

Chapter 2

From Demand to Supply: Layers and Model Quality

Although the development of an appropriate Business Architecture is essential in realising organisation's strategy, reflection on the desired information system service functionality and information system architecture is needed as well. Moreover, from an enterprise information systems development perspective, we need to help software engineers in creating generic, flexible and (therefore) future-proof systems. Focusing in the first place on the demand side of a software project and ensuring the quality of enterprise models is one means to achieve this. At the same time, bridging the gap between demand and supply is also of prime importance. Therefore, the MERODE approach builds on two main principles which, next to being of benefit to the demand side, benefit to the supply side as well.

The first of these principles is the layered organisation of a system. This is a well-known high-level architectural pattern for software architecture, but MERODE also follows this principle during requirements engineering. As a result, the method guarantees an extremely large degree of independence between the domain model and the information system services. This feature further improves flexibility and extensibility of the resulting system. Moreover, the method's advice is to focus first on the enterprise layer whereas many business analysts would gather requirements in an information system service driven way, starting with the identification of desired information system support and user interfaces.

A second main principle that ensures a smooth transition from demand to supply is the quality assurance of the artefacts delivered by the enterprise architects to the software builder. This focus on quality is the natural consequence of the transformation-based approach. As models need to be precise and complete enough to be manipulated by a transformation engine, this transformation-based approach poses stringent requirements on the quality of the specifications.

In the next section, we will first explain the essentials of the layered information system architecture and further motivate how this benefits the enterprise architect and the constructor of software. The second section will then outline essential aspects of quality assurance and explain how this contributes to modelling practice and to the construction of software.

2.1 A Layered System Architecture Both for Requirements and System Architecture

2.1.1 Enterprise, Information Services and Business Process Layer

When gathering requirements, users express their desires about many aspects (functional and non-functional) of the system-to-be. Moreover, as enterprises are functioning in a highly dynamic environment, requirements and expectations towards information system support change rapidly too. The need for flexible and adaptable systems is therefore very real. One way to contribute to the adaptability of a software system is to organise it as a set of layers.

From a construction perspective, the guiding principle of the organisation of the layers is that each layer is able to use the functionality of inner (and more stable) layers but is agnostic of the functionality of outer layers. As a result of this, a modification of an inner layer will have a ripple effect through all outer layers. However, an outer layer can be modified or stripped off and replaced by a new version, without affecting inner layers. Consequently, the kernel of the system should provide stable functionalities that do not change very often. Around this kernel, layers are arranged such that the outer layers incorporate functionality with a lower level of stability than the inner layers they are built on.

This idea of layering software code along the dimension of stability obviously also applies to enterprise information systems but moreover can easily be applied to requirements as well. If we consider different types of requirements, we see that essential business concepts tend to be quite stable. Business rules and information needs will change more frequently and user interfaces and business processes show the highest change rate. Logically then, software components that embody the stable part of requirements should form the kernel layer of a system, whereas software components that embody highly variable aspects requirements should be put in the outer layer.

Domain modelling focuses on capturing the requirements pertaining to the essential business concepts that remain valid even if there is no information system at all: ideally a domain model is fully information system agnostic. The domain object classes form a stable core on top of which information system functionality is modelled as a set of independent information system services. So the domain model provides the model for the stable kernel of the enterprise information systems: the enterprise layer. Around this stable layer built according to the domain model, we put an outer layer of information system services (ISS for short). In order to mirror the state of the real world in the enterprise information system, input services allow capturing information about events in the real world and register this information as new, modified or terminated business objects. Output services enable user to ‘look into the mirror’ and extract information about the enterprise through output services such as reports, dash boards, etc. An additional user interface layer glues the individual services together and provides the user with facilities to trigger, interrupt

and resume the execution of services. If in addition we take care of keeping the information system services perfectly independent from each other by requiring that all interaction goes through the enterprise layer, we obtain a highly modular and flexible system: information system services can be plugged in and out without affecting the enterprise layer.

On top of the information system service layer, we find the business process layer. Business processes define the work organisation as sequences of tasks to be performed by actors. In order to perform their tasks, actors may use the information system services. If the business processes are supported by software such as a business process management system, this can be implemented as a third layer on top of the information system services layer (see Fig. 2.1).

If we position this set of layers against the models of the Business Architecture, we notice that the Business Architecture is actually spread across the outmost and the innermost layer. Indeed, the Business Architecture is defined by on the one hand activity models (business process models) and on the other hand by a model of the core business concepts (the domain model). Business process models belong to the outer layer. The domain model describes the core business objects about which the business needs information, the relations between these business objects and their individual behaviour. Because the business objects identified in this domain model are key concepts in the Enterprise Architecture, the bottom layer or enterprise layer is sometimes also called the ‘Business Object Layer’.

Finally, this layered information system architecture should not be confused with a three-tier architecture for distributed systems, a concept that belongs to the Technology Architecture. In that kind of three-tier implementation architecture, a separation is made between presentation, application logic and data. In terms of the aforementioned layers, the information system service layer contains the application logic as well as the application data and the enterprise layer contains the business object data as well as the business logic (see Fig. 2.2).

The separation between an enterprise layer, an information system service layer and a business process layer leads to a natural partition of systems in three weakly

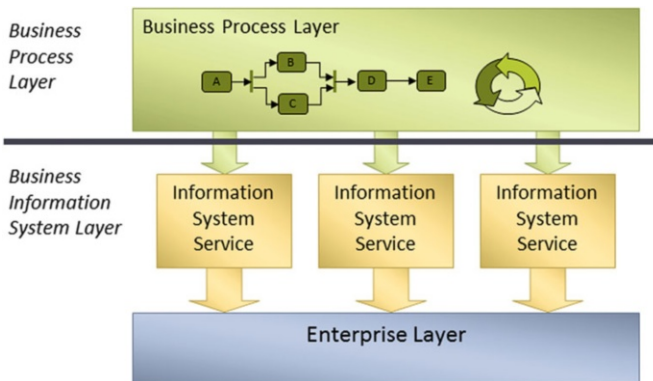


Fig. 2.1 Architectural layers

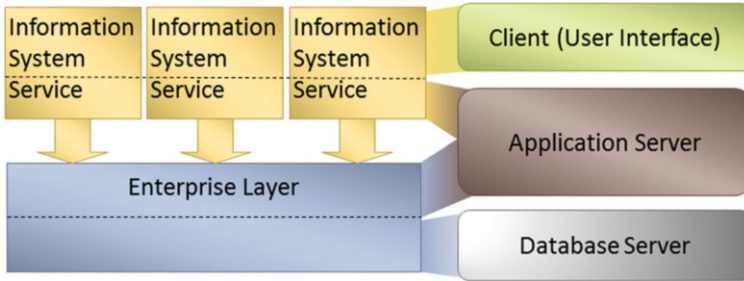


Fig. 2.2 Layers versus Application and Technology Architecture

coupled subsystems, each corresponding to a single aspect of the problem and evolving independently of all other aspects. The three independent aspects are:

- The business logic (the enterprise layer)
- The work organisation (the input services and the business process layer)
- The information needs (the output services)

As a result, on the construction side of a project (i.e. the supply side), this layered approach ensures the adaptability of the software and hence lowers maintenance cost. In response to the changing environment and the need for innovation, organisations need to be able to swiftly adapt their business processes. As these are put in the outmost layer, they can be adapted without ripple effects on the lower layers. Innovation may also drive changes in information system services: new technologies and the need for more or other information may require the implementation of new or adapted input and output information services. Logically, such new services may induce changes in the business processes as well. Again, when information system services are implemented as independent services, ripple effects will remain limited. Only business processes using these services will be affected. Finally, changes to essential business concepts will affect the kernel of enterprise information systems, which is the natural consequence of the fact that they embody the ‘essence’ of an organisation. Changes to the enterprise layer may affect information system services as well as business processes.

Next to significant productivity gains during maintenance, the layered architecture also offers better insight in the cost of changes. Changes in the enterprise layer may affect all outer layers. Such changes are therefore more expensive and should therefore be scarce. Changes to the business process layer and the information system layer do not affect the lower layers and should be easier to manage. As information system services can more easily be plugged in and out, changes to these services are easier to perform and therefore less expensive. In addition, because information services are always based on a specific part of the domain model, the set of services that is affected by a modification of business objects is easy to determine. This allows to better estimate the cost of modifying the domain model.

A final advantage that benefits both the enterprise architect and the software builder is that the layered approach using a domain model as kernel also allows

improved management of servicing users operating in a similar context, but with different preferences. An example is the development of a software package for different customers that operate in the same business domain. The layered approach facilitates the development of a product line in the following way. The domain model can be offered as a unifying kernel, common to the whole product line. In this way, the domain model is an instrument to harmonise requirements across users and to reduce variability. It creates a common language that defines a common view on essential business objects. If one succeeds to let users agree on a common domain model, variability can then be realised by offering each user a set of information system services tailored to his/her preferences. Higher levels of variability can be achieved by adding proprietary extensions to the domain model and corresponding information system services.

2.1.2 Layers Versus Requirements Gathering and Engineering

The order of the layers is in principle independent of the steps in the requirements gathering and engineering phase. In practice, however, the requirements gathering step occurs most frequently in a top-down manner: many software engineering methods will propose to elicit requirements by interviewing users about their business processes and the use cases for the information system. During requirements elicitation, when users formulate system requirements, they usually tend to mix domain knowledge specifications with information system service requirements. Figure 2.3 shows a subset of specifications for a medical software package for general practitioners. It is clear that only statements 2 and 5 are defining the

1. The software should be web-based.
2. Each consultation is for a single patient. If a visit concerns issues of several patients (e.g. a parent coming with two children), treating the issues of the different patients is considered as having one consultation per patient.
3. The general practitioner should be able to consult reports on medical examinations at the hospital through a secure connection, but only for patients that gave a prior consent for this.
4. The general practitioner should be able to send reports on a medical examinations to a colleague through a secure connection.
5. A medical report is always for a single patient and a single examination.
6. It must be possible to access the file of a patient by means of a maximum of five mouse clicks.
7. Shortcut keys must be provided for quick access to system functionalities (e.g. ctrl + P for printing).

Fig. 2.3 Example of user requirements

domain, while statements 3, 4, 6 and 7 are a description of the required information system services. Finally, statement 1 reflects a technology requirement. One of the most important tasks for a system analyst is then to separate the different types of specifications in order to identify business requirements and information system service requirements.

During requirements engineering, we will classify the different requirements according to the different layers. This means we will separate requirements that describe objects in the problem domain, information system service requirements and business process requirements and put all these requirements in the appropriate layer.

Requirements engineering methods do not always have such clear distinction between domain modelling and information services modelling. Often, the specification of a business object contains not only the core business attributes and routines but also input and output services. We advocate that such a grouping should not be allowed at the modelling stage. If specific reasons apply, the technical architect may still choose to group things that belong to different layers in a single software component when transforming the models to implementation.

An important characteristic of MERODE is that, next to the organisation of requirements into layers, MERODE advises to engineer requirements in a bottom-up way and to focus on creating the domain model first. The benefits of doing so are that this will:

- Foster problem orientation rather than solution orientation and in this way avoid an early implementation bias
- Contribute to generic thinking and hence more future-proof systems
- Stimulate the development of a common language across business users, hence contributing to a better business-IT alignment

In essence, the method proposed in this book is a requirements engineering method. So it concerns the initial phases in software development, where users are questioned on their expectations about the new information system. The focus of MERODE is on requirements *engineering*¹ rather than requirements *gathering*. Requirements gathering can be considered as collecting the pieces of a jigsaw puzzle, whereas requirements engineering pertains to sorting pieces, throwing away irrelevant pieces, detecting and searching for missing pieces and making sure that everything fits together. We already explained that user requirements differ in stability depending on the system aspect they pertain to and therefore belong to different layers. That's one aspect of 'sorting' the pieces of the jigsaw puzzle.

¹ Some authors define requirements engineering as consisting of requirements gathering and requirements analysis. In this book we define requirements engineering as a phase that follows the requirements gathering step and that not only consists of analysing the requirements but also of building a model where every piece of requirements fits in well. Because of this construction aspect, we prefer the term requirements engineering over the term requirements analysis.

In addition to this ‘sorting’ of requirements, this method also advocates a *focus on domain modelling first* in order to foster problem orientation and to ensure the genericity of the system to be. If one simply lets users tell their requirements, one will notice that most users tend to formulate solutions rather than problems and only report on their current use of a system. As a result, simple requirements gathering is indeed not enough: it should be followed by a thorough analysis and engineering phase to make sure that the final model captures the right problem in the correct way.

One of my former students told me this wonderful story about users formulating solutions rather than their true problems:

As a young analyst, one of his first assignments was to investigate the request of a user, who requested that the pink fill of the header of a form would be made yellow. Upon asking for the reasons for this change of colour the user explained that the form was used for some contractual agreement (e.g. a quote). It was therefore faxed to the customer who had to sign it and fax it back. By going twice through the fax machine, the pink fill turned black, making the customer's signature illegible. So the user had figured out that if pink was changed into yellow, this would solve his problem.

This little story beautifully illustrates that one should always search for the real problem behind the problem formulated by the user. In the above case, the true problem was not about colours but about obtaining a customer's signature, which was eventually solved by providing the means for electronic signatures, making the use of the fax obsolete and contributing to business process optimisation. Users also tend to describe their requirements based on their current use of the system: the user is used to work with a faxing machine and therefore formulated his request in terms of a solution with a faxing machine. This story illustrates how problem-oriented thinking contributes to the development of generic and future-proof systems.

This problem-oriented thinking receives little attention in current practice. Requirements gathering is supported (and encouraged) by many techniques such as semi-structured interviews, storyboards, scenario walkthroughs and use case analysis. Most of these techniques focus on the way information systems are currently used by the users and/or desired future uses of an information system. The results can, for example, be documented using the technique of use case diagrams from the UML.

Documenting the desired ‘uses’ of a system is without any doubt an essential aspect of requirements gathering. But in order to ensure the creation of generic and future-proof systems, a major point of attention is to pay attention to the fact that a system should not only support today's use cases but should also be capable of supporting tomorrow's use cases.

Suppose, for example, that when interviewing users, the first one comes up with the request to calculate $1 + 1$, the second one needs the calculation of $15 + 15$ and the third of $7 + 7$. What the users probably need is some facility to add numbers, either twice the same or possibly two different numbers. Maybe future uses even motivate the development of a simple calculator, a scientific calculator or even a full computer. Although each of these solutions offers increasing genericity, they all come with a different development cost and a price in terms of ease of use.

Unfortunately, many users are limited in their ability of generic thinking about information systems, mainly because they are unfamiliar with information system design or because they are unfamiliar with the potential services that can be realised with information technology. As a result, they may fail to envisage interesting future use cases. Therefore, discovering possibilities for genericity is an important task of a good business analyst. The business analyst can achieve this by focusing on the problem, rather than on the solution. Obviously, solution-oriented thinking cannot be completely avoided. But one should always keep in mind that different solutions address different problems: the simple calculator, the scientific calculator and the computer all address different problems with different levels of complexity. The business analyst is therefore responsible to discuss and evaluate alternative options with the users: genericity comes with a price, and it remains up to the user to decide how much genericity he/she is willing to pay for, depending on the problems he/she expects to face in the future.

Analysing potential areas for generic and future-proof solutions and assessing the impact of different solutions on the work organisation of an enterprise require thorough requirements analysis and engineering on top of requirements gathering.

Even when focusing on current *and future* use cases of an information system, the use case perspective on requirements gathering still suffers the problem mentioned before that use cases are often used to collect requirements about all aspects of the system, in this way blurring together domain model aspects, business rule aspects, information system aspects, user interface aspects and workflow aspects (see Fig. 2.4). So, whereas use cases are a very practical technique for requirements gathering, use case modelling needs to be followed by an engineering phase during which requirements are categorised according to the identified layers (the ‘sorting’ part) and further analysed and engineered to define good models.

The question remains then how such ‘engineering’ of the requirements should be done. As pioneered by the Jackson Systems Development (JSD) methodology [38], modelling the real world or the ‘domain’ allows to construct a solid core for a software system. The complexity of most systems is caused by the complexity of

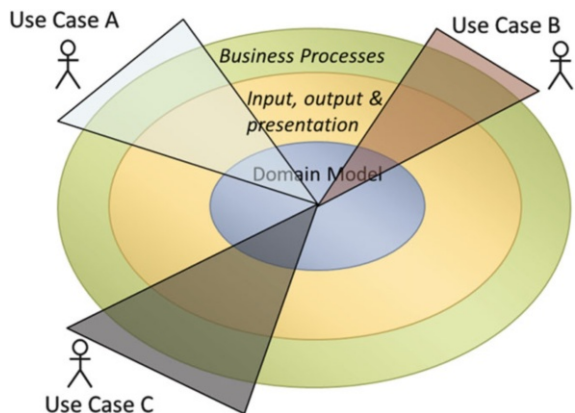


Fig. 2.4 Use cases blur different layers together

the reality they have to deal with. In addition, statements about the required information system functionality always have some underlying assumptions about the real world. It thus makes sense to build such a real-world model prior to the development of an information system. On the other hand, it makes no sense to build a real-world model without having some information system functionality in mind. Similarly the required information system services are mostly determined by the business processes, and reversely, business processes are designed with some available information system service in mind. Hence, both models are an integral part of an analysis model with equal importance.

In this book, following the idea of Jackson, the focus will be on domain modelling first, meaning that the requirements *engineering* process will start with the creation of the domain model. This is not such an evident choice: for users it is much easier to start telling about their business processes. However, if the entire requirements engineering process is executed as a top-down process (starting with business processes, then identifying the required information services to support the tasks and then deriving the required business objects from there), the danger exists that the domain model, which is after all the kernel of the system, will be entirely focused and tailored to the currently envisaged business processes. This way of working is quite natural and easy but severely limits generic thinking about information systems. A requirements *gathering* process can start with business process modelling. However, the requirements *engineering* process should start with domain modelling, then continue with possible information system services that can be build and finally end with defining the business processes to-be. As domain modelling focuses on essential business concepts and business rules, creating an enterprise-wide conceptual model forces the business to develop a common view on how the business operates. In this way domain modelling helps to avoid the fragmentation due to a use case by use case view and stimulates focusing on the true problem and devising generic solutions. Also, the focus on domain modelling can help to overcome the limitations of a project-based view and hence facilitate the alignment of IT with business needs.

In practice, the entire requirements gathering and engineering process will be as follows:

1. Requirements gathering starts with identifying current and envisaged business processes. During this first step, a first set of information system services and business objects are filtered out of the requirements as the result of the ‘sorting’ of the pieces of the jigsaw puzzle.
2. The requirements engineering phase starts with constructing a first version of the domain model.
3. The completeness and validity of the domain model is verified by checking whether this model provides all that is needed to support the envisaged information system services and business processes identified in step 1. Potentially new information system services and new business processes are identified.
4. The domain model is revisited such as to account for incompleteness and validity problems identified in step 3.
5. Steps 3 and 4 are repeated until a stable domain model is reached.

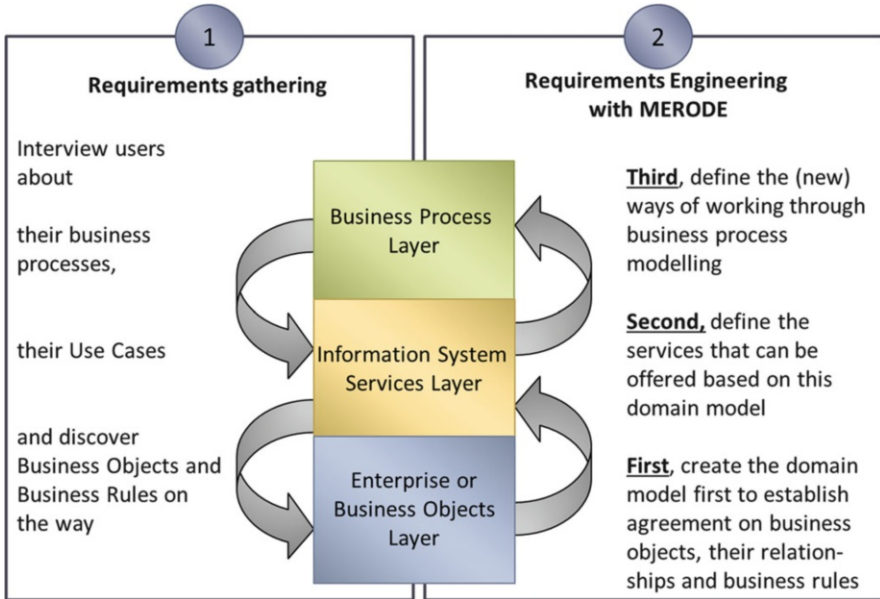


Fig. 2.5 Requirements gathering versus requirements engineering with MERODE

6. An inventory is made of the required information system services (by-product of step 3, further completed in this step).
7. The business processes are finalised (by-product of step 3, further completed in this step).

Figure 2.5 gives an overview of how requirements gathering is a top-down activity through the layers, whereas according to MERODE requirements engineering follows a bottom-up approach through the layers (modulo the necessary iterations to achieve a satisfying level of validity and completeness).

2.2 Formal Verification of Models

2.2.1 Model Quality

Hidden, unspoken requirements, ambiguously formulated requirements or a lack of agreement on requirements may all affect software development projects in a negative way. The quality of requirements and hence also of an enterprise model is of prime importance for the successful development of enterprise information systems. A wrong, incomplete or ambiguous domain model on which there is no agreement reflects the fact that the business is not clearly understood in the same way by all stakeholders.

To achieve a certain level of quality in the production of software, software engineering takes a lot of inspiration from the manufacturing process. In manufacturing, one way of guaranteeing the quality of final products is to standardise the production process. For example, in an assembly line, the production steps are completely standardised. As a result, all end products are identical. Standardising the production method is a way to ensure a certain level of quality for all delivered products. Research on software process improvement (SPI) and models such as ISO 9000 and the capability maturity model (CMM) [73] all focus on the software development *process*. By rigorously defining tasks and activities in this development process, such frameworks hope to ensure the quality of the final product. Information systems development is however quite different from manufacturing: each development project aims at the construction of a system that is perfectly tailored to the needs of the users. Therefore, it is impossible to completely standardise the software development process, although many software development frameworks would like us to believe otherwise.

This impossibility to achieve quality by focusing on the process only should however not prevent us from demanding a certain level of quality for the final result. The way to do this is to establish a number of quality criteria that must be met by the final result. The same is true when I hire someone to clean my house: I do not tell this person to sweep twice here and three times there; I only want my house to be clean at the end. The exact intermediate process is of lesser importance. For the same reason, this book does not focus primarily on development processes. In contrast to many other approaches, this book focuses on a number of strict criteria that will be defined to ensure the quality of the outcome of the process: the final outcome of a process is subject to a list of very stringent norms that will qualify it for further development. In particular, as the focus lies on domain modelling, the method in this book will give some guidance on the process of modelling but will put most emphasis on the quality of the produced models. Concentrating on the quality of the modelling process is like ensuring the cleanness of water pipes. This is beyond doubt an important element in quality assurance. But as judiciously remarked by J. Voas [97], clean pipes can produce dirty water. And thus, we believe that an essential component in the development of a modelling method is the elaboration of tests that can assure the ‘quality of the water’, hence, model quality. This way of working is equivalent to Mintzberg’s coordination mechanism by standardisation of output, as opposed to coordination by standardisation of process [59].

Furthermore, in a model-driven approach, the outcome of domain modelling is a platform-independent model that will be used as input for transformations to other models or code. As it should be possible to *automatically* transform such a model to models or code by means of model-to-model and model-to-code transformations, the model should be absolutely unambiguous and precise. Indeed, in case of *automated* transformations, there is no human intervention, and so remaining errors and ambiguities cannot be corrected during the design and implementation step. Transformation engines obey the law ‘garbage in, garbage out’.

In the past, different frameworks for model quality have been proposed. Modelling quality frameworks define levels of quality that a certain model should accomplish. There are many frameworks that attempt to bring order and enlightenment to the quality of conceptual modelling representations and to the quality of the conceptual modelling process. Some frameworks stand out from the mass: the first is a framework introduced by Lindland, Sindre and Sølberg [47] known as SEQUAL in its latest versions [43, 44]. It builds on Morris' semiotic theory and defines several quality aspects based on the relationships between a model, a body of knowledge, a domain, a modelling language and the activities of learning, taking action and modelling. The second is a framework developed by Wand and Weber based on Bunge's ontological theory (the Bunge–Wand–Weber representational model: BWW) [98]. Both frameworks have solid theoretical foundations, and they look at quality in conceptual modelling from two different perspectives. The SEQUAL framework focuses on the product of conceptual modelling whereas the BWW framework focuses on the process of conceptual modelling. Focusing on the integration of the SEQUAL and BWW frameworks in 2012, a third Conceptual Modelling Quality Framework (CMQF) [62] appeared. This framework is useful for evaluating the end result of the conceptual modelling process, the conceptual representation and the quality of the modelling process itself in terms of the knowledge aspects that come into play during the modelling process. A detailed explanation of this framework being out of scope for this book, we limit the definition of model quality to the basic cornerstones (for a complete explanation, see [62]).

Models can be considered as sets of statements that can be checked by comparing them to other statements or verifying them against a number of rules. The following basic types of quality can be identified.

2.2.1.1 Syntactic Quality

This refers to checking whether all statements expressed by a model are well formed, according to the conventions of the modelling language the model has been expressed in. For example, class diagrams are composed of classes (rectangles) and associations (connectors), and each association must always have at least two association ends. An association with only one end does not satisfy the syntax of a class diagram. If you compare this with verifying a text, syntax checking means that the text is verified for spelling and grammar. A good modelling language offers a clear definition of the syntax of its modelling techniques so as to enable this type of verification.

2.2.1.2 Intentional Quality

Next to obeying the syntax imposed by a modelling language, the model should also obey the intended meaning of the symbols. Symbols shouldn't be (mis)used to

represent something else than they are defined to represent. An architect shouldn't use the symbol for a window where he/she intends to have a door simply because he/she doesn't know how to draw a door. Likewise, in a domain model, a class/rectangle is intended to represent a set of business objects from a same type (e.g. the set of customers). Hence, using the class symbol to represent a collection of events (e.g. mouse clicks) is an example of bad intentional quality of a model.

2.2.1.3 Semantic Quality

This refers to level to which the set of statements in the domain model is a 'good' representation of the 'real world'. Semantics is sometimes also referred to as the 'meaning' of a model. Two types of semantic quality can be defined. **Completeness** means that all relevant statements about the domain have been included in the model. **Validity** means that all statements contained in the model are a correct representation of the domain at hand. Compared to the verification of a text, validity means that everything that is said in a text is true, and completeness means that no omissions are observed. Validity can further be subdivided into *external validity* and *internal validity*. External validity means that statements are verified against the truth as defined by the 'external' user. Internal validity verifies whether different statements that compose a model do not contradict each other. It is also referred to as 'consistency checking'. Stated differently, external validity asks the question 'Is this the right model?', whereas internal validity focuses on the question 'Is the model right?'.

Checking the semantic quality of models can be very hard. First of all, each individual diagram should be valid and complete. This in itself is already a challenge and requires a precise and unambiguous definition of the modelling techniques (preferably in a formal way). The intrinsic difficulties arise from the fact that different diagrams are used to define different aspects of the problem and that all these diagrams need to be consistent with each other. The first type of verification is called intra-diagram verification and the second type, inter-diagram verification. As an example, consider again the plans in Fig. 2.6. On the left hand is a floor plan with a separation wall between the sitting room and the dining room (indicated by the arrow). On the right-hand side is a transversal cut plan, used to define the height of roof, ceilings and windows and where the separating wall has been drawn at the other side of the dining room. This is an example of bad internal quality: it is a problem of inter-diagram inconsistency. Notice that consistent plans do not guarantee the external validity: it yet remains to be seen where the wall should be built in the real-world house according to the owner's requirements.

The same type of problem can appear in domain modelling. Consider an approach in which object behaviour is specified by means of one diagram and object interaction is specified in some other diagram. Whatever technique is used, in any way, the behaviour of the final system is a composition of all the individual object behaviour definitions and the definition of how objects interact with each other. For large systems it is difficult, if not impossible, to have a complete mental

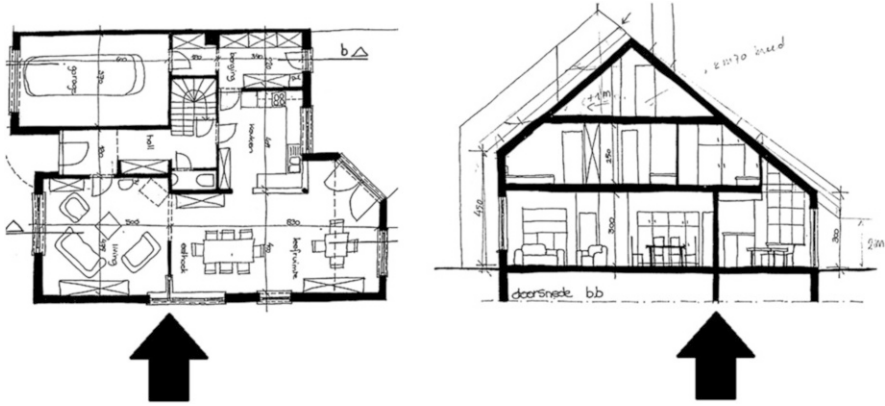


Fig. 2.6 Inconsistent plans for a house. Published with kind permission of © M. Van de Wouwer 2000. All rights reserved

model of what the behaviour of the overall system will be. As a result it is likely that the overall system will show some undesired behaviour such as starvation or deadlocks. Although some of these errors can be discovered by careful testing of *the implemented system*, it is much better if the conceptual modelling language provides the necessary formal underpinning enabling the use of verification tools and algorithms to check and *prove* the correctness of specifications *during model construction* [17, 82].

2.2.1.4 Pragmatic Quality

This quality dimension refers to the level to which the statements expressed by the model are correctly understood by the actors that will read the model. This quality dimension is often referred to as understandability of a model. Ensuring pragmatic quality can be positively affected by choosing the right modelling language and the right representation for the right public. Also, using adapted representations for the same conceptual model, depending on the targeted public, may positively influence model understanding.

As an example, converting the two-dimensional plans of a house to a computer-animated 3D model can help the owners to create a mental model of what their future house will look like.

2.2.2 Modelling Language

The choice of a modelling language will affect model quality in different ways. First of all, syntactic and intentional quality will be affected by the level to which

the symbols, their meaning and their composition rules are well defined and algorithms can be used to verify the syntax. Semantic quality will be affected by the level to which the languages provide algorithms to verify consistency and completeness. Finally, the level to which a language is widely known and understood will affect pragmatic quality.

Several authors have advocated the use of formal languages [15] to provide for better quality of models and the necessary support for model verification. An important observation is that formality on its own is not sufficient to achieve high quality.

Consider, for example, the statement:

John is a person.

Any student is a person.

Any employee is a person.

We can translate this statement to mathematics as follows:

$$\begin{aligned} \text{John} &\in \text{PERSON} \\ \text{STUDENT} &\subseteq \text{PERSON} \\ \text{EMPLOYEE} &\subseteq \text{PERSON} \end{aligned}$$

Despite this formality, we still don't know whether in our universe all people are either students or employees and we also don't know whether one can be a student and an employee at the same time. Formally, one still needs to check whether the following statements are true in our universe of discourse:

$$\begin{aligned} \text{PERSON} &= \text{STUDENT} \cup \text{EMPLOYEE} \\ \text{STUDENT} \cap \text{EMPLOYEE} &= \emptyset \end{aligned}$$

Although formality does not offer a guarantee for completeness of specifications, it does offer the major benefit that formal languages enable reasoning (in a mathematical sense) about properties of models [14]. This reasoning enables one to investigate models for omissions, ambiguities and contradictions. Eventually, one may even *prove* that certain anomalous system behaviour, such as deadlock, cannot occur [17, 82]. The formal underpinning of modelling techniques enables quality control at a level that is impossible to reach with informal techniques.

When talking about formal techniques, one nearly automatically thinks of complex specification languages with many mathematical symbols and that require a lot of training before they can be used. Fortunately, formal techniques are not necessarily mathematical specification languages but can be graphical techniques as well, provided that the syntax and semantics of these techniques are precisely described. The useful thing about formal languages is the reasoning they enable. This reasoning can fortunately be hidden behind more user-friendly graphical symbolic representations.

As language, ideally we would opt for a formal language, given its advantages for formal verification. However, such a language has the severe drawback that it would make models very hard to read, understand and validate by the users, jeopardising pragmatic quality. This book therefore opts for a domain-specific language, largely based on the widely known UML. As the UML is a general-purpose modelling language, it offers quite a lot of techniques and features, suited for all possible areas of software development: it can be used for models in the domain of enterprise information systems as well as for modelling real-time systems or embedded systems. It also supports the description of artefacts in all stages of the development process: from high-level conceptual models such as domain models down to the specification of design models close to implementation. Given that the language is widely known in the software development community, the choice for UML ensures (in terms of understandability) a much better fit with our users than a formal language. On the other hand, the multitude of constructs holds the danger of low intentional quality. Also, we still need the formality to be able to offer the users the power of formal reasoning and code generation. To achieve this, we limit our use of UML to the most frequently used techniques and complement the existing UML semantics with a further formalisation by means of process algebra. The formal reasoning power is kept hidden behind the known graphical representation of UML, such as to shield the business analyst from the complexities of the MERODE-process algebra. The basis of the MERODE domain modelling approach is the UML techniques for class diagramming and life cycle modelling (state machines), augmented with the object-event table (borrowed from James Martin's Information Engineering method [53]) to model object interaction. The MERODE approach is a domain-specific use of UML in the sense that:

1. It doesn't use all the UML techniques, but only a relevant subset for the context of domain modelling.
2. It doesn't use all the features of the chosen UML techniques, but is limited to the features relevant for domain modelling.
3. It complements the UML techniques with the CRUD-matrix of the Information Engineering approach.
4. It complements the known semantics of UML with a more thorough formalisation by means of a CSP-like process algebra [35, 82, 17].

2.2.3 Tool Support for Quality Control

From the point of view of quality of the modelling process, the formal definition of a modelling technique will allow for correctness checking. As models serve as plans for software development, error-free models substantially reduce development costs. But evenly important, the precise definition of the syntax of a modelling technique is a prerequisite for the development of a supporting computer-aided software engineering (CASE) tool: the precise definition of semantics and the availability of a formal procedure for checking consistency between views allow

to add intelligence to such a CASE-tool. Without these features, CASE-tools cannot offer much more support than simple diagram editing. CASE-tools in general and modelling tools in particular are for models what a word processor is for a text. They offer the possibility to capture requirements in the form of visual models, called diagrams. Diagram editors range from simple drawing tools to intelligent requirements engineering tools, capable of thorough quality checks.

For the verification of consistency (internal validity), three basic approaches can be distinguished [87]. A first approach is consistency by analysis, meaning that an algorithm is used to detect all inconsistencies between two deliverables (each representing a particular view of the same system), and a report is generated thereafter for the developers. In this kind of approach, the requirements engineer can freely construct the different views of the system. At the end of the specification process or at regular intervals, the algorithm is run against the models to spot errors and/or incompleteness in the various views. The verification can be done manually, but obviously building the algorithm into a CASE-tool will substantially facilitate the consistency checking procedure.

The second approach can be denoted as consistency by monitoring, meaning that a tool has a monitoring facility that checks every new specification against the already existing specifications in the CASE-tool's repository. Whenever an attempt is made to enter a specification that is inconsistent with some previously entered specification, the new specification is rejected. The advantage of this approach is that the model is constantly consistent. Whereas the first approach puts the burden of correcting inconsistencies on the requirements engineer, the second approach avoids the input of inconsistencies. At the end of the specification process, the model must still be verified for completeness. The possible disadvantage of this approach is that a too stringent verification procedure will turn the input of specifications into a frustrating activity. The two approaches can be compared to two spelling and grammar checking strategies in word processing: the first checks spelling and grammar by running the spelling and grammar checker periodically, whereas the second approach is the equivalent of the option 'check spelling and grammar as you type'.

A third approach is consistency by construction, meaning that a tool generates one deliverable from another and guarantees completeness (to a certain extent). Whenever specifications are defined in one view, those elements in other views that can automatically be inferred are included in those views. Also in this approach, the requirements engineer can only define consistent models. The major advantage is however that the specifications are more or less constructed in an automated way: everything that can automatically be inferred is generated by the CASE-tool. This saves a lot of input effort while at the same time contributing to the completeness of models. In addition, whereas the monitoring approach leads to a CASE-tool that generates error messages at every attempt to enter inconsistent specifications, the self-constructing approach avoids the input of inconsistent specifications by completing the entered specifications with their consistent consequences. The result is a much more user-friendly environment. This type of feature can be compared to an autocomplete functionality.

Text processing	Requirements engineering
Text	Model
Language	Modelling language
Word processing tool	CASE-tool
Checking spelling and grammar	Verifying syntactic quality, i.e. the correct use of modelling techniques
Cross referencing	Verifying consistent naming of model items
Checking the content of the text	Verifying semantic quality (validity and completeness)
Checking the understandability of the text	Verifying pragmatic quality
Check spelling and grammar afterwards	Consistency by analysis
Check spelling and grammar as you type	Consistency by monitoring
Autocomplete	Consistency and completeness by construction

Fig. 2.7 Comparing requirements engineering to text processing

Figure 2.7 compares the different types and approaches of quality control based on the analogy with a word processing tool.

Although CASE-tools can offer substantial help in verifying the quality of models, the human verification remains very important. Moreover, external validity can, for example, only be checked having models checked by domain experts. Although the method presented in this book tries to achieve maximal support by means of algorithmic checking, we nevertheless need to ensure that we can rely on users that are knowledgeable in the business domain, so that they can validate models. Interestingly, the interaction with domain experts will be stimulated by advanced CASE-tool support. When a tool detects an inconsistency, inconsistency resolution will require interrogating the domain experts about what is correct and what not. Referring to the example given in Fig. 2.6, the owner will have to decide which of both plans has the correct representation of the position of the separation wall. Also the autocompletions generated by the tool will have to be verified with domain experts. In this way, the formal verification is a powerful mechanism that substantially contributes to the completeness and external validity of models.

2.2.4 Quality Checking in MERODE and JMERMAID

One of the crucial elements in the development of MERODE is that we want the method to provide effective support for model quality. First, this has been achieved by formalising the domain model by means of process algebra. This book translates all the formal definitions in natural language and illustrates them with many examples. As a result, the user of the methodology does not necessarily need to understand these formal definitions for being able to use MERODE. Notwithstanding, the formal definitions have an added value in that they define concepts more precisely than ever could be done in natural language.

Secondly, to ensure the ease of modelling, a lot of effort has been spent on making all concepts orthogonal. There is a one-to-one mapping between graphical

representation and underlying formal concept and concepts can be composed without restriction: the composition of valid specifications is in its turn a valid specification.

Finally, quite some effort has been put in offering adequate tool support for quality control. This has resulted in the JMermaid tool, a proprietary CASE-tool that specifically supports the creation of MERODE domain models.² The tool implements a mix of the three quality assurance strategies. Whenever possible, the tool implements the ‘consistency by construction’ strategy and autocompletes specifications. This feature can be turned off as well. Next, the input of new specifications is always monitored against the quality rules outlined in this book for each of the modelling techniques. Finally, a final check can be run against a model, to check whether the model satisfies all quality criteria and, if desired, to check whether the model is fit for the generation of Java code.

2.3 Summary

In terms of information system architecture, MERODE advises a layered architecture, organising enterprise information systems (from kernel to outer layer) into an enterprise layer, an information system service layer and a business process layer. This layered architecture should also be applied to functional requirements.

Next to the organisation of requirements into layers, MERODE advises to engineer requirements in a bottom-up way and to focus on creating the domain model first. The benefits of doing so are that this will:

- Foster problem orientation rather than solution orientation and in this way avoid an early implementation bias.
- Contribute to generic thinking and hence more future-proof systems.
- Stimulate the development of a common language across business users, hence contributing to a better business-IT alignment.

On the supply side, the layered information system architecture ensures the adaptability of the software and hence lowers maintenance cost.

The quality checking approach of MERODE and its companion tool JMermaid offer a number of benefits on the demand side of a project:

- The quality checking rules contribute to the ease of modelling as they offer practical guidelines on how to create and improve models.
- The autocomplete functionality of the JMermaid tool but also the quality checking rules in general contribute to the completeness and the external validity of models as each quality check will generate questions stimulating and feeding the discussion between architects and business users.

² JMermaid stands for Java MERODE modelling aid.

- The internal validity of models is ensured, making them fit for implementation and for a model-driven approach to software engineering.
- Thanks to the MDE approach, models can immediately be transformed to a working prototype, which can be used by the architect to validate the models with the business users. In other words, the automatic generation of code makes a MERODE model truly executable.

On the supply side, the quality assurance ensures that the architect delivers complete and consistent models that enable a model-driven approach to software engineering. Transformations can already be developed at the start of the project, in parallel with the requirements engineering phase which facilitates the iterative development of software. Moreover, transformations are reusable across projects and therefore allow to substantially speed up the development of software.

Enterprise Information Systems Engineering

The MERODE Approach

Snoeck, M.

2014, XX, 280 p. 178 illus., 27 illus. in color., Hardcover

ISBN: 978-3-319-10144-6