned.

4.1.2 Component-Based Development Methods

In the current sub-section, we will consider firstly the *Rational Unified Process*, secondly – *KobrA*, and thirdly – *Catalysis*.

Rational Unified Process

The **Rational Unified Process** or **RUP** for short, is not only the development process usually applied with *UML* (the *Unified Modeling Language*) but also a useful development method (process) as far as *component-based development* is concerned, which method covers the entire software development life-cycle [38].

The key *RUP* concept is the definition of *activities*, called *workflows*, throughout the development life-cycle, such as requirements elicitation, analysis, design, implementation, and testing. Unlike the classical waterfall process, those *activities* can be overlapping and performed in parallel [72]. Within each of the *activities*, there are well-defined stages of inception, elaboration, and transition. A support to *component-based development* is encouraged even though that support is just declarative and implicit, being directed towards physical packaging, as it can be seen from the RUP's defining a *component* as '*a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture, and that conforms to and provides the physical realization of a set of interfaces*'. Finally, one of the main advantages of *RUP* is that it provides an opportunity for iterative and incremental *system* development, which is seen as the best development practice [72].

KobrA

Our analysis on **KobrA** has been supported mainly by the following two sources: [4,5]. Interested readers could find there information about all concepts related to *KobrA*, which have not been considered in the current sub-section.

The *KobrA* method is a state-of-the-art approach to *component-based* product-line engineering with *UML*. Among the key characteristics of *KobrA* are: architecture-centricity; systematic *COTS component* re-use; integrated quality assurance. The major strengths of *KobrA* are its overall consistency, the embracement of the *component* concept in all phases of the software life-cycle, and the UML-based graphical specification of *components*. The main limitation is that there are no clear guidelines how to relate the specification of software to a prior *enterprise* analysis and modeling.

A complementary workbench has been developed to support the use of the *KobrA* method in conjunction with commercial *CASE* tools. A test bed for the approach has been provided in the domain of *enterprise resource planning*.

KobrA is conceptually based on the foundation of *product-line engineering*. Hence, before proceeding further, we would briefly introduce it. *Product-line engineering* is an inherent part of the *KobrA* method. When pursuing a *product-line* approach in *KobrA*, the overall software life cycle consists of two basic *product line engineering* activities:

- *Framework engineering*. It applies the *komponent* (*komponent* means *component* as seen from the perspective of the method *KobrA*) modeling and implementation activities, accompanied by additional sub-activities for handling variabilities and decision models, to support a family of similar *applications* (i.e. development for reuse). A framework therefore contains a generic *komponent* tree that captures the common and variable characteristics of a *product-line*.
- *Application engineering*. It uses the framework developed during framework engineering to build particular *applications*. Since one of the goals of *application* engineering is to remove the variabilities in the framework, and resolve the decisions in the decision model, *komponent* containment trees for *applications* are very similar to those for a single *system*. The only difference is that *komponents* are accompanied by a decision model instance, which captures the decisions made in resolving the decision model for a particular *komponent*.

Based on the (above outlined) brief information about *KobrA*, we will come (below) through some basic principles and issues characterizing the method.

A core principle of *KobrA* is the strict and systematic separation of concerns, so that at all times during a development project developers are aware of what they should be attempting to do and what *concern* they are working on. A manifestation of this principle in *KobrA* is in the separation of the *product* from the *process* (contrary to methods which arbitrarily mix the description of *what* engineers should be trying to produce with the definition of *how* they should produce). Another fundamental separation of concerns in *KobrA* is the organization of the method in terms of three orthogonal dimensions of development: one dealing with the *level of abstraction*, one dealing with the *level of genericity*, and one dealing with *composition*.

At the largest level of *granularity*, the *product-line* paradigm takes precedence in *KobrA*. This *splits the overall development cycle into two parts*: (i) one dealing with the development of a framework – a re-usable set of software artefacts whose core is embedded within all products developed by the *enterprise*; (ii) another one concerned with the development of an application – a concrete instance of the framework, adapted and extended to meet the needs of a specific customer.

At the intermediate level of *granularity*, *KobrA* is driven by the *component* paradigm. *KobrA* frameworks and *applications* are all organized in terms of hierarchies of *components*. However, the *components* in *KobrA* represent the *logical* building blocks of a software *system* (not *physical components*, as in *CORBA* – see above).

A central goal of *KobrA* is to enable the full expressive power of the *UML* to be used in the modeling of *components*. To this end, the use of the *UML* in *KobrA* is driven by four basic principles:

- *Uniformity*. Every behavior-rich entity is treated as a *komponent*, and every *komponent* is treated uniformly, regardless of its *granularity* or location in the containment tree.
- *Encapsulation*. The description of *what* a software unit does is separated from the description of *how* it does it.
- *Locality*. All descriptive artifacts represent the properties of a *komponent* from a *local perspective* rather than a global perspective.

- *Parsimony*. Every descriptive artifact should have '*just enough*' information, no more and no less.

As for the life-cycle of a *KobrA*, at the highest level of *granularity*, this life-cycle is composed of a sequence of phases in which new versions of the central framework are developed and new applications are instantiated from it to meet the expectations of new customers.

In summary, the strict *separation of concerns* makes *KobrA* compatible with a large number of practical implementation and middleware technologies. Its embracing the *component* paradigm allows for adequately benefiting from re-use possibilities. Its being soundly founded on the principles of the *product-line engineering* provides a good theoretical foundation. Its consistency with *UML* results in a specification of software, which is fully in tune with the current software design standards.

We outline as limitation nonetheless, the way *KobrA* is addressing the very early software specification tasks and in particular - the relation to the original *enterprise system* that is to be supported by the software-to-be. As mentioned before, *there are no clear guidelines how to relate the specification of software to a prior enterprise analysis and modeling*. This could be improved either by extending *KobrA* backwards (towards a consideration of very early *enterprise* modeling activities) or by a combination with a business process modeling tool.

Catalysis

Our analysis on **`Catalysis`** has been supported mainly by the following two source: [24]. Interested readers could find there information about all concepts related to *Catalysis*, which have not been considered in this sub-section.

Catalysis is a method for *component-based* and *object-oriented* software development, which provides a strongly coherent set of techniques for *enterprise* analysis (characterized by unambiguity about requirements) and *system* development using *UML* as well as a coherent method for *object-oriented* analysis and design. *Catalysis* provides also well-defined consistency rules across models and powerful mechanisms for composing different views to describe complex *systems*.

Catalysis is specifically targeted as a method for *component-based* development, in which families of products are assembled from kits of *components*. The method also allows for re-use of other artefacts of the design process, such as frameworks of collaboration between *objects*.

Catalysis includes techniques to *map* between (*UML*-based) *system* design and an analysis model. The gap and inconsistencies are reduced by:

- unambiguous *interface* specification;
- techniques to define powerful *component* 'connectors' abstracting above the level of *object-oriented* messages;
- 'retrieval' techniques for relating the differing models that different *components* (especially bought-in or *legacy components*) usually have (this might include, for example, different notions of what a customer is).

Use-cases [54] have a central role in *Catalysis*; they are applied at different abstract levels. With each decomposition, the *objects* interact to fulfil the goals of the more abstract *use cases*.

The *Catalysis* method basically comes through the following *phases*:

- A model of the domain is produced, specifying first, what *objects* are there and second, the goals which are associated with the major *use-cases*.
- Scenarios are drawn on how (certain) *component* could help realizing the major *use-cases*, breaking them down into individual steps.
- Viewing a *component* as a specification (this would be possible because at this stage it is to be known what a *component* is supposed to do). The *component* has some defined responsibilities, and defined collaborations with the actors around it.
- *Component's* responsibilities are distributed between *objects* inside it and also, interactions between *components* are defined (*use cases* are used for that goal). It is possible (if necessary) defining generic interactions between *components*, so that they are made 'pluggable'. This is done through template models.

Thus, essential *characteristics* of the *Catalysis* method are:

- Usability of generic chunks of software with robust, well-defined *interfaces*. Dynamic coupling of *components* is just one form of re-use. Other forms include the import of a generic chunk of design into many other designs. In this sense, a '*component*' can include any piece of development work (code, models, rules, design patterns, and so on).
- Issues which concern the *inter-component connections* - 'connectors' play a significant role in this task. They are specified independently on the specification on (relevant) *components*. Just like *objects*, connectors are encapsulated: the specification of what one achieves is independent of its implementation.
- Software development evolving firstly through the rapid assembly of end products from *components* and secondly – through the development of high-quality *components*.

In *Catalysis*, there are particular *validation mechanisms*. The validation suite is a set of *ancillary components* for two purposes: (i) some of them test a *component* once it is installed in a particular *context*, to ensure it is running properly; (ii) others are test versions of *components,* exercising the *components* they are connected to, to make sure they behave as required.

According to Shishkov [54], *Catalysis* has certain *limitations*, particularly as it concerns the proper alignment between *enterprise* modeling and software specification since:

- the method does not offer a solid mechanism for the reflection of the original business requirements in the specification of the software functionality - that is because *Catalysis* is not rooted in any way in any *social theory*, that would have allowed for a better grasp of real-life aspects;
- *Catalysis* is insufficiently focused as it concerns *re-use*, considering for *re-use* not only *components* but also pieces of code, rules, and so on – this would assume thorough multi-perspective *re-use* guidelines and such guidelines are not available;
- *Catalysis* is insufficiently capable of grasping human-to-human communication, similarly to *KobrA*.

In summary, we have considered *CBD*, touching upon its main characteristics, the *component* notion, *component* implementation models, and *component-based* development methods. In the following section, we will consider *service-orientation*.

4.2 Service-Oriented Architecture

The **Service-Oriented Architecture** (**SOA**) is considered to be a promising paradigm building upon *CBD*, which shifts the focus from the operation of a software *component* to the **service** the *component* is delivering to its user(s) [64].

Our analysis on *SOA* has been supported mainly by the following source: [76].

SOA was originally motivated by the need of *enterprises* to better match *information systems* with their business goals, combined with the market trend of more and more flexible cross-organizational collaboration between *enterprises* [49]. Vertical integration (*business-IT alignment*) and horizontal integration (*IT supported cross-organizational collaboration*) are considered crucial for modern *enterprises*, but traditional IT architectures have serious integration deficiencies. Architectures often comprise monolithic (silo) *applications* that are effective for the specific purpose they were created, but which do not allow integration without custom coded connections. Architectures with *component-based applications* provide units of business logic, which ease the definition of connections, but still require that the flow of control and the transformation of data formats are bound into the business logic.

SOA is an *IT architectural style* that tries to achieve integration by way of **defining composite applications as an orchestration of services**, with *services* potentially offered by different organizations. A *service* externalizes public functions of an *application* that implements a repeatable business task. Since a composite *application* can also be offered as a *service*, integration may involve *multiple levels of composition*, and a *service* can be internal to an organization or cross-organizational.

Those issues will be addressed in this section, by: (i) surveying (in Sub-section 4.2.1) the concepts and architectural elements of *SOA*; (ii) briefly discussing (in Sub-section 4.2.2) *web services* that constitute one of the widely adopted technologies to implement *SOA*.

4.2.1 SOA Foundations

The central concept of *SOA* – the *service* concept, has several interpretations, partly due to the fact that *SOA* addresses two distinct disciplines, namely *enterprise engineering* and *software engineering*, and each of those two disciplines has been considering the *service* notion in its own perspective:

- in an *enterprise context*, a *service* involves the exchange of some action, performance or promise for value between a client and a provider [64]. Examples are transportation *services*, health *services*, education *services*, outsourcing *services*, and helpdesk *services*;

- in an *IT context*, a *service* refers to the external behavior of an IT *system*, as can be observed and experienced by the users of that *system* [77]. Examples are data communication *services* and application *services*.

For convenience, we will use the terms *business service* and *IT service* to distinguish between the *enterprise* view and the IT view on *services*.

SOA holds the promise to bring business and IT together, by repeated aggregation of *IT services* into composite *applications* supporting *business services* that in turn are aggregated into *business processes* [75]. Figure 4.3 shows **the basic architectural pattern that underlies SOA**. In this pattern, three roles are distinguished: *service provider*, *service broker* and *service requestor* [50]. A *service provider* offers one or more *services*, which may be implemented using arbitrary technologies and involving backend *systems* protected by a firewall. Each *service* has well-defined *interfaces* referred to in a *service* description. *Service* descriptions may be published with a *service broker*, thus opening the possibility for *service requestors* to find *services* by providing required *service* properties to the *service broker*. The *service broker* searches for *service* descriptions that satisfy the required *service* properties, and the *service requestor* can select from the result of this search. Based on the location/access details in the *service* description, the *service requestor* can then bind to a *service provider* that offers the selected *service*. After a successful binding, the *service requestor* can invoke the *service*, according to the *interface* details in the *service* description.

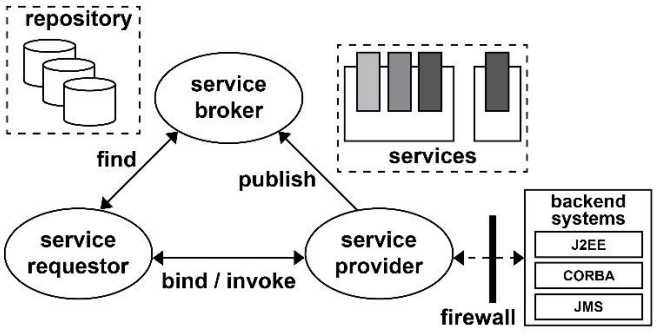


Fig. 4.3. The basic SOA pattern.

Using this pattern, *vertical integration* is tackled by presenting a *service* as a virtual *component* that can be implemented by alternative concrete *components* using different technologies. The *service requestor* is therefore decoupled from the implementation concerns of the *service provider*. Using *SOA* for *application* design and proving a *service* wrapping for legacy *applications* thus presents a viable approach to *enterprise application integration*.

Vertical integration, or *business-to-business integration*, requires that each potential business partner defines a public view on its private process, with corresponding

services and associated incoming and outgoing message exchanges that allow linking to external partners. The previously presented basic *SOA* pattern only shows a single *service provider* and a single *service requestor* role. In a *business-to-business* collaboration scenario, business partners may play either role for any number of supported *services*. An individual partner coordinates the *services* used and provided through its private process. Since this in general does not determine the overall coordination involving all partners, a **coordination protocol** can be defined that concerns the public view on how the partners should work together. Such a *coordination protocol* does not provide a concrete and executable process for the coordination of a service. It only defines the order in which messages should be exchanged, where messages are used to invoke a *service* or return a *service* result in accordance to a *service* provided by one of the partners. A definition at this level of abstraction is also referred to as *service* **`choreography`**, see Figure 4.4 (up):

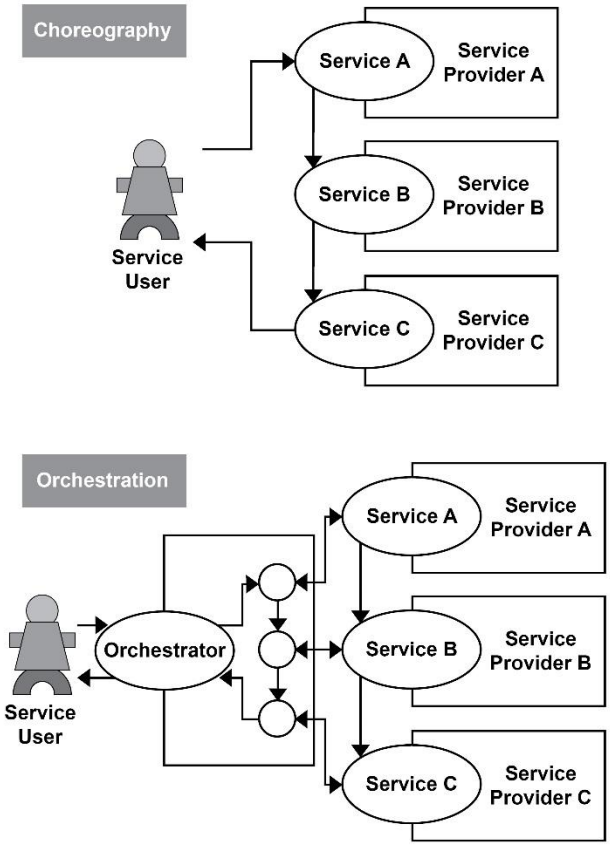


Fig. 4.4. Service choreography and service orchestration.

Said otherwise, the *choreography* reflects the collaboration among different *services*. *Services* participating in the *choreography* may belong to different providers; the aim is that the participating *services* collaborate to implement a *business process* [30]. In Figure 4.4 (up), the *business process* consists (for example) of three different *services*. The *service user* triggers the *business process*, by invoking *Service A* with a request. *Service A* processes the *user* request and then invokes *Service B*. *Service B* processes the request from *Service A* and then invokes *Service C*. *Service C* processes the request from *Service B* and then sends the result to the *service user*.

It is to be noted that we use the term SERVICE REQUESTOR in Figure 4.3 and we use the term SERVICE USER in Figure 4.4. Those terms are not conflicting and we use different terms because both figures mentioned above reflect a simplified view on reality. In Figure 4.3, we recognize a *service requestor*, emphasizing on the role of formulating a request, searching for candidate *services*, making a selection, and binding to a corresponding *service provider*. We abstract from the fact that the same entity requesting the *service* is then the *service user*. In Figure 4.4, we abstract from the request formulation, *service* discovery, and so on, emphasizing on the role of using the selected *service(s)*.

It is to be noted also that in Chapter 2, a *business process* is defined as 'a structure of (connected) *transactions* that are executed in order to fulfil a *starting transaction*' (*Definition 6*) while what we discuss above concerns a structure of (connected) *services* that are executed in order to fulfill a *'starting' service*. How would then the *transaction* and *service* concepts relate to each other and how would the *business process* and *choreography* concepts relate to each other? Answering this question is considered challenging because of the following reasons:

- The notion of *transaction* is not only grounded in *enterprise engineering* but is also reflected in a pattern (Figure 3.4) while the notion of *service* addresses two distinct disciplines – *enterprise engineering* and *software engineering*, as mentioned above, leading to different interpretations.
- Within a *business process* as in line with *Definition 6*, a *starting transaction* is triggered and possibly, in order for it to be executed, it is necessary that another *transaction* is triggered, and this is done by the *executor* (*producer*) of the *starting transaction* – it is the *executor* who *initiates* the second *transaction*, and the *executor* of the second *transaction* (in turn) might need to *initiate* a third *transaction*, and so on. Then, each result is delivered to the corresponding *transaction initiator* which means that the result of the second *transaction* would be delivered to the *executor* of the *starting transaction* who in turn would deliver the final result to the *customer (user)*. In contrast, the collaboration among *services*, as presented above, is not that elaborate as the collaboration among *transactions* since we go as far as establishing that the *starting service invokes* another *service* which in turn *invokes* yet another *service*, and so on. Further, when we consider a collaboration among *transactions* part of a *business process*, it is the *starting transaction* that delivers the result to the *customer* while in the *service choreography*, it is the last *service* being *invoked* that delivers the result to the *customer*, as illustrated above.

For this reason, we allow ourselves to use the term *business process* in the *service choreography* context only under the condition that we make it explicit that even

though similarities can be found, a '*choreography of services*' is not the same as a '*business process of transactions*'.

What we consider conceptually closer to transactions-driven *business processes* is *service* **`orchestration`** – see Figure 4.4 (down), assuming that the overall coordination (concerning the collaborative behavior of different *services*) is assigned to and executed in a centralized way by some computing node [76].

As in *service choreography*, also in *service orchestration*, the *services* (participating in the *orchestration*) may belong to different providers. The difference is nonetheless that in an *orchestration*, those *services* are coordinated from a central entity, the **orchestrator**; the *orchestrator invokes* each *service* according to a given strategy. We considered a *choreography* example featuring three *services* (see Figure 4.4 (up)) and we now consider an *orchestration* example featuring the same three *services* (see Figure 4.4 (down)). As it is seen from the figure, in the *orchestration* case, *services* are coordinated by another *service*, the *composite service* (called '*orchestrator*') – this *service* defines the composition of the *services* participating in the *business process*. The *service user* triggers the *business process*, by *invoking* the *orchestrator*. Once the *orchestrator* receives the *user* request, the first action it takes is to *invoke Service A* and *Service A* would respond in turn with a message. Then (based on this response) the *orchestrator* would *invoke Service B* and *Service B* would respond in turn with a message. Then (based on this response) the *orchestrator* would *invoke Service C* and *Service C* would respond in turn with a message. Then (based on this response) the *orchestrator* would deliver the result to the *service user*. It was stated above that *service orchestration* is conceptually closer to *transactions*-driven *business processes* (compared to *service choreography*) because similarly to how a customer approaches the *executor* of a *starting transaction* and in the end the *executor* of the *starting transaction* would deliver the result to the customer (no matter how many other *transactions* the *executor* of the *starting transaction* would have (directly or indirectly) triggered in order to be able to *execute* the *starting transaction*), the *service user* approaches the *orchestrator* and in the end the *orchestrator* would deliver the result to the *service user* (no matter how many *services* the *orchestrator* would have triggered in order to be able to respond to the request of the *service user*).

In order to illustrate the patterns discussed above (the basic *SOA* pattern – Figure 4.3, the *choreography* pattern – Figure 4.4 (up), and the *orchestration* pattern – Figure 4.4 (down)), we use the following simple real-life examples:

[**Example 1**]: *Jamall Caribbean Custom Tailors* (service provider) are active in the Toronto area in Canada; they have advertised their services at *http://www.yellowpages.ca*. *John* lives in Toronto; *he has ripped his trousers* (service user) and discovers *Jamall Caribbean Custom Tailors'* services in *Yellowpages* – Canada. Then *John* would contact *Jamall Caribbean Tailors*, discussing the problem and negotiating the conditions about their fixing his trousers. Once they reach an agreement, *John* would bring his ripped trousers to the nearest collection desk of *Jamall Caribbean Custom Tailors* whose rules on handling orders would be dominant and *John* would have to adapt to the conditions of their services (for example: for how many days this order would be handled, are week-end days counted, what is the extra pay for a priority order, what are the compensations for damage on the clothing, and so on), which conditions *John* must have discussed with them during the above-mentioned negotiations. This example points to the basic SOA pattern.

[**Example 2**] *Hristo* is Bulgarian living in Sofia, Bulgaria, who has a PhD degree from Delft University of Technology in The Netherlands. Hristo is appointed as Assistant Professor at the Bulgarian Academy of Sciences and for this he needs a *legalization of his PhD degree*. He applies for this to *Delft University of Technology*, by: (i) submitting via e-mail a scanned copy of a filled in and signed form, and (ii) transferring a corresponding fee. Then:

- A representative of *Delft University of Technology* (Delft) would issue a duplicate of the diploma, send it to the *DUO* Agency of the Dutch Ministry of Education (Groningen), and pay on behalf of the university a processing fee to *DUO*.

=> SERVICE 1.

- A representative of *DUO* (Groningen) would match the information in the document to corresponding information in their databases and if all is OK, the person would apply on behalf of *DUO* a sticker at the back of the document, send the document to the *Courthouse in Groningen*, and then pay on behalf of *DUO* a processing fee to the *Court*.

=> SERVICE 2.

- A representative of the *Court* (Groningen) would check the details in the document and the details of the diploma holder in the Dutch registries, and if all is OK, the person would apply an apostille on the document and send the document to *Hristo*.

=> SERVICE 3.

This example points to service *choreography* because the coordination is realized among the services themselves: *Hristo* is triggering *Service 1* and then those who are executing *Service 1* know what to do and how to deliver it to and trigger *Service 2* and then those who are executing *Service 2* know what to do and how to deliver it and trigger *Service 3* that in turn delivers the result to *Hristo*.

[**Example 3**] *Jimmy* is the leading manager of a small company in Sofia and *Alice* is his business assistant who is authorizes to sign for Jimmy declarations, application forms, to order payments on behalf of the company, and so on. Jimmy needs a *certificate of good standing* concerning the company, and he asks *Alice* to get it for him. Then:

- *Alice* would visit a *solicitor*, asking him or her to prepare the *application letter*, and *Alice* would pay the *solicitor* for the service, on behalf of the company.

=> SERVICE 1.

- Having the *application letter* (for reference), *Alice* would go to the bank and transfer a corresponding *fee* to the *Court*.

=> SERVICE 2.

- Having the *application letter* and the *proof of payment*, *Alice* would go to the *Court*, submit those documents and immediately collect the *certificate of good standing*, if everything is OK with regard to the company.

=> SERVICE 3.

Then *Alice* would go back to *Jimmy*, giving him the *certificate of good standing*.

This example points to service *orchestration* because the coordination is realized through *Alice* who is just like the '*orchestrator*' in Figure 4.4 (down): *Jimmy* is triggering *Alice* who knows what and how to do, and in what order – *Alice* would firstly sort things out with the *solicitor*, then she would do the fee payment, and finally, she would go and collect the *certificate of good standing* at the *Court*. Based on this all, *Alice* would go back to *Jimmy* and deliver the *certificate* to him.

Even though those examples illustrate the corresponding *SOA* patterns in terms of underlying internal logic, the examples are not to be considered straightforwardly because they are reflecting real-life situations while *SOA* is an *IT architectural style*, as already mentioned.

Finally, after outlining the basic *SOA* pattern and touching upon *service coordination*, it is necessary to discuss **service composition** since often the user needs cannot be satisfied by simply using one particular *service* and *composite services* are to be considered. According to Eduardo Goncalves da Silva [30], the *service composition* is initiated by the specification of a *service* request where the *service requestor* / *user* indicates requirements and preferences for the *composite service* to be created. Following that, candidate *services* for the *service composition* are discovered in the *service registry*. In case no *services* are discovered, the requirements for the *service* may need to be re-formulated and/or refined. Following that, the discovered *services* are composed to meet the specified requirements and this may be accompanied by further interactions with the *service registry*, in case other *services* are necessary to complement the already discovered *services*; once the specified *service* requirements can be fulfilled by the created *service composition*, the resulting *service* can be executed, such that the *service requestor* / *user* makes use of it. It is also possible that the *service developer* is driving the *service composition* process – in such a case, the resulting *service composition* may be published in the *service registry* so that it can be used by other *users* or *service developers* in the future.

As it concerns the **implementation** of *SOA*, we mentioned at the beginning of the current section that we will consider (in the following sub-section) *web services* that constitute one of the widely adopted technologies to *implement SOA*.

4.2.2 Web Services

Web Services (**WS**) are a collection of emerging standards, which are widely accepted as the technology of choice for implementing *SOA* [50]. *WS* to a large extent support the concepts, patterns and principles mentioned in the previous sub-section. An application designed and implemented according to *WS* standards is self-contained and modular, has a description which can be published, can be found on basis of its description, and can be located and invoked over networks.

The core ***WS* standards** are the following:

- **Simple Object Access Protocol** (**SOAP**): this is an *Internet* protocol for *web* (*service requestor* and *service provider*) *applications* to communicate on top of other standard *Internet* protocols, including *HTTP*. *SOAP* defines how messages are structured and processed in a platform-independent way. It comprises two message exchange patterns, viz. one-way and request-response.
- **Web Service Description Language** (**WSDL**): this is the language for specifying the *WS* interfaces. It is used to provide a description of the *service* for the (potential) *service requestors*. Such a description includes information on which messages are related to each operation that is supported by the *service*, how these messages are related (e.g., operation input and output), and how *SOAP* messages are exchanged.

- **Universal Description, Discovery and Integration** (**UDDI**): this standard is defined to enable the storage of information for organizing and discovering *WS*. *UDDI* consists of data structures and APIs for publishing and querying *WS*. The *UDDI* APIs are themselves *WS*, and thus are described and can be invoked as any other *WS*.

In addition, all *WS* standards rely on the Extensible Markup Language (XML) to represent structured data. *XML* documents and schemas are defined to standardize the format and typing of data communicated by *WS*. The basic *SOA* pattern (see Figure 4.3) can be supported with *SOAP*, *WSDL* and *UDDI*. Those standards are, nevertheless, insufficient to correlate messages exchanged between a *service requestor* and a *service provider*, to distinguish between multiple instances of the same *service*, or to coordinate the use of different *services*. Also they do not address policies that govern the use of *WS*, non-functional aspects of *WS* such as reliability, security and atomicity. For this purpose, several other *WS* standards have been developed. Figure 4.5 shows an overview of standards supporting different aspects of *SOA*, as according to Van Sinderen [76].

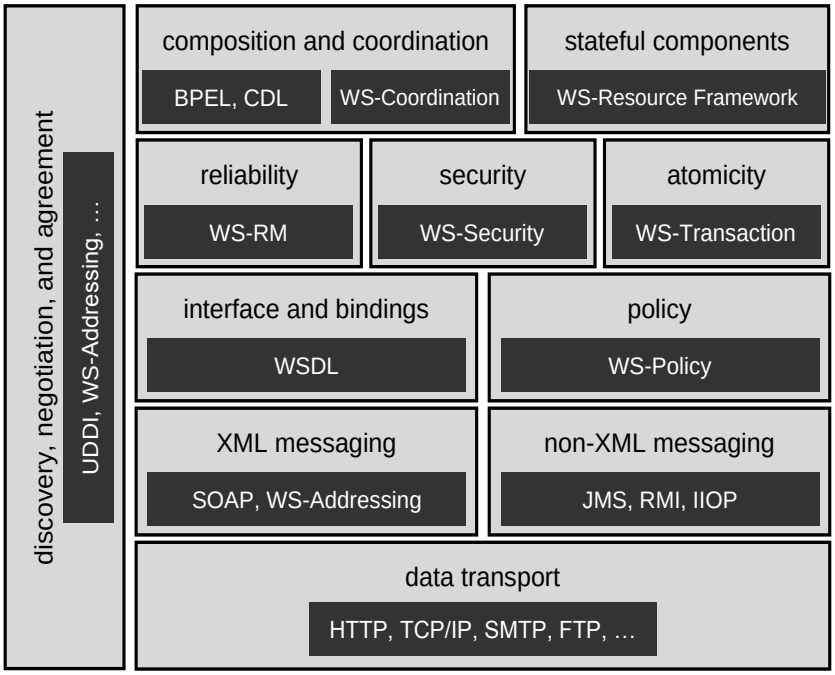


Fig. 4.5. WS and some other standards supporting SOA.

We argue that those standards have reached a certain level of technical maturity and thus represent an adequate *WS* basis with regard to the implementation of *SOA*. This in turn reflects promising, in our view, developments based on *CBD* (see the previous section), such that COMPONENTS are considered useful UNITS OF DEVELOPMENT while SERVICES are considered useful UNITS OF UTILIZATION with regard to developing (distributed) software and making it available to users. Complementing this, we will consider (in the following section) *model-driven engineering*, featuring the development process itself, no matter if this concerns *component-based development of software applications* or *composition of services* for the sake of generating software-based solutions.

4.3 Model-Driven Engineering

Any subject using a *system A* that is neither directly or indirectly interacting with a *system B*, to obtain information about the *system B* is using *A* as a **`model`** for *B*, according to *Definition 9*. In reflecting that definition in real life, we establish that the human mind would often 're-work' reality, simplifying things, driven by an intuitive 'push' to identify similarities among *objects*, emphasizing those similarities in perceiving different *objects*. For example, both the small *Mitsubishi Colt* and the big *Cadillac Eldorado* are intuitively matched to the '*car*' *model* by a person, firstly, and the huge differences among those two *objects* go on second place. Said otherwise, upon perception, a person would firstly try to relate the observed *object(s)* to a category item already existing in his or her mind, abstracting from very many details. **`Abstraction`** (pointing to the capability of finding the commonality in many different observations) is hence essential with regard to how people perceive reality and reason about it – people often generalize specific features of real objects (**`generalization`**), classify the objects into coherent clusters (**`classification`**), and aggregate objects into more complex ones (**`aggregation`**). Thus *abstraction* reflects the natural human behavior in real life while in science, ABSTRACTION RELATES TO MODELING, as suggested by the above definition. Hence, a *model* is a simplified and/or partial representation of reality. Models are of importance in many scientific disciplines, such as *physics* and *chemistry*, for example, where through simplified *models* of natural phenomena, one would draw conclusions about the phenomena themselves. In this, one would aim either at reflecting (through *modeling*) just a selection of relevant properties, hence reducing complexity or at considering the features of an individual for the sake of *generalization*. Further, *models* can be used to describe reality, to determine the scope and details at which to study a problem, and so on. Through *modeling*, features of products can be analyzed and discussed before the corresponding products get produced. Finally, with us focusing on the development of software artefacts in this chapter, we would consider particularly **`model-driven engineering`**, by which we mean model-driven *software development*. According to [9], the need for *model-driven engineering* is justified taking into account the following facts:

- Software artefacts are becoming more and more complex, and therefore they need to be discussed at different *abstraction levels*, depending on the profile of the involved stakeholders, phase of development, and objectives of the work.

- Software is more and more pervasive in real life, and the expectation is that the need for new pieces of software or the evolution of existing ones will be continuously increasing.
- Software development is not a self-standing activity: it often imposes interactions with non-developers (e.g., customers, managers, business stakeholders, and so on) which need some mediation in the description of the technical aspects of development.

For this reason, it is not surprising that by applying *model-driven engineering*, software developers increase efficiency and effectiveness [9]. This nonetheless does not assume just using *models* and corresponding notations, for example *UML*; in *model-driven engineering*, *models* do not constitute documentation but are considered equal to code, as their implementation is automated, for example: a car order that includes customer features is straightforwardly reflected into reality, in the context of a current advanced automotive production line. Hence, the **domain** is essential for *models*. *Model-driven engineering* thus aims at finding *domain-specific abstractions* and making them accessible through formal *modeling*, this leading to automation of software production, which in turn leads to increased productivity (since both the quality and maintainability of software *systems* increase) – *models* that are *domain-specific* and *computation-independent* can be understood by *domain* experts and at the same time, those *models* are restricting accordingly the *technology-specific models* that are essential for the *construction* of the software *system* under development. To successfully apply this, three requirements must be met: (i) *Domain-specific* languages are required to allow the actual formulation of *models*. (ii) Languages that can express the necessary *model-to-code transformations* are needed. (iii) Compilers, generators, or transformers are required that can run the transformations to generate code executable on available platforms [66]. Said otherwise:

- It is necessary to consider computation-independent *models* that capture adequately the *domain* features, abstracting from any computation and technical details; such *models* would ideally capture the *as-is* situation, describing the context in which the software *system*-to-be will be integrated.
- It is necessary to consider technology-independent *models* of the software *system*-to-be, which *models* are already focused on the *system*-to-be (maybe both *functionally* and *constructionally*) but just conceptually, not imposing any technical restrictions whatsoever.
- It is necessary to consider technology-specific *models* that capture adequately all technical features of the software *system*-to-be, which *models* are straightforwardly reflect-able to corresponding code.

As studied by Shishkov [54], two *modeling* facilities are meeting those requirements, namely the **Model-Driven Architecture** (**MDA**) and the *Open Distributed Processing Architecture* (*ODP*), with *MDA*'s adopting influences from *ODP*. Further, **meta-modeling** is one of the most important aspects of *model-driven engineering* since so-called '*meta-models*' are needed for describing the *abstract* syntax of *domain-specific modeling* languages, and that in turn allows *models* to be validated against the constraints defined in the *meta-model*, and that allows also for mappings between two *meta-models*; this is all necessary with regard to the desired automated code generation. Hence, *meta-models* are *models* that make statements about *modeling*. Four *meta-levels*

being defined and considered widely, are reflected in **MOF** – the **Meta-Object Facility** [66]. For this reason, we will consider *MDA* and *MOF* in Sub-section 4.3.1 and Sub-section 4.3.2, respectively.

4.3.1 Model-Driven Architecture

Model-Driven Architecture (*MDA*) is a software architecture framework consisting of a set of standards that assist in *system* creation, *system* implementation, *system* evolution, and *system* deployment [66]. The key *MDA* technologies are *UML*, *MOF* (to be considered in the following sub-section), and the *XML Meta-data Interchange –XMI* [84,83]. *MDA* emphasizes the importance of *modeling* for the software architecture design, suggesting a three-layered approach:

- **Computation-Independent Model** – **CIM**, describing a system from the computation-independent point of view, to address structural aspects of the system;
- **Platform-Independent Model** – **PIM**, defining a system in terms of a technology-neutral virtual machine or a computational abstraction;
- **Platform-Specific Model** – **PSM**, capturing the technical platform concepts and geared towards implementation.

A taxonomy of the models that play a central role in MDA is presented in Figure 4.6:

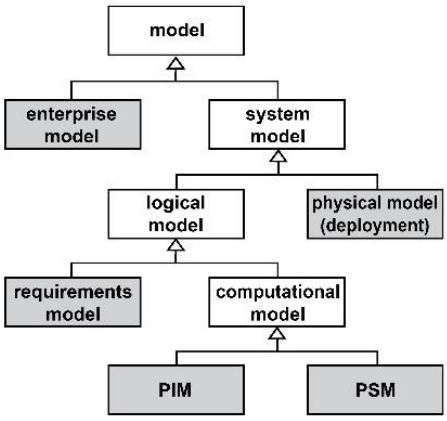


Fig. 4.6. Classification of models in the MDA context.

Since resolving the mismatch between (user) requirements and software application functionality is an essential software development concern [54], *MDA* needs to address it and this regard, one would inevitably face the necessity of bridging different *abstraction levels* – a high-level *business* logic and a technology-driven application

functionality. A *business function* (corresponding to a unit of *business logic*) is specific for a particular *business* and necessarily abstracts from technological solutions that can be used to support it. A *technology platform* offers a generic engineering abstraction (hence *hiding implementation details*) that is nonetheless technology-oriented. According to [64], an adequate business – application alignment can only be achieved if the **initial *enterprise model*** is: (i) a valid reflection of the **relevant real-life aspects** and (ii) a suitable foundation for the **generation of application *models***, preferably by using automated transformations. The alignment nevertheless cannot be accomplished only by prescribing how to define an *enterprise model* – an additional demand should be that: (iii) the '**architectural style**' used for organizing the application *modeling* should facilitate the alignment; it cannot be obtained solely from *top-down*, but also requires a *bottom-up* 'preparation'.

Hence, we consider *enterprise modeling* to be *computation-independent*, with no focus on the (partial) automation of *business processes* – this corresponds to the *CIM* layer. Further, we consider we consider *application modeling* from a *platform-independent* perspective, with no focus on the specific technological platform(s) on which the application *components* are (to be) implemented – this corresponds to the *PIM* layer. Thus:

```
the enterprise-modeling-driven
generation of software specification
corresponds to a CIM-to-PIM transformation.
```

As for the *PSM*, it is specific with regard to *J2EE*, *.NET*, or other implementation platforms. A *platform-specific model* is created from a *platform-independent model* via a *model transformation*. Thus:

```
the application-modeling-driven
implementation of software
corresponds to a PIM-to-PSM transformation.
```

In the following sub-section, we will consider *meta-modeling* and *MOF*, as already mentioned.

4.3.2 Meta-Object Facility

The *Meta-Object Facility* (*MOF*) provides an open and *platform-independent* meta-data management framework and associated set of meta-data services to enable the development and interoperability of *model* and meta-data -driven *systems*. Examples of *systems* that use *MOF* include *modeling* and development tools, data warehouse systems, meta-data repositories, and so on [48]. The above-mentioned four meta-levels are of importance with regard to *MOF* [66] – they are: (i) **`M0`** – **`Instance`**; (ii) **`M1`** – **`Model`**; (iii) **`M2`** – **`Meta-model`**; (iv) **`M3`** – **`Meta-meta-model`**.

Between **`M0`** and **`M1`**, we have typical *classification*/*instantiation*, at **`M1`** we have the *class* level and at **`M0`** we have the *instance* level, for example: a *class* is given the

name ‘Person’ and has a number of *attributes*, in the example – ‘sir name’ and ‘first name’; an *instance* of that *class* is created at level **M0**, in the example – ‘Person’ is *instantiated* to the persons with ‘ID 12345’, and we give corresponding *values* to the *attributes* ‘sir name’ and ‘first name’ – ‘Smith’ and ‘Michael’, respectively, in this case. Logically, a *class* can have more than one *instance*. As seen in the above example, during the *instantiation* of a *class*, *values* are assigned to *attributes* of the *class*.

As for the **M2** level, at this level, the constructs that are used at the **M1** level are defined. The elements of the **M1** *model* are hence *instances* of the elements of the *meta-model* at the **M2** level; since in the above example we use *classes* in the **M1** model, the construct *Class* must be defined in the **M2** *meta-model*. The construct *Class* in turn is to be considered as an *instance* of the meta-meta element *MOF Classifier* (*MOF classes* are hence defined at the **M3** level. Said otherwise, the *MOF* serves to define *modeling* languages (such as *UML*, for example) at the **M2** level.

Further, besides meta-relationships in which *meta-models* define the concepts needed for creating corresponding *models*, it has to be acknowledged that *models* can be located on different **`abstraction levels`** even though they can be located on the same *meta-level*, for example: *CIM*, *PIM*, and *PSM* (see above). As already discussed, *transformations* are used to map *models* at a *higher abstraction level* to models at a *lower abstraction level*, and as mentioned before, each *model* is inevitably an *instance* of a *meta-model*.

If we take the *PIM*-to-*PSM transformation* (where we reflect the *higher abstraction level model*, *PIM* to lower level, *PSM*), we stay at the **M1** level because no matter the *abstraction level*, both *PIM* and *PSM* represent *models*. Each of those *models* thus has a corresponding *meta-model* (at the **M2** level): the *PIM* is an *instance* of the *PIM meta-model* and the *PSM* is an *instance* of the *PSM meta-model*. In turn, both *meta-models* are *instances* of *MOF*, *MOF* being positioned at the **M3** level. This is illustrated in Figure 4.7:

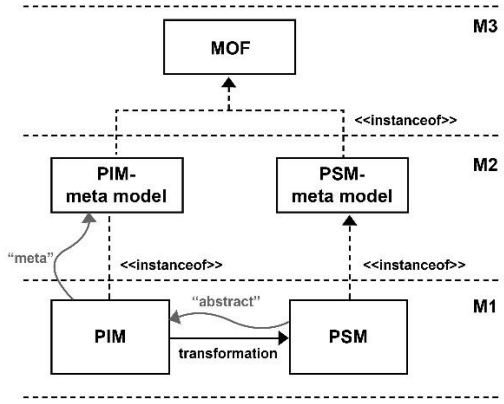


Fig. 4.7. Meta versus abstract.

In this section, we considered the *model-driven* software development, touching upon *abstraction levels*, *meta-levels*, and corresponding *transformations*. In the following section, we will consider the impact of *mobility* on the development and utilization of software *systems*, featured mainly by *cloud computing* and corresponding infrastructures.

4.4 Cloud Computing

Consolidated *enterprise*-IT solutions have proven to enhance business efficiency when significant fractions of local computing activities are migrating away from desktop PCs and departmental servers and are being integrated and packaged on the *Web* into the *computing cloud*, according to Ivanov [36]. Whether referred to as *grid computing*, *utility computing*, or **`cloud computing`**, the idea is basically the same: instead of investing in and maintaining expensive applications and *systems*, users access and utilize dynamic computing structures to meet their fluctuating demands on IT resources efficiently and pay a fixed subscription or an actual usage fee. The immense economic demands in the last several years, in conjunction with the immediate reduction of upfront capital and operational costs when cloud-based services are employed, increase the speed and the scale of *cloud computing* adoption both *horizontally* (across industries) and *vertically* (in organizations' technology stacks). All that poses the need for organizational changes – organizations would have to re-think and re-engineer (in some cases) their traditional IT resources, advancing them with cloud architectures and implementing services based on dynamic computing delivery models. The changes and business transformations are underway on a large scale, from providers and customers to vendors and developers. The key issues are not only in economics and management, but essentially how emerging IT models impact organizational structures, capabilities, business processes, and consequential opportunities.

There are usually three cloud service models under consideration, namely: **`Software as a Service`** (**`SaaS`**), **`Platform as a Service`** (**`PaaS`**), and **`Infrastructure as a Service (IaaS`**), that relate to the **`cloud provider`** [12]:

- *SaaS* moves the task of managing software and its deployment to third-party services, such as security services, caching services, networking services, and so on.
- *PaaS* functions at a lower level than *SaaS*, typically providing a platform on which software can be developed and deployed, such as streaming platforms, application development platforms, web platforms, and so on.
- *IaaS* in turn comprises highly automated and scalable compute resources, complemented by cloud storage and network capability which can be self-provisioned, metered, and available on-demand, such as e-mail building blocks, ERP building blocks (‘ERP’ standing for ‘Enterprise Resource Planning’), CRM building blocks (‘CRM’ standing for ‘Customer Relationship Management’), and so on.

The *cloud* provisioning is hence bottom-lined by a *SaaS-PaaS-IaaS* basis, and reaching out to customers via the Internet, such that the customer’s computers, servers,

databases, mobile devises and so on can actually benefit from corresponding *cloud services* that are in turn utilized by customers in the form of images, news, music, chat facilitations, ID management, TV, and so on, as illustrated in Figure 4.8:

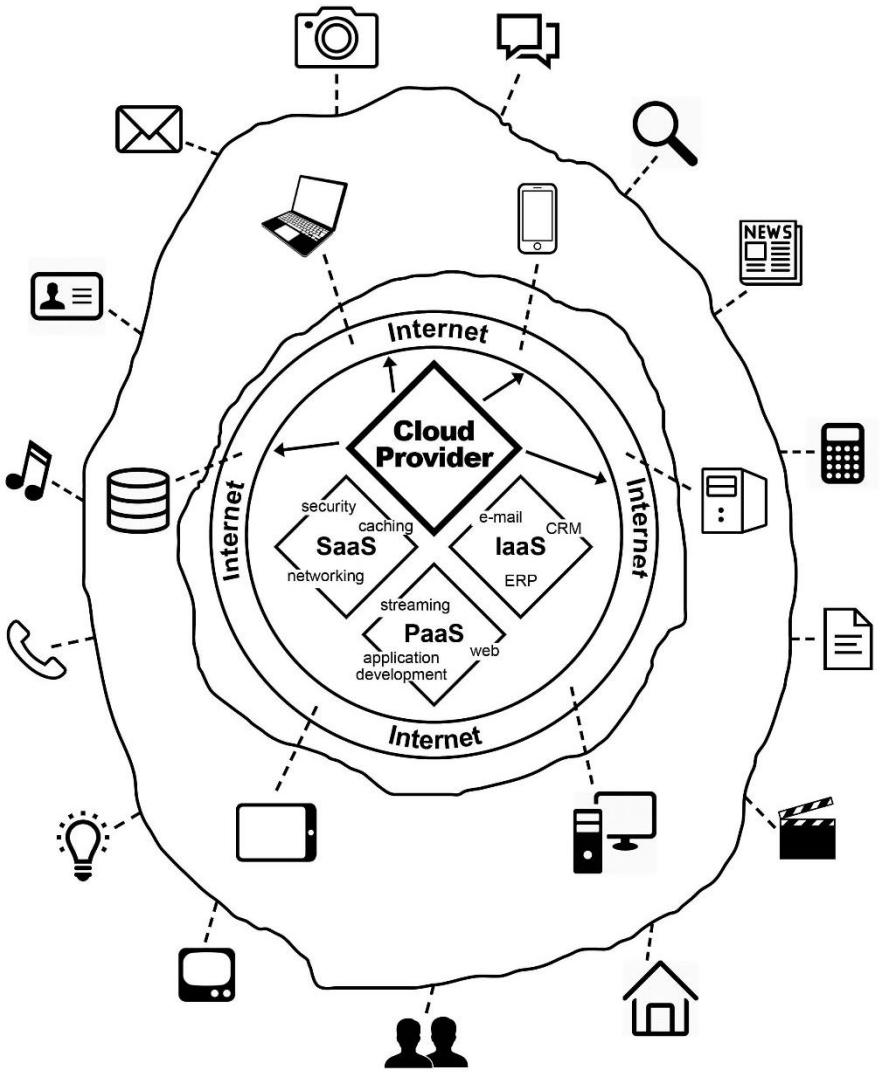


Fig. 4.8. Vision of cloud computing.

As the figure suggests, customers utilize *cloud services* at high level, in an intuitive and seamless way, such that the underlying *SaaS-PaaS-IaaS*-related technical complexity remains hidden and would only become explicit for the customer as reflections in corresponding (subscription) contracts. Thus, *cloud computing* brings together many technical, organizational, contractual, and other concerns which we will not discuss in more detail in the current chapter. Our goal was to present *cloud computing* as a natural 'extension' of *service orientation* (already discussed) where the utilization of *services* is combined with the utilization of resources, empowering **`mobility`** – it is only through *cloud computing* that it is possible to access distant resources / systems through a (portable) mobile device.

All this reflects the move from *components* through *services* to *cloud solutions*, and we acknowledge the relevance of *model-driven engineering* (discussed already) to the challenge of developing such *components-services-cloud*-based *systems*. What remains uncovered nevertheless is the adequate consideration of non-functional issues, such as *privacy*, for example, which are crosscutting and have reflection in different components, at different development phases, and so on. We will discuss this in the following section.

4.5 Aspect-Oriented Software Development

Privacy, transparency, traceability, and so on are labelled as **`values`** that are to be *weaved* in the functioning of *enterprise systems* and *EIS* [2] and for this reason, they are considered as **`crosscutting concerns`** because:

- Weaving them in the functioning of a *system* would not assume reflections in one particular *component* only, instead: multiple *components* would need to be 're-factored' as well as their interrelations, as well as their relations to other *components*.
- Addressing such *values* in the software development context would come through all the phases of the software development life-cycle.

Further, such *values* / *crosscutting concerns* have a *non-functional* essence because they do not have any particular purpose or function, instead: they represent something like 'desired system qualities'.

Finally, even though the *values* / *crosscutting concerns* are *non-functional*, we should find *functional* solutions for them, because we argue that a *system* could only *functionally* achieve effects with impact on its *environment*.

This all (as above stated) concerns broadly *enterprise systems* touching upon both human issues and technical issues. Narrowing this further to software *systems* nevertheless brings us to such *crosscutting concerns* that are particularly touching upon software development issues, such as security, distribution, recoverability, logging, performance monitoring, and so on [8]. This is featuring the notion of **`aspect-oriented software development`** whose foundations are separating concerns, filter technologies, improving modularity, integrating new features, and so on. [27]. We are not going in more detail in this direction.

What we only like to emphasize is that addressing such *non-functional* concerns is to be *functional* which means that:

- We should 'translate' those concerns into system requirements.
- System development should not go in any unusual way, it should just ensure that all requirements are properly reflected in the design and implementation.
- Introducing metrics and/or performance indicators would be necessary for establishing how well the desired *values* have been reflected in the performance of the *system* and if it is necessary, the requirements may have to be re-factored.

Aspect-orientation is thus necessary for properly *weaving* desired *values* in the functioning of the *system*-to-be. If is featuring *non-functional* issues that nevertheless have to be resolved *functionally*.

IN SUMMARY, in Chapter 2 we have considered some essential the *concepts* and *views*; in Chapter 3 we have presented and discussed *social theories*, including *human relativism*, the *theory of organized activity*, the *language/action perspective*, *enterprise ontology*, and *organizational semiotics*, justifying their relevance to different aspects concerning *enterprise systems* and *EIS*; in Chapter 4 we have considered *computing paradigms* that are currently actual and also well-combinable with the *social theories* and *concepts* considered. In the following chapter, we will bring those issues together, featuring the *SDBC approach*.

Chapter 5

THE SDBC APPROACH

'*SDBC*' stands for '*Software Derived from Business Components*'. **SDBC** is a software specification approach that covers the early phases of the software development life cycle and is particularly focused on the derivation of *software specification models* on the basis of corresponding (re-usable) *enterprise models*. *SDBC* is based on three key ideas: (i) The software *system*-to-be is considered in its *enterprise* context which not only means that (as mentioned above) the *software specification models* are to stem from corresponding *enterprise models* but means also that a deep understanding is needed on real-life (*enterprise*-level) processes, corresponding roles, behavior patterns, and so on. (ii) Bringing together two disciplines, namely *enterprise engineering* and *software engineering*, *SDBC* pushes for applying *social theories* in addressing *enterprise-engineering*-related tasks and for applying *computing paradigms* in addressing software-engineering-related tasks, and also for bridging the two, by means of sound methodological guidelines. (iii) Acknowledging the essential value of re-use in current software development, *SDBC* pushes for the identification of re-usable (generic) *enterprise engineering building blocks* whose *models* could be reflected accordingly in corresponding *software specification models*.

The initial ideas behind *SDBC* have been proposed by Shishkov in 2005 [54] and since then the approach has been maturing slowly. Since no sound and widely recognized methodology has appeared to take the above focus and that lack is widely claimed to continue to cause numerous failures of software projects, we are inspired to work further on the *SDBC* project. Nevertheless, this has never been and is not a matter of any kind of commercialization whatsoever neither it is related to branding or product positioning. *SDBC* remains fundamentally driven by a scientific and research inspiration, and for this reason, it is not aligned with particular commercialized development tools. Hence, *SDBC* is positioned as an open modeling platform that may accommodate different tools, as far as the overall principles of the approach are met, and what stays essential about *SDBC* is the challenge of bringing together *social theories* (in an *enterprise engineering* context) and *computing paradigms* (in a *software engineering* context), aiming at the *enterprise-modeling*-driven specification of software.

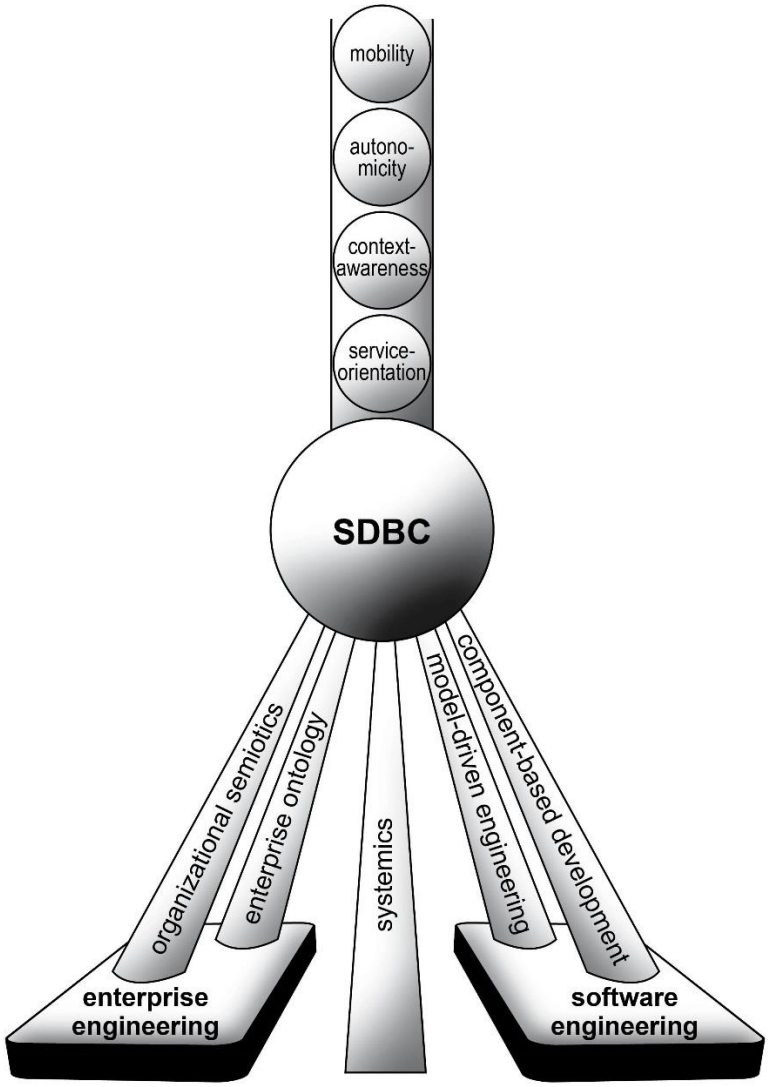


Fig. 5.1. The SDBC foundations.

As it concerns the modeling itself, *SDBC* assumes four modeling perspectives, namely: *Structural perspective* that reflects entities and their relationships; *Dynamic perspective* that reflects the overall business process and corresponding to this – the states of each entity, evolving accordingly; *Data perspective* that reflects the information flows across entities and within the business process; *Language-action perspective* that reflects real-life human communication and expression of promises, commitments, etc. as also relevant to soundly building an exhaustive enterprise model.

In this, *SDBC* is grounded, as Figure 5.1 shows, in the principles of *systemics* (see Chapter 2) and also in:

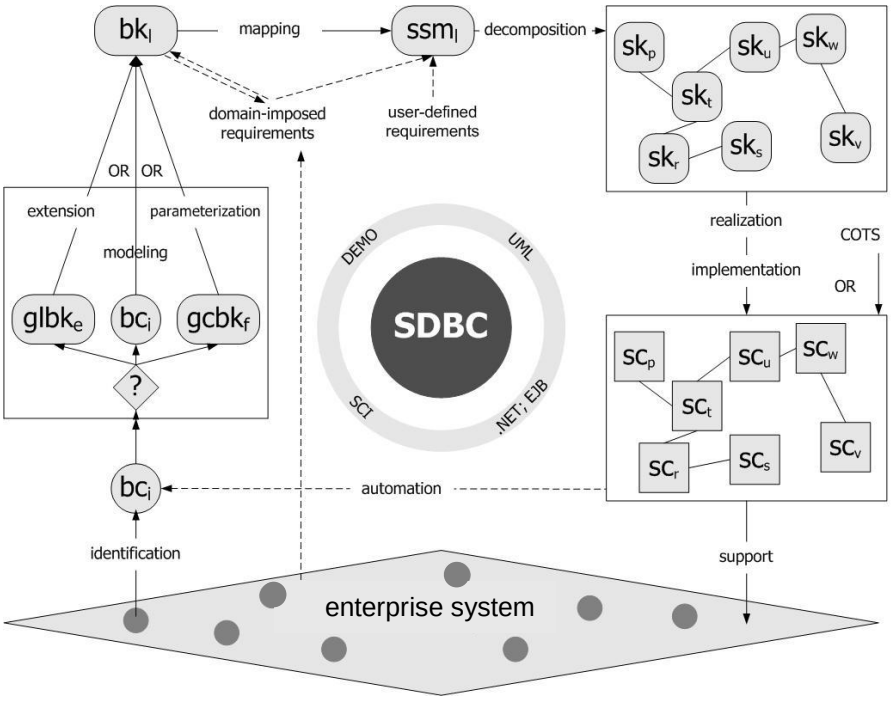
- *Enterprise engineering* and in particular, *enterprise ontology* and *organizational semiotics* (see Chapter 3);
- *Software engineering* and in particular, *model-driven engineering* and *component-based development* (see Chapter 4).

As also suggested by the figure, *software specification models* derived by applying *SDBC*, can be further updated to accommodate features pointing to: (i) *service-orientation* (and *mobility* utilization related to this), as studied in [63]; (ii) *context-awareness*, as studied in [53]; (iii) *autonomic system behavior*, as studied in [65].

Further, with regard to concepts, among the main *SDBC* concepts are the following:

- *Component* vs *CoMponent*: while *components* represent part of the whole, *coMponents* reflect a *model* of a *component* adequately elaborated in all four perspectives (see above), and we could thus have *business components* (business sub-systems) and *software components* (pieces of implemented software) as well as *business coMponents* and *software coMponents*, respectively; Refer to *Definition 8*, *Definition 11*, *Definition 13*, *Definition 14*, and *Definition 15*.
- *General* vs *Generic*: those concepts are both about *re-use*, still – *general* is about re-using an abstract core (a general reservation engine, for example) while *generic* is about parameterizing something that is multi-specific (a car system to be adjusted to automatic or gear regime, for example).
- *Software Specification Model* – this is a *technology-independent functionality model* of the software *system*-to-be.

To summarize the *SDBC* outline, we use Figure 5.2. As seen from the figure, we consider an *enterprise system* from which a *business component*(*s*) is to be identified and then reflected in a relevant *model* – a *business coMponent*. Another way for arriving at a *business coMponent* is by applying *re-use*: either extending a *general business coMponent* or parameterizing a *generic* one. Then, the *business coMponent* should be elaborated with the **domain-imposed requirements**, in order to add elicitation on the particular context in which its corresponding *business component* exists within the *enterprise system*. Then, a **mapping** towards a *software specification model* should take place and the **user-defined requirements** are to be considered, since the derived software *model* should reflect not only the original business features but also the particular *requirements* towards the software *system*-to-be. The *software specification model* in turn needs a precise elaboration so that it provides sufficient elicitation in terms of *structure*, *dynamics*, *data* and *language-action –related aspects*. It needs also to be decomposed into a number of *software coMponents* reflecting functionality pieces. Those *coMponents* then are to undergo *realization* and *implementation*, being reflected in this way in a set of *software components*. Some *software components* could also be purchased. The *software components* are implemented using *software component* technologies, such as *.NET* or *EJB*, for instance. Finally, the (resulting) *component-based ICT application* would support informationally the target *enterprise system*, by automating anything that concerns the considered *business component* (identified from the mentioned *system*).



Abbreviations:

bc – Business Component	ssm – Software specification model
bk – Business CoMponent	sc – Software Component
glbk – General Business CoMponent	sk – Software CoMponent
gcbk – Generic Business CoMponent	

Fig. 5.2. SDBC – outline [54].

In order to bring forward further elaboration with regard to the *SDBC* approach, it is necessary to consider the *SDBC* design trajectory: As suggested by Figure 5.3-a [54], one should firstly consider the *initial descriptive information* (provided by the future user(s) of the software *system*-to-be) which is a usual input in any software project, as it is well-known. Then a *description of the approached business reality* is derived. However, it might be necessary to conduct *re-design* (imagine that the original business reality consists of a local service provider and users; introducing mobility, we could rely on a number of service providers based in different locations; thus, before specifying software, we would need to describe the 'future' (desired) business reality accordingly). Then, we should *delimit* a relevant part of the business reality depending on our particular software goal (on what we are going to automate, according to the requirements of the users). Figure 5.3-b [54] summarizes those issues.

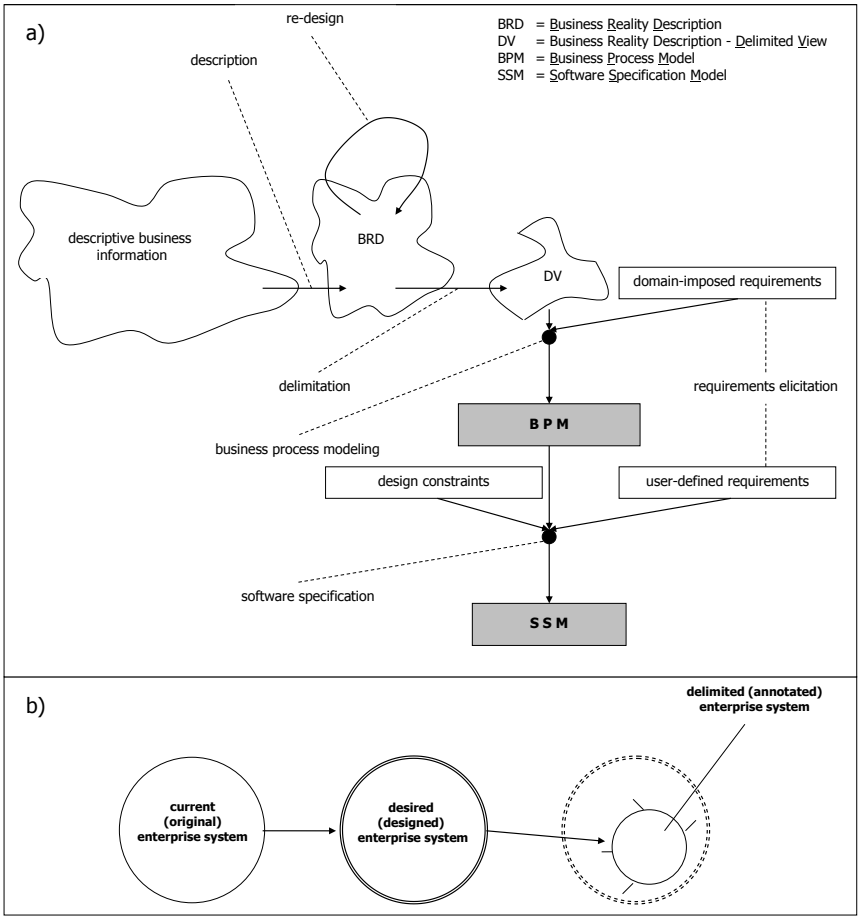


Fig. 5.3. SDBC: design trajectory.

Hence, having the description of the delimited part of the original (or eventually re-designed) business reality, we could proceed towards the business process *modeling* task (Figure 5.3-a). As seen from the figure, another related input is to be the *domain-imposed requirements* characterizing the original *enterprise system*.

We build a *business process model* that in turn is to be mapped towards a *software specification model*. However, as it is depicted on the figure, besides the *business process modeling input*, the *SDBC design trajectory* envisions two other necessary inputs:

- the *user-defined requirements* – the requirements which the future user(s) of the software *system*-to-be have stated concerning its functionality;
- *design constraints* – the design limitations which should be followed as a result of software/hardware/netware (and other) project restrictions.

Thus, five basic tasks could be identified, namely description (plus eventually re-design), delimitation, business process modeling, software specification as well as requirements elicitation.

The figure shows as well that the *requirements elicitation* task would span not only over the *software specification* but also over the *business process modeling*.

Concerning the items depicted on Figure 5.3-a: from left to right and from top to bottom they become smaller (in area) and more regular (in shape). This is to indicate that each following state relates to a smaller part of the original business reality (in the delimitation, we exclude issues from the original *model*, in the *business process modeling*, we further exclude issues from the delimited *model*, and so on) and is becoming more and more structured.

We will now bring forward further insight on four of the above-mentioned tasks, since they require elaboration - those are: (i) delimitation; (ii) business process modeling; (iii) software specification; (iv) requirements elicitation.

(i) Delimitation

As seen from Figure 5.3-a, before the *software specification* and even before the *business process modeling* activities take place, it is necessary to conduct a sound *business process* study that thoroughly reflects the considered business reality, achieving in this way a precise *delimitation*. We consider this necessary because, as it is well-known, an adequate modeling should be conducted based on a proper description and understanding of the addressed reality as well as on a precise focus on the part of the reality to be considered in the modeling process [57]. In *SDBC*, we respond to this through 'description+filtration':

- It is necessary to thoroughly describe the *enterprise system* being approached (the business reality under consideration, which might be (eventually) re-designed) and the suggested starting point in this regard is the consideration of the original documentation of the studied *system*; however, it should be taken into account that such information is usually insufficient and often full of errors. Thus, it should be additionally analyzed and/or refined. The decision how detailed the description should be depends on the selected granularity level that in turn should be adequate to the characteristics of the software system-to-be.
- Then, with regard to only those issues from the description, which are relevant to the software *system*-to-be, filtration needs to be applied. They are to be, however, soundly rooted in the broader context of the approached business reality. This link would contribute to building software that is well integrated in the target *enterprise*.

In order to illustrate the above, we consider an example featuring a restaurant: to make a DESCRIPTION with regard to a restaurant means to cover a number of issues, such as location, opening hours, food details, price details, reservation procedure, service peculiarities, reputation, and so on. There would be much information collected along those lines which information would nevertheless remain unfocused. If we would be introducing some technology within the restaurant, for example – an electronic reservation system, then we would have to apply FILTRATION with regard to the description, such that we extract only those description elements that are relevant to the reservation functionality.

However, *description* and *filtration* are not to be always realized as two separate tasks, it is possible that they overlap. Returning back to the example: it might be obvious from the beginning that describing the porter (of the restaurant) is of no use since the 'functionality' of the porter is irrelevant to the restaurant (electronic) reservations; irregardless of other circumstances, the Porter must stay by the restaurant's entrance during the opening hours.

It might be concluded that *filtration* concerns the alignment between *business process modeling* and *software specification* since it focuses the business study on particular part(s) of the studied business reality, which are to be automated through (software) technology [57].

(ii) Business Process Modeling

Inspired by *Definition 8*, *Definition 10*, and *Definition 11*, we establish the need to conduct *business process modeling* with providing elaboration in three perspectives, namely: (i) *Structural perspective*; (ii) *Dynamic perspective*; (iii) *Data perspective*. Further, inspired by the notion of *transaction* (see *Definition 5* and Figure 3.4) and LAP (see Chapter 3), we add another perspective, namely the *communication perspective*. All this is illustrated in Figure 5.4:

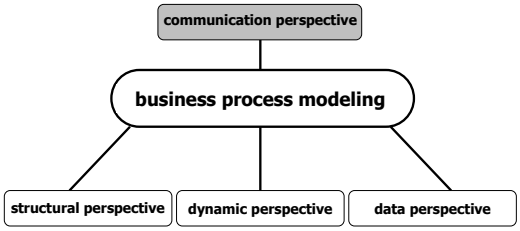


Fig. 5.4. SDBC – business process modeling perspectives.

As for the perspectives: the *structural perspective* is about the entities and their interrelations; the *dynamic perspective* is about the flow(s) of events; the *data perspective* is about the factual issues; the *communication perspective* is about the communicative acts exchanged during the business operation.

(iii) Software Specification

Since *SDBC* is to deliver a *software specification model* that is derived based on a corresponding *enterprise model* that features in turn (among other things) *business processes* to which four perspectives are applied, as discussed above, we need to reflect a multi-perspective *business process model* in corresponding *software specification* reflections. Further, if possible, such an alignment *between business process modeling* and *software specification* is to be *component-based*. Said otherwise, the *software specification model* is to be derived based on (re-usable) *business coMponents*.

(iv) Requirements Elicitation

Requirements relate directly to the *specification of software* [79]. They are descriptions of how the *system*-to-be should behave, application domain information, constraints on the *system*'s operation, or specifications of a *system* property or attribute [37]. Thus, a

proper consideration of the original business requirements in the specification of a software's functionality is of significant importance in the process of aligning *business process modeling* and *software specification*. Our consideration of the *requirements* issue as illustrated in Figure 5.2 is in consistency with the *SDBC design trajectory* (Figure 5.3).

Building a *business process model* should concern the discovery of a part of the system *requirements*, namely those *requirements* that characterize particularly the *enterprise system* under consideration, as discussed already. They are often called *domain-imposed requirements*, as already mentioned. It is to be mentioned in this regard that not only the *domain-imposed requirements* could affect the initial *business process model*, by causing some updates in it but also that the *business process model* affects the *requirements* elicitation, by stimulating the discovery (or specification) of additional *requirements*.

As mentioned already, besides the *domain-imposed requirements* one should identify also the so-called *user-defined requirements* that are determined by the users of the *system*-to-be and are not directly related to the *business process model*.

In summary: during the *business process modeling*, the *domain-imposed requirements* are to be discovered and considered in the mapping towards *software specification*; next to that, the *user-defined requirements* are to complement the *business process model* in providing the input for the derivation of the *software specification model*.

Further, *transactions* (see *Definition 5* and Figure 3.4) are considered as the fundamental *enterprise modeling* building block in the *SDBC* context. Still, there is a particular *SDBC* interpretation of the *transaction* concept.

***SDBC* interprets the *transaction* concept** as centered around a particular *production fact* (see *Definition 5*). The reason is that the actual output of any *enterprise system* represents a set of *production facts* related to each other. They actually bring about the useful value of the business operations to the outside world and the issues connected with their creation are to be properly modeled in terms of *structure*, *dynamics*, and *data*.

However, the already justified necessity of considering also the corresponding *communicative* aspects is important. Although they are indirectly related to the *production facts*, they are to be positioned around them. As already stated, *SDBC* realizes this through its interpretation of the *transaction* concept, as depicted in Figure 5.5; as seen from the figure, the *transaction* concept (as featured *Definition 5* and Figure 3.4) has been adopted, with a particular stress on the *transaction's* output – the *production fact*. The *order phase* is looked upon as an input for the *production act*, while the *result phase* is considered to be the *production act*'s output. The dashed line shows that a *transaction* could be successful (which means that a *production fact* has been (successfully) created) only if the *initiator* (the one who is initiating the *transaction*, as presented in Figure 5.5) has accepted the *production act* of the other party (called *executor*). As for the (*coordination*) *communicative acts*, grasped by the *SDBC transaction*, they are also depicted in the figure. The *initiator* expresses a *request* attitude towards a proposition (any *transaction* should concern a *proposition* – for example, a shoe to be repaired by a particular date and at a particular price, and so on). Such a *request* might trigger either *promise* or *decline* - the *executor* might either

promise to produce the requested product (or service) or express a *decline* attitude towards the *proposition*. This expressed attitude actually triggers a discussion (negotiation), for example: 'I cannot repair the shoe today, is tomorrow fine?... and so on'. The discussion might lead to a compromise (this means that the *executor* is going to express a *promise* attitude towards an updated version of the *proposition*) or might lead to the *transaction's* cancellation (this means that no *production fact* will be created). If the *executor* has expressed a *promise* attitude regarding a *proposition*, then (s)he must bring about the realization of the *production act*. Then the *result phase* follows, which starts with a *statement* expression from the *executor* about the requested *proposition* that in his/her opinion has been successfully realized. The *initiator* could either accept this (expressing an *accept* attitude) or reject it (expressing a *decline* attitude). Expressing a *decline* attitude leads to a discussion which might lead to a *compromise* (this means that finally the *initiator* is going to express an *accept* towards the realized *production act*, resulting from *negotiations* that have taken place and *compromise* reached) or might lead to the *transaction's cancellation* (this means that no *production fact* will be created). Once the realized *production act* is *accepted* the corresponding *production fact* is considered to have appeared in the (business) reality.

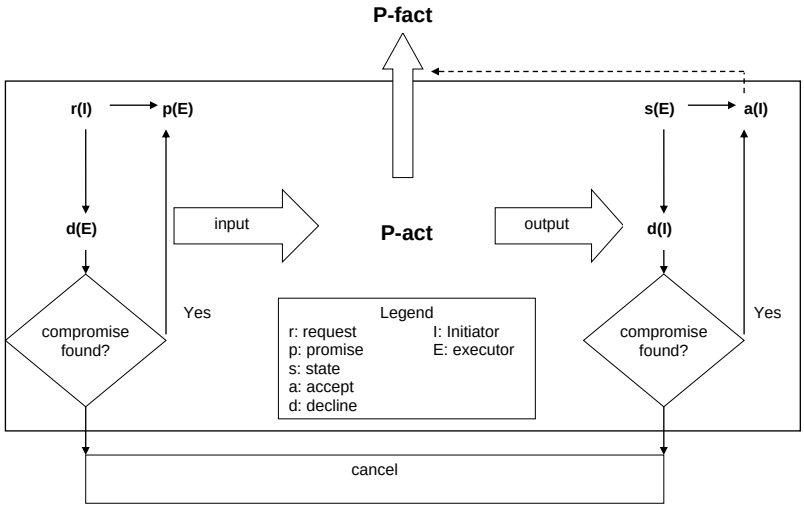


Fig. 5.5. The SDBC interpretation of the transaction concept.

Further, the **component based enterprise-software alignment** is considered crucial with regard to *SDBC* and justified, inspired by the indisputable advantages of component-based development (see Chapter 4) and related to this – the power of *re-use*. The *component-based alignment* between *business process modeling* and *software specification* is illustrated in Figure 5.6:

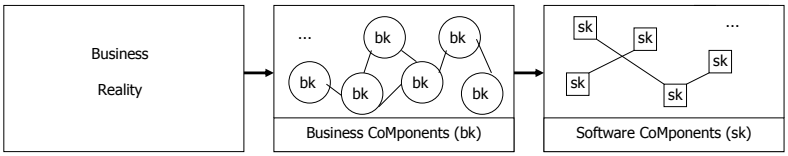


Fig. 5.6. From business coMponents to software specification.

As depicted in the figure, the target business reality is to be reflected in a set of identified *business coMponents* (see *Definition 11*). Based on them, a *component-based software model* is to be specified, in terms of *software coMponents* (see *Definition 15*). The *business coMponents* and *software coMponents* are not to be necessarily mapped one-to-one (the former is a purely *enterprise engineering* concern while the latter should have the perspective of the *software system-to-be*).

Still, that kind of alignment allows for: (i) ease of modifications (both at *enterprise* level and *software* level) that are 'localized' in a particular *business* / *software coMponents*; (ii) traceability – one could easily 'trace' between *enterprise* level and *software* level, being capable of analyzing, for example, what would be the *software* impact of a newly introduced *enterprise*–level feature (and vice versa); (iii) *business coMponents* and/or *software coMponents* could be conveniently re-used.

As for **re-use**, *three re-use levels* are essential for *SDBC*, namely:

- Re-use of *software coMponents* (lowest level)
- Re-use of *business coMponents*;
- Re-use of *business processes* (highest level).

Re-using *software coMponents* is an option within the *SDBC* approach, acknowledging the power of re-using *software components* as according to *component-based development* (see Chapter 4). Still, dealing with *re-use* at the *software component* level goes beyond the direct scope of *SDBC* that focuses at the derivation of SOFTWARE SPECIFICATION. Hence, dealing with *software coMponents* (see *Definition 15*) is well within that focus. At the same time, methodologically *re-using software coMponents* is a good basis for straightforwardly reflecting this in corresponding *software components*. As for the *re-use* itself (of *software coMponents*), we will discuss it only after explaining how *software coMponents* are to be identified within *SDBC*. This is illustrated in Figure 5.7.

As it is seen from the figure, a *business coMponent* is to be methodologically reflected in the *specification of software*. As also seen, such a 'business process input' alone is insufficient for specifying a piece of *software*. One is to consider as well what do the (future) users of the system-to-be require, as discussed already. Said otherwise, this is about considering the *user-defined requirements*.

One is to consider as well some technical (and technological) issues leading to design restrictions (since *software systems* are about the technological solutions of some 'problems' in *enterprise systems*).

Based on all that input, a *business coMponent* could find its reflection in a *software specification model* of the system-to-be. The model could be presented, for instance, in the *use case* notations. However, for the purpose of *re-use*, we might find it useful to

identify (by decomposing the *model*) some *software coMponents*. Hence, we arrive at the identification of a *software coMponent(s)*. As shown in the figure, there is also another possibility, especially when we do not have the usual situations of a number of *software coMponents* corresponding to one *business coMponent*: the situation might be (because of the granularity of a *business coMponent*, for example) that a *business coMponent* is reflected in a *software specification model* which is not wise to undergo decomposition (because it is *re-usable* as it is, for example). In such cases we directly arrive at the identification of a *software coMponent*, on the basis of the *business coMponent*. Figure 5.7 (its right part) illustrates particularly how in the first situation (Situation 'a') we reflect a *business coMponent* in a number of *software coMponents*, and in the second situation (Situation 'b') we reflect a *business coMponent* in just one *software coMponent*.

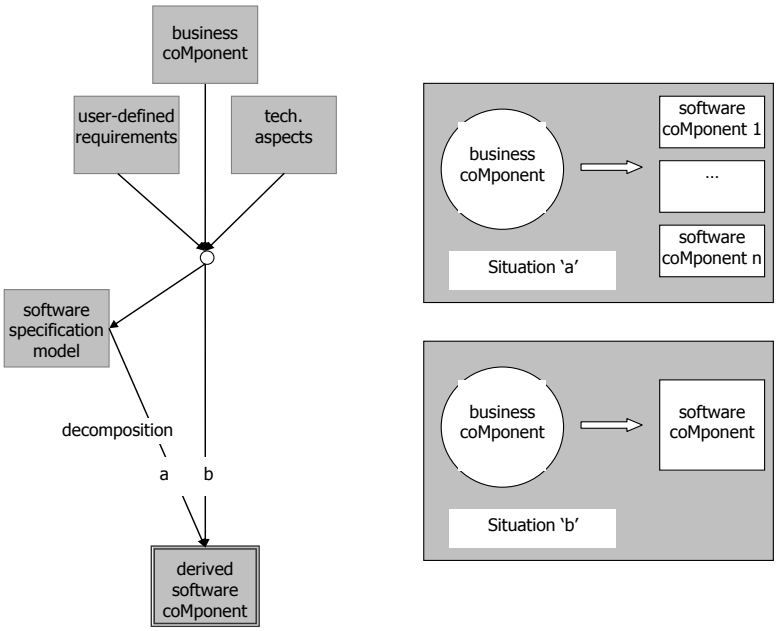


Fig. 5.7. Deriving a software coMponent.

Hence, re-use at the level of *software coMponents* is about re-using *modeling patterns* representing *software specifications*.

Re-using *business coMponents* points to the enterprise modeling level where we identify BUSINESS ENGINEERING BUILDING BLOCKS. As it concerns re-use, we are hence interested in re-usable (business engineering) building blocks that in turn can be either GENERAL building blocks or GENERIC building blocks – see Figure 5.8-a:

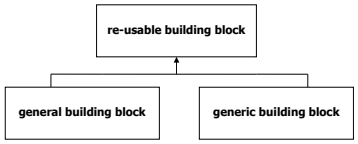


Fig. 5.8-a. Re-usable building blocks.

To illustrate this:

- An analogy for general is a lorry platform – it can be 'extended' in one way if the lorry would be transporting flowers and in another way – if the lorry would be transporting cars, for example.
- An analogy for generic is a universal plug adaptor – it can be 'adjusted' in one way if used in Japan and in another way – if used in UK, for example.

Hence, with regard to the *re-usability* of *business coMponents*, if *general* or *generic business coMponents* are identified, they could be *re-used* in the *specification* of different software artefacts; this could be realized either by *extending* a *general business coMponent* or by *parameterizing* a *generic business coMponent*, as illustrated in Figure 5.8-b:

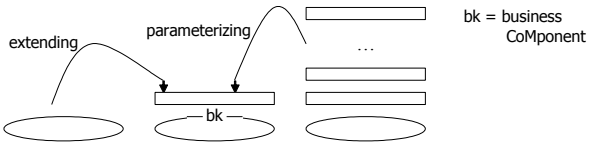


Fig. 5.8-b. Extending a general business coMponent or parameterizing a generic one.

General business coMponents are *models* that reflect core issues and can be *extended* in a number of directions. For example, a general brokerage model could be further developed – in one way for building an e-trade system and in another, for building a hotel reservation system, for example. Hence, the particular extension of a *general business coMponent* is motivated by the purpose of use. On the contrary, a *generic business coMponent* should contain in itself more than one optional functionalities. Through *parameterization*, such a *coMponent* can be adjusted depending on the desired purpose of use.

In summary - within *SDBC*, it is possible to derive a *business coMponent* in *three ways*: either in the trivial way (by building a *model* corresponding to a *business process*), or by *extending a general business coMponent*, or by *parameterizing* (adjusting) *a generic business coMponent* (Figure 5.9):

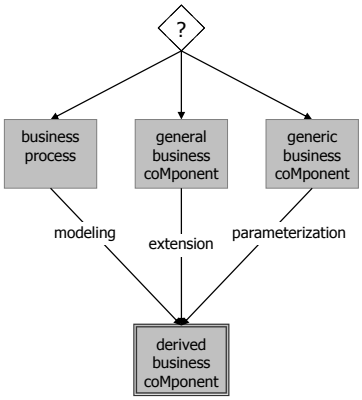


Fig. 5.9. Deriving a business coMponent.

Re-using a *business process* within *SDBC* is a matter of making a general *business process description* that is sufficiently abstract, such that *re-use* is possible. For example, an <arrangement of a service> IN GENERAL may be specified as coming through <registration> + <payment> + <reduction approval> + …, for example. Then, this abstract description can be extended in different ways:

- One example could be a HOTEL RESERVATION ARRANGEMENT that in particular comes through: NO REGISTRATION + PAYMENT OF A DEPOSIT & PAYMENT OF ADMINISTRATIVE COSTS + EARLY BOOKING REDUCTION APPROVAL + …;
- Another example could be an AUTO INSURANCE ARRANGEMENT that in particular comes through: REGISTRATION IN AN INSURANCE COMPANY + INSURANCE PAYMENT & PAYMENT OF ADMINISTRATIVE COSTS + NO-CLAIM REDUCTION APPROVAL + …;
- and so on.

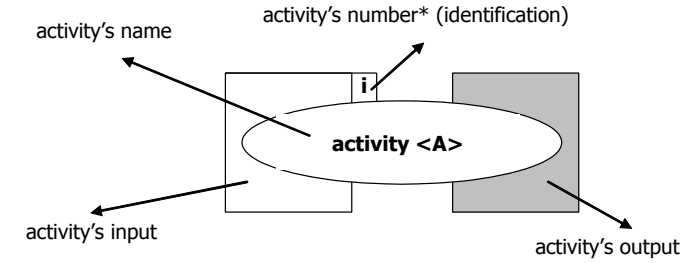
Hence, a *general business process* could be reflected in different *special business processes*, by adding some particular content to the general business description.

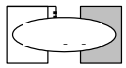
We have put forward the SDBC foundations and in the remaining of the current chapter, we will firstly present the SDBC outline (in Section 5.1) and then – the main SDBC notations (in Section 5.2).

5.1 SDBC Outline

Based on the essential *SDBC* fundaments presented already in the current chapter, this section briefly outlines the approach. Two graphical techniques have been developed for that purpose: the ACTIVITY MODEL and the INPUT/OUTPUT MODEL. The development of such techniques was considered necessary because neither of the popular ones (*activity diagram*, *flow charts*, *petri net*, *IDEFo* and so on [54]) proved to be sufficiently effective for thoroughly representing the *SDBC* steps, by providing information on both the dynamics of the activities to be realized and the inputs and

outputs of each of them. It is particularly useful not only that the *activity model* and the *input/output model* provide views respectively in those two essential directions but also that the two graphical techniques are completely consistent with each other. Hence, the dynamic aspect and the 'input-output' aspect are soundly matched between the two models [54]. The *activity model* itself (Figure 5.10) is sophisticated in terms of dynamics (it supports parallel processes, two types of synchronization, and so on) of the activities to be realized in applying *SDBC*; the *input/output model* in turn (Figure 5.11) represents the inputs and outputs of each activity. The legend regarding the graphical representation of those tools is as follows:





- activity
- decision point
- precedence
- connection between an output and its relevant input
- a point to which a (sufficient) number of iterations have to be made before proceeding further
- OR synchronization bar
- AND synchronization bar
- a synchronization bar's IN point
- a bank to store models in and/or use models from
- trigger to the SDBC modeling

* minor activities are not assigned numbers

Next to that: bp/bc stand for business process/coMponent

ATTENTION: representing *business coMponents* in different figures in the current book, we use either the label '**bk**' or '**bc**'. No matter if a *business coMponent* is labelled '**bk**' or '**bc**', we mean the same. The difference in labelling is only due to 'convenience' with regard to the particular figure, such that all used notations are easy to follow.

We will firstly consider the *SDBC activity model*, depicted in Figure 5.10. There are nine activities on the figure, and also four minor activities (they are not assigned a number; their names are backgrounded in grey).

There are three decision points and a point to which a sufficient number of iterations have to be made before proceeding further. There are two OR synchronization bars: the first one is associated with the IN points 'A' and 'B' (the AB bar); the second one is associated with the IN points 'E', 'F', and 'G' (the EFG bar). There is an AND synchronization bar; it is associated with the IN points 'C' and 'D' (the CD bar). There is a trigger to the application of *SDBC*, pointing to *Activity 1* ('information structuring'). The last activity from the model is *Activity 9* ('integration'). *Activity 1* and *Activity 9* are thus assigned 'start' and 'end' labels, respectively.

The trigger is pointing to Activity 1. It is about the *information structuring*, concerning a focused structured description of the target business reality; this includes thus a *delimitation* step (see above in the chapter). Then we arrive at the first decision point ('*conduct business process generalization?*'). There a decision is to be made on whether the mentioned structured business reality description should be used for the specification (modeling) of a *particular business process* (e.g. hotel reservation match-making), as reflected in Activity 2 ('*identification of a business process*'), or the description is to be used for achieving a *generalized view* (e.g. match-making), as reflected in Activity 3 ('*generalization of a business process*'). This decision should be based on certain criteria discovered in the process of studying the particular domain. For example, it might be known that an issue is unique for a company and thus, there is no sense to develop a generalized *model* of it. As seen from Figure 5.10, such a *business process generalization* (*Activity 3*) could be realized not only based on a structured description of the studied *enterprise system* but also based on the specification of a particular *business process* (this should be done if a *generalization* of such a specification will be also needed further by the modeler). That is why both before and after *Activity 2*, it is allowed for reaching the 'AB' synchronization bar which leads to *Activity 3*.

As also seen from Figure 5.10, a *model* of a particular *business process* (realized within *Activity 2*) might be used as well for building a *generic business coMponent* (Activity 5), as it is according to the second decision point ('*model a generic business coMponent?*'), in particular if the process flows towards the 'CD' synchronization bar. Otherwise, the process would flow towards the minor activity 'MODELING', from where we arrive at Activity 6 ('*constructing a business coMponent*'), through the 'EFG' synchronization bar. This reflects the situation in which no re-use is realized – we just specify a *business process* (*Definition 6*) and reflect it into a *business coMponent* (*Definition 11*).The *re-use* facilities of *SDBC* hence relate to Activities 3, 4, and 5.

As for *Activity 3*, after it there follows the third decision point ('*model a general business coMponent?*'). There a decision is to be made on whether a *general business coMponent* is going to be modeled; a *general model* of a business process is considered sufficient for building a *general business coMponent*. If *Yes*, Activity 4 ('*modeling a general business coMponent*') is reached, leading afterwards to the minor activity 'EXTENSION', from where we arrive at *Activity 6* ('*constructing a Business CoMponent*'), through the '*EFG' synchronization bar.* Otherwise the *'CD' synchronization bar* is reached. It leads to *Activity 5* ('*modeling of a generic Business coMponent*').

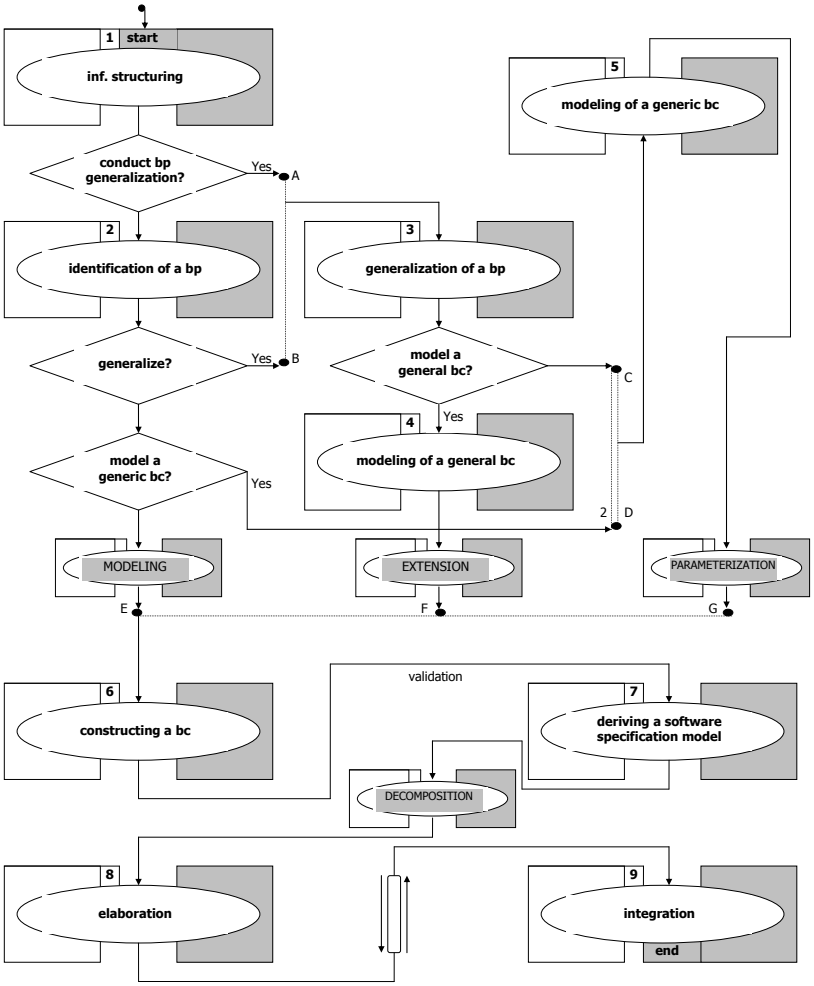


Fig. 5.10. SDBC – activity model.

As seen from the figure, for *modeling* such a *coMponent*, the required input is a specification of at least two (seen from the "**2**" at IN point 'D') models of particular *business processes* AND a *general business process specification (model)*. The reason is that the *generic model* would require not only a *general specification* which captures 'core issues' (derived from a *generalized business process model*) but also at least two particular *business process specifications* to be related to (at least two) corresponding selection options (options to be selected by *parameterizing* the *model*); actually, the rationale behind using *generic modeling patterns* (that capture, as discussed already, several possible design outputs based on grasped core issues) is that the modeler would be able to easily adjust the *generic pattern*, arriving at either of the optional design outputs offered by the *pattern*. After *Activity 5*, the process flows towards the minor

activity 'PARAMETERIZATION', from where we arrive at *Activity 6* ('*constructing a business coMponent*'), through the *'EFG' synchronization bar*.

Thus, the *'EFG' synchronization bar* reflects the three ways of deriving (within *SDBC*) a *business coMponent*: either without realizing *re-use* (by reflecting a *business process model* in a *business coMponent*), or by extending a *general business coMponent*, or by parameterizing a *generic business coMponent* (see Figure 5.9).

A constructed *business coMponent* is then to be reflected in a *software specification model*; hence, we arrive at Activity 7 ('*deriving a software specification model*'). A sound mapping is to be accomplished allowing for a precise reflection between the two. Both the *business coMponent* and the resulting *software specification model* should undergo at least structural and dynamic validation [54]. This is indicated by the label 'validation', positioned along the line between *Activity 6* and *Activity 7*.

Regarding the *software specification model*, as mentioned before, depending on the *granularity* of the source *business coMponent*, the model could or could not refer to a particular *software coMponent* (Figure 5.7). The question of software *granularity* is to be addressed particularly from the perspective of the *software* system-to-be. Usually, a derived *software specification model* is to be reflected in more than one *software coMponents*. So, progressing from *Activity 7* to Activity 8 ('*elaboration*') comes through the minor activity 'DECOMPOSITION' (indication of the need to *decompose* the *software specification model* into more than one *software coMponents*). However, in the cases in which no *decomposition* would be necessary, the *software specification model* is considered itself being a *software coMponent*.

Once identified, a *software coMponent* needs to be specified in more detail – further elicitation should be provided concerning the *coMponent's* entities and interactions. So, once identified and specified, a *software coMponent* should undergo *elaboration* (*Activity 8*).

And in the end, after a sufficient (see below) number of *software coMponents* have been identified, specified, and elaborated, they should be *integrated* (Activity 9) in the process of specifying the functionality of the software *system*-to-be. Hence, there is a more special relation between *Activity 8* and *Activity 9*; an indication for this is the symbol positioned on the line between those activities, showing that many *software coMponents* would be necessary that would represent together a sufficient input for specifying a complete model of the software *system*-to-be. However, it is often not easy to provide guidelines on how to decide what particular *software coMponents* represent a sufficient input for specifying the software *system*-to-be; this decision is often subjective and/or intuitive; anyway, we adopt in *SDBC* the relevant general guidelines provided in [5], related to the *component-based product-line engineering* [4].

So, after considering the *SDBC activity model*, we proceed to the *SDBC input/output model*. It is depicted in Figure 5.11. As seen from the figure, the starting input for applying *SDBC* is any (informal, unstructured) *description of the enterprise system* to be considered (Input 1.1), including *domain-imposed requirements* possibly representing *norms* [43]. The description might be *textual* or it might be a *graphical model*, a conversation or any other form. The first activity's output (Output 1.1) should be a *structured description of the studied system*. This description should thoroughly reflect the considered business reality; next to that, the description must be precisely *delimited*, as mentioned before. As seen from the figure, such a structured and delimited

description might be stored in a bank (**D bank**) from where to be usable also in other relevant modeling tasks.

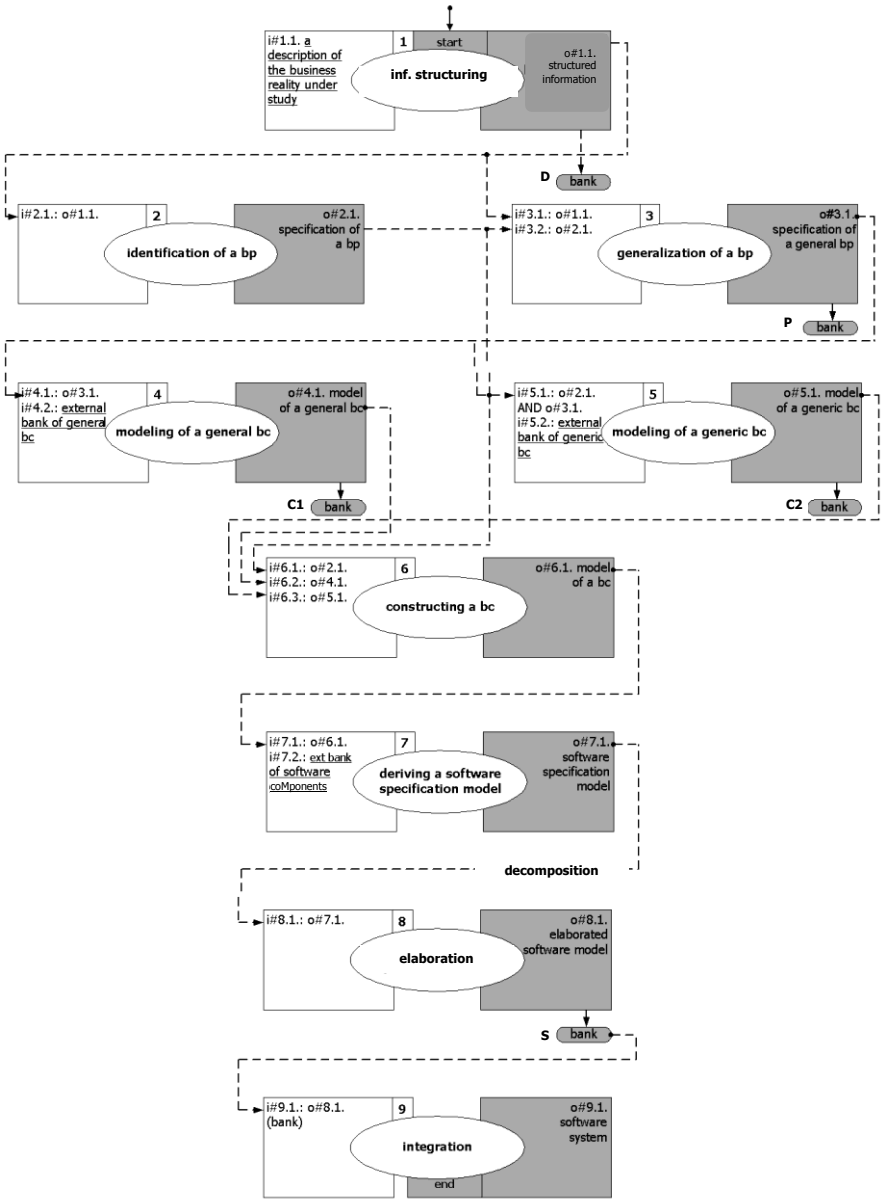


Fig. 5.11. SDBC – input/output model.

Such a description could be used as an input for either *Activity 2* (Input 2.1) or *Activity 3* (Input 3.1) (either for identifying a *business process* or for building a *generalized business process model*). Building a *generalized business process model* could be done as well based on an identified *business process* (Input 3.2). An indication for this is the line between *Activity 2* and *Activity 3*.

A *generalized business process model* could be stored in a bank (**P bank**) for multiple uses. It could also be used as an input for constructing (*Activity 4*) a *general business coMponent* (Input 4.1). As seen from the figure, *general business coMponents* could also be taken from an external bank (**C1 bank**) (Input 4.2). A constructed *general business coMponent* could be either stored in a bank – *C1 bank* (for use in other project(s)) or used as an input for the construction (*Activity 6*) of a *business coMponent* (Input 6.2). As seen from Figure 5.10, this comes through *extending* the *general business coMponent*.

Regarding the modeling of a *generic business coMponent*, it should be based on a *generalized business process* model AND at least two (Figure 5.10) models of particular *business processes*; this concerns Input 5.1, Figure 5.11. *Generic business coMponents* could also be taken from an external bank (**C2 bank**). As seen from Figure 5.11, a constructed *generic business coMponent* could be either stored in a bank (*C2 bank*) (for use in other project(s)) or used as an input for the construction (*Activity 6*) of a *business coMponent* (Input 6.3). As seen from Figure 5.10, this comes through *parameterizing* the *generic business coMponent*. And finally, as seen from Figure 5.11, the third possible input (Input 6.1) for the construction of a *business coMponent* is a *business process model* (*Output 2.1*).

Deriving a *software specification model* (from which *software coMponents* could be identified, by applying decomposition, as already mentioned) is based either on a *business coMponent* constructed in the above proposed way (Input 7.1) or on import of *software coMponents* from an external bank (Input 7.2).

Each of the derived *software coMponents* should be *elaborated* (*Activity 8*; Input 8.1) in terms of structural, dynamic, and data aspects (in order to bring sufficient elicitation for the further software design activities, as already mentioned) and stored in a bank (**S bank**). From there, *software coMponents* will be taken (Input 9.1) and integrated for the purpose of *specifying the software system-to-be*.

A *specification model* of a software *system* represents the final output (Output 9.1) of the *SDBC* approach. Hence, the *end point* is reached and this is indicated by labelling *Activity 9* with 'end', as stated already.

In summary, we have outlined the *SDBC* approach, by means of the *SDBC activity model* and the *SDBC input/output*, developed exclusively for that purpose. In the following section, we will present the notations to be used for the *SDBC modeling* itself.

5.2 SDBC Notations

SDBC is an approach that has its *underlying theoretical roots* and also its *process outline* elaborating on *what* and *how* to do in implementing the approach – all those have already been introduced.

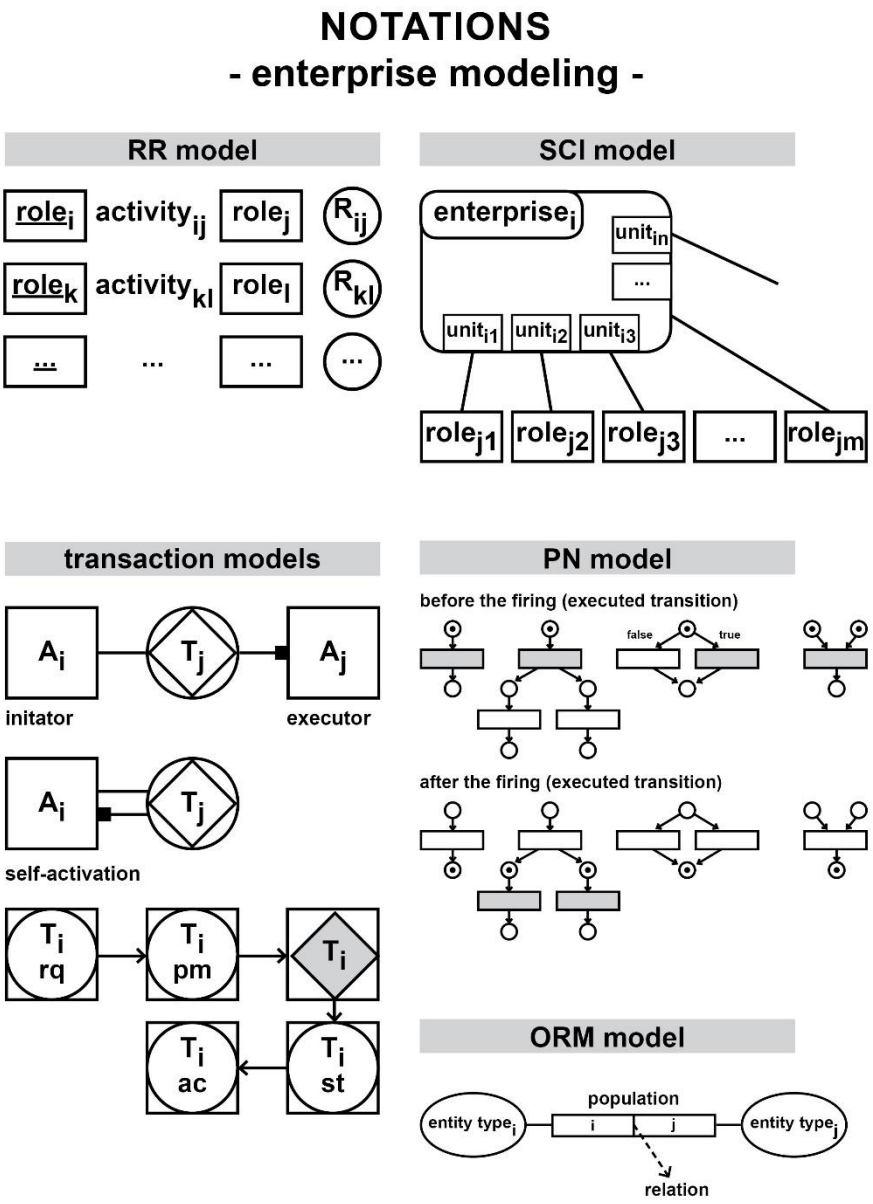


Fig. 5.12. SDBC – enterprise modeling notations.

Hence, it should be possible to apply *any* (graphical) *notations* in realizing *SDBC* modeling as far as they conform to the approach's *underlying concepts*. Still, we are proposing particular graphical *notations* for *SDBC* modeling, making sure (based on previous research [54]) that those *notations* are well aligned with *SDBC's underlying concepts* and *supportive theories*. For this reason, we recommend using those *notations* although we do not claim that they are exclusive with regard to the implementation of *SDBC*.

Since *SDBC* has two 'grounding points', namely *enterprise engineering* and *software engineering* (see Figure 5.1), we will firstly present in this section several most important *enterprise-modeling*-related *notations* (Figure 5.12) and then we will present several most important *software-specification*-related *notations* (Figure 5.13).

Those notations will be featured in the following chapter, when the SDBC approach will be demonstrated by means of a case study and illustrative examples.

With regards to the *enterprise modeling notations*, as it is seen from Figure 5.12:

- The RR ('RR' standing for 'Roles and Relations') model (or chart) that is depicted up-left in the figure, reflects a RELATION between TWO roles (meaning role types), assuming that any MORE COMPLEX relation can be decomposed in a number of relations that are between two roles. In the chart, the two roles are put in boxes and the label of the corresponding relation is put in between, while the role pointing to the realization of the relation is underlined. For example, if the two roles are 'expert' and 'customer', and the relation is 'realize expertise', then we should underline the role 'expert' because it is the expert who realizes the expertise. Finally, each role-to-role relation is given a unique code, as it can be seen from the right side of the RR model visualization.
- The SCI ('SCI' standing for 'Structuring the Customer Information') model (or chart) that is depicted up-right in the figure, assumes an INSTANTIATION with regard to the addressed enterprise and elaboration with regard to its structure. In the chart, the addressed enterprise is modeled in a rounded rectangle with smaller rectangles inside, corresponding to the internal organizational units of the enterprise. Outside the rounded rectangle, there are rectangles that correspond to the roles (not instantiated) collaborating with the addressed enterprise, in general, and to its corresponding internal units – in particular. For example, ABO Supermarket in Sofia, has a number of Departments including Finance department, Sales department, Marketing department, and so on, while at the same time, there are a number of related ABO-external role types, such as Customer, Supplier, Insurer, and so on.
- Those relations (see above) are to be reflected in the end in corresponding *transactions* (see *Definition 5*) that in turn are modeled using notations as presented middle-left in the figure: we have the *initiator* and the *executor* put in boxes while the *transaction* itself is modeled as a disk+diamond, conforming to *enterprise ontology* [19]; the small black box in the chart is to indicate who the *executor* is. Further, modeling self-activation is also possible, assuming that the initiator and the executor are the same 'entity'. Finally, zooming-in with regard to a *transaction* is possible, such that all corresponding coordination acts are revealed (modeled as a disk+box) as well as the corresponding production act (modeled as a diamond+box), with 'rq', 'pm', 'st', and 'ac' meaning 'request', 'promise', 'state', and 'accept', respectively.

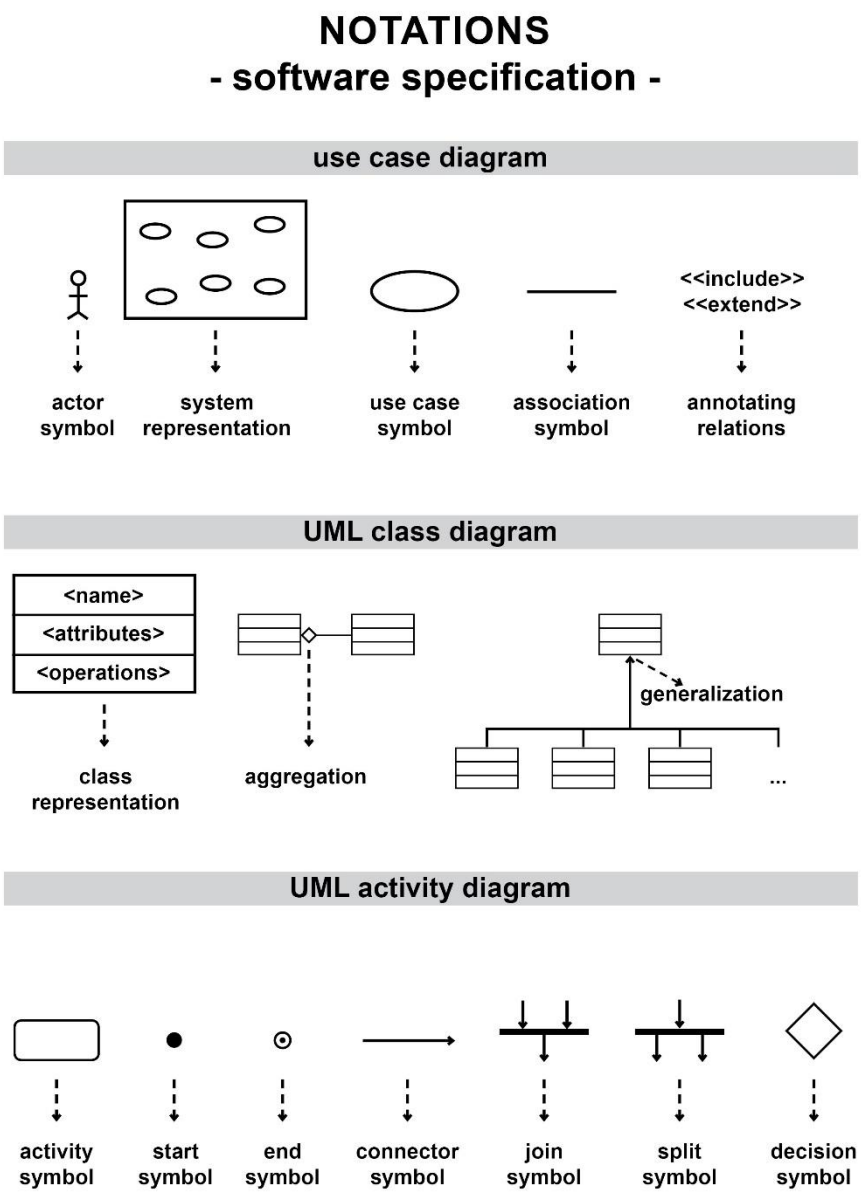


Fig. 5.13. SDBC – software specification notations.

- With *transactions* making up corresponding *business processes* (see *Definition 6*) which in turn are to be also modeled in terms of overall *behavior*, we need appropriate notations and we have opted for the Petri Net (PN) notations [54], depicted middle-right in the figure, and allowing for modeling sequential behavior, parallel behavior, decision points, and so on, as shown there.
- Finally, with regard to factual (data) modeling, we have opted for ORM (the Object Role Modeling), as presented in [54], that is presented at the bottom of the figure. Using ORM notations, one could model (similarly to the RR notations) two TYPES of entities/roles and a relation between them. What is special about ORM is that it is about POPULATING the model in terms of data corresponding to instantiations. For example, if we have the types 'Professor' and 'Department', and the relation 'works for', populating the model would mean instantiating as follows: Professor John Smith works for the Computer Science department, Professor Ben Starkey works for the Physics department, Professor George Ashley works for the Chemistry department, and so on.

With regards to the *software specification notations*, as it is seen from Figure 5.13, they are based on *UML* since the *Unified Modeling Language* is claimed to be a *de facto* a notation standard with regard to the specification of software [54,74], and in particular:

- The *use case* diagram is appropriate for capturing the functionality of the software system-to be at high level, and for this reason, the system is represented as a number of *use cases* (ovals) in a rectangular area, surrounded by the primary *actor* (the *system*'s customer) and other *stakeholders* with related interests. There may be relations among *use cases* or between an *actor* and a *use case* – those are represented by lines (association symbol), as the figure shows. Finally, there are two stereotypes considered, namely 'include' and 'extend'.
- The *UML class diagram* is featuring classification and is capable of modeling classes (specifying attributes and operations accordingly), aggregation, generalization, and so on, as shown in the figure.
- The *UML activity* diagram is capable of modeling overall system behaviors, having explicit notations that allow to model sequential behavior, parallel behavior, decision / join / split patterns, as shown in the figure.

An in the end, it is to be noted that neither the enterprise modeling notations considered above (see Figure 5.12) nor the software specification notations considered above (see Figure 5.13) reflect exhaustive lists of notations since this is not considered necessary. The notations we have presented are possible notations of choice when applying SDBC and are expected to 'cover' most typical modeling situations.

IN SUMMARY, in this chapter, we have presented the SDBC approach, elaborating its foundations, outline, and recommended notations. In this way, we have shared our ideas on how enterprise engineering and software engineering can be brought together, driven by the goal of specifying software. In the following chapter, we will demonstrate this, by means of a case study and illustrative examples.