Chapter 19

EMBEDDING OBJECT-ORIENTED DESIGN IN SYSTEM ENGINEERING

R.J. Wieringa
Department of Computer Science
University of Twente
roelw@cs.utwente.nl

Abstract The Unified Modeling Language (UML) is a collection of techniques intended to document design decisions about software. This contrasts with systems engineering approaches such as for example Statemate and the Yourdon Systems Method (YSM), in which the design of an entire system consisting of software and hardware can be documented. The difference between the system- and the software level is reflected in differences between execution semantics as well as in methodology. In this paper, I show how the UML can be used as a system-level design technique. I give a conceptual framework for engineering design that accommodates the system- as well as the software level and show how techniques from the UML and YSM can be classified within this framework, and how this allows a coherent use of these techniques in a system engineering approach. These ideas are illustrated by a case study in which software for a compact dynamic bus station is designed. Finally, I discuss the consequences of this approach for a semantics of UML constructs that would be appropriate for system-level design.

1. INTRODUCTION

It is an important principle of linguistics that the semantics of a notation cannot be separated from its intended use. This means that if you change the intended methodological use of a design notation, you have to change its semantics and vice versa. It also means that you cannot define a semantics for a notation without making assumptions about its methodological purpose, that should be made explicit. Object-oriented design techniques such as the Unified Modeling Language (UML) are intended to document design decisions about software. This intended methodological use of the UML corresponds to the intended semantics as sketched by the OMG [UML99]. This semantics stays close to the execution semantics of object-oriented programs.

If we want to use UML techniques at a higher level of abstraction, for example the level where we want to model the requirements of an entire system rather than the execution behavior of an object-oriented program, we have to define a different semantics. This higher abstraction level, which is called the essential modeling level, is more appropriate for system-level models, because at this level it has not yet been decided which part of the requirements will be implemented by software, which by hardware and which, perhaps, by people. In this paper I show how some of the UML techniques can be used at the essential model level, alongside with some of the techniques from the Yourdon Systems Method (YSM, [Y93]). The essential model level is the level at which we model purely system requirements and do not take implementation restrictions into account. To use an old phrase of McMenamin & Palmer [MP84], at the essential level, we assume perfect implementation technology.

Section 2. elaborates the relationship between meaning and use of a design notation. To show how UML techniques can be used at the essential modeling level, I give in section 3. an integrated framework for software and systems engineering in the next section. In section 4., I show how UML and YSM techniques fit into this framework and give an example of an essential-level specification where these techniques are used. The major conclusion is that techniques taken from YSM are suitable for the specification of external functionality and that techniques from the UML are suitable for modeling an essential-level system architecture. In section 5., I draw conclusions about desirable features of the semantics of UML techniques used as essential architecture modeling techniques in an integrated systems and software engineering process. The major conclusion of that section is that the UML semantics as currently proposed by the OMG is appropriate for the implementation level but not for the essential level.

2. MEANING AND USE OF DESIGN NOTATIONS

When the central heating system of a house is designed, a subcontractor makes a drawing which represents the location of the central heater, the radiators, the pipes connecting the heater with the radiators, and other components that make up the central heating system. In the Netherlands, different symbols are used for radiators with one, two and three plates and for pipes through which warm or cold water flows. The symbols used in the diagram have a meaning that is directly related to their use: representing the topology of a central heating system. The diagram is used for communication among stakeholders, including installers, the house owner, electricians and other contractors, who can use it to plan their work, detect conflicts among their various views of the house, negotiate about the design, etc. For example, comparing the central heating view with the electrician's view, it may turn out that a radiator has been planned at a location where an electricity outlet is planned too.

Observe that to describe the meaning of a notation to designers is the same thing as to explain to them how to use it. Use a green fat line to represent a radiator; use a red line to represent a pipe that transports warm water and a blue line for a pipe that transports cold water; etc. A description of the meaning of a design notation is a description of the way how the notation is to be used.

The identification of meaning with use stems from the later Wittgenstein and is connected to the idea of language games [W71, paragraph 41]. It is opposed to the early

Wittgensteinian idea that meaning is a mapping from symbols to their denotation, that can be defined independently from any context of use. The identification of meaning with use of design notations does not entail that these notations have no denotational semantics. But it does entail that any such denotational semantics should be closely connected with methodological instructions about how to use the notation.

The identification of meaning with use does not go so far that, in the definition of the meaning of a design notation, we actually tell people how to design a system. The symbols in the design notation for central heating systems do not come with heuristics that tell the designer how to design a central heating system. They merely allow the designer to represent standard components that can be put together in various ways. However, they do help the designer of the central heating system to order his or her thoughts about the design and to communicate the design to others. And they do help the designer to some extent to make design decisions, for the notation says that a central heating systems consists of a central heater, pipes, and radiators, which have certain properties, such as the number of plates of a radiator. And the syntax of the notation disallows the representation of certain combinations of components that cannot work. Similarly, a UML model says that a software system consists of software objects that have certain properties, such as the messages they can exchange, and the syntax of the UML disallows certain combinations of objects that cannot work.

Two interesting aspects of the use of a design notation that should be defined for any practical design notation are the following:

- The intended practical context of use should be specified, including the purpose for which the notation is to be used, in this context. For example, the intended context of use of a design notation could be the design of a central heating system, and its purpose could be to describe the topology of a central heating system—as opposed to for example its energy efficiency or its esthetics.
- The permissible combinations of the symbols should be specified. For example, a line representing a pipe must must be connected to a symbol representing a radiator to a symbol representing another radiator and/or a central heater.

It is possible to identify structures in a context of use and study the properties of these structures mathematically, independently from the particular usage context. It is also possible to study symbol manipulations and the way they can be interpreted in mathematical structures, as is done in logic. Mathematics and logic tell us how certain notations can be used in contexts that exemplify certain abstractly defined structures. To actually use these notations in practice, these abstract structures have to be mapped to concrete usage contexts; and the abstract meanings must make sense with respect to the particular purpose for which the notation is used.

The definition of a practical design notation should therefore consist of a definition of its syntax and of its intended use, which includes at least a description of the intended context and purpose of use of the notation. In the UML, the intended context is software design, but even within this wide context, the purpose of the notations in this context is not fixed. For example, for every notation in the UML, Booch et al. [BRJ99] list many different possible uses. Since there is no single intended use of the UML notations, there is no single intended semantics of UML notations. Rather, there

are many different possible semantics, that are useful for different purposes within the general context of software design.

In this paper, I investigate how design notations from the Unified Modeling Language [UML99] and the Yourdon Systems Method [Y93] can be used in a systems engineering context, that includes software engineering as a part. The answer that will emerge from our investigation is that we can use notations from YSM to represent system and software *requirements*, and notations from the UML to represent a software *architecture* at an essential level of abstraction. This differs from the way the UML is given a semantics in, for example, the Rhapsody Case tool [i-L99, HG97]. But this is just to say that these different semantics are useful for different software design purposes.

3. SYSTEMS ENGINEERING

To define the usage context of the YSM and UML notation aimed at in this paper, we start from the traditional concept of a system as a set of interrelated components working together toward some common objective [BF90]. This very general definition applies to any kind of system, including software systems. It contains two elements.

- First, not any set of elements make up a system: They must work together in a coherent whole. In other words, they must fit together in an architecture.
- Second, the elements of the architecture must work together towards a common objective. In other words, the system architecture must match the system requirements.

Two concerns in any system engineering effort are therefore the design of the architecture of the system and the identification of the objectives to be served by the system. Hence, the two main dimensions of our framework are architecture and requirements. I discuss these in the next two subsections.

3.1 ARCHITECTURE

Systems can be decomposed into subsystems, which can be further decomposed, etc. This gives us a hierarchical architecture of systems. For software systems, this simple picture is incorrect because there are *two* different kinds of decomposition that are relevant, the conceptual and the physical decompositions. Software is a *state* of a physical system. We can decompose the physical system into subsystems, such as PC's and a connecting network, and partition these subsystems further in lower-level subsystems, such as printed circuit boards, plugs, wires, disks, screens, etc. But in a completely different decomposition hierarchy, we can decompose a software system into subsystems, such as a user interface subsystem and a database subsystem, but all the way down to the level of bits, these systems are conceptual. They are abstractions of states of hardware systems.

The conceptual (software) hierarchy and the physical system hierarchy are independent from each other. It is not the case that a component in one decomposition is a "part" of a component in the other decomposition. Rather, there is a many-many relationship between these two decompositions, in which a software component may

be allocated to several physical components and a physical component may execute several software components.

This distinction is related to another, commonly recognized distinction in software engineering, namely that between essential architecture and implementation architdcture. An **essential architecture** is a decomposition defined solely in terms of the desired external functionality of the system. An **implementation architecture** is a decomposition motivated in terms of both external functionality and in terms of the underlying implementation platform. Both architectures are *software* decompositions and thus reside in the conceptual part of the system; they are decompositions of abstractions of hardware states. But where an essential architecture remains close to the concerns of the system user, an implementation architecture reflects the concerns of the engineer who must allocate the software to a network of physical resources.

There is nothing new in the concept of an essential architecture. It is called the logical architecture by some. It corresponds to the essential model of McMenamin & Palmer [MP84], to the specification model of Syntropy [CD94], and to the analysis model of Objectory [JCJO92] and the unified software development process [JBR99]. Typical techniques used to represent an essential architecture are the data flow diagram in structured analysis and the class diagram in object-oriented analysis. Since it does not take implementation limitations into account, it assumes perfect technology. This simplifies the design, improves our understanding of the requirements and facilitates the traceability from external requirements to lower-level implementation parts. The essential architecture should remain invariant under changes in implementation.

To clarify the distinction between the essential and implementation levels, it helps to give some decomposition guidelines for each of these two levels. Examples of essential architectural design guidelines are the following.

- *Functional decomposition*. Identify essential components that correspond to external functions. This is the major decomposition guideline in structured analysis and in systems engineering.
- *Event partitioning*. Identify essential components that correspond to events to which the system must respond. This is also one of the major guidelines in structured analysis [MP84].
- *Interface partitioning*. Identify components that correspond to entities at the interface of the system, such as for example devices with which the system must communicate, users of the system, other software with which the system must communicate, etc. This guideline is used in structured analysis [Y93, page 515] and in object-oriented analysis [JBR99, page 183].
- *Subject domain-oriented decomposition*. Identify components that correspond to subject domain entities, which are entities in the environment of the system referred to by messages that cross the system interface. (The concept of a subject domain is eleborated upon below.) This is the major guideline in object-oriented design.

Despite what some people may say, these criteria are very well compatible. For example, JSD uses subject domain-oriented decomposition to identify the stable part

of a decomposition, and functional decomposition to identify functional components whose meaning is defined in terms of the subject domain-oriented part [J83]. Objectory [JCJO92] as well as the Unified Software Development Process [JBR99] identify entity objects, boundary objects and control objects, that can be found by subject domain-oriented decomposition, interface partitioning and functional decomposition, respectively.

At the essential level, no implementation errors occur, each component has unlimited processing power and memory space available, etc. At the implementation level, by contrast, software must be decomposed into parts that can be allocated to processor nodes in a network of physical resources. Each of these parts must be a sequential process and may be run in parallel to other sequential processes on the processor. They must deal with the finite capacity of the processor and with errors that may occur. Examples of guidelines for implementation architecture are the following.

- *Event-based.* Define one process for each event that must be responded to.
- *Actor-based.* Slightly less greedy in the number of processes that must be run in parallel on one processor is to define one process for each device that can generate events.
- *Time-based.* Still less greedy is to define one process for each group of devices that have similar temporal characteristics. For example if several input devices have to be polled within the same period, these can be dealt with by the same periodic process.
- *Geography-based.* Processes that must communicate with devices in a network must be allocated to a node close to those devices.
- *Purpose-based.* Tasks with a related purpose can be allocated to one process. For example, one process can deal with alarm-handling, another one with error-handling, etc.

More guidelines are given by Awad et al. [AKZ96], Cook & Daniels [CD94], Gomaa [G93] and Douglas [D98]. Some of these guidelines coincide with essential architectural design guidelines, others are different. This explains why the relationship between the essential and implementation architectures is many-many.

A final observation about architecture to make is that the parts into which a software system are decomposed are *processes* in structured analysis and *objects* in object-oriented analysis. Both are entities that have an interface and a behavior. But processes may be instantaneous data transformations, or control processes specified by a state machine, or reactive processes specified by an event list. Objects, by contrast, are uniformly modeled as state machines with local variables (called instance variables or attributes). An object may be specified by a trivial state transition diagram, with only one node. But even in this case, there are usually many different possible attribute values, and therefroe many different possible states. This difference in the kind of entities into which a software system is decomposed—processes or objects—is unrelated to the use of decomposition guidelines. All guidelines listed above, from functional decomposition to subject domain-oriented decomposition, can be used to

yield architectures consisting of processes as well as to architectures consisting of objects.

3.2 REQUIREMENTS

The second main dimension of our framework is that of system requirements. I will not attempt a classification of kinds of system and software requirements; these may range from financial requirements to ergonomic requirements, performance requirements and functional requirements. However, one distinction is important to point out: the difference between business requirements and system requirements. The system under design is embedded in an environment, that I will call the *business environment* or simply the "business". This is the part of the world where the system must have its desired effects (called the problem domain by Jackson [J95]). It must be distinguished from the *implementation environment* of the system, which consists of the lower-level hardware and software components from which the system will be constructed or assembled.

Business requirements are desired properties of the business. They consist of the wishes and goals of people in the environment. They exist in the business and they are about the business, e.g. about a desired way of doing business or a desired way of behaving of people in the business. Business requirements are not about the system but about the business. Techniques to discover them include stakeholder analysis, business strategy analysis and task analysis.

System requirements are desired properties of the system under design. They are motivated by business requirements. They are derived from the business requirements by answering the following question: If these are the goals of the business, what can the system do to help people achieve these goals? An elevator system can help people to achieve the goal of easy transport across floors; an information system can help people to maintain organizational memory; a groupware system can help people to achieve asynchronous communication across time and space. This paper discusses system requirement specification techniques, not business requirement specification techniques.

An exhaustive survey of structured and object-oriented analysis methods [W98b] reveals that the system requirement specification techniques offered by these methods can be classified into three groups: techniques to specify the messages that pass the software boundary, techniques to specify behavioral properties of these messages and techniques to specify communication properties of these messages.

- *Messages.* Because software manipulates symbols, a software system communicates with its environment exclusively by exchanging symbols, or in other words messages, with its environment. I define the **subject domain** of a message as the part of the world referred to by the message, and the subject domain of a software system as the sum of the subject domains of all its messages. The subject domain is part of the business environment of the system. Techniques to specify messages that cross the software boundary include event–response lists (structured analysis), where each event and each response is a message, and use cases to specify message sequences that are of value to an external actor (UML).

- *Behavior* is the ordering of messages in time. Techniques offered for behavior specification include a large variety of state transition diagrams such as Moore (used by Shlaer & Mellor [SM92]), Mealy (used in YSM [Y93]), statecharts (used in Statemate [HP98] and the UML) and SDL state machines [BHS91] as well as a large variety of temporal and real-time logics.
- *Communication* is the ordering of messages over "space". Less metaphorically, it consists of the way in which the system is connected to actors in its environment and interacts with those actors. Techniques to specify communication include the context diagram and data flow diagram of structured analysis, the object communication model of Shlaer & Mellor [SM92], SDL block diagrams [BHS91] and various process algebras. The sequence and collaboration diagrams of the UML can be used to represent communication sequences, but I will argue below that, in the way they are currently defined, they are techniques for illustration rather than specification.

3.3 THE MAGIC SQUARE AND TRACEABILITY

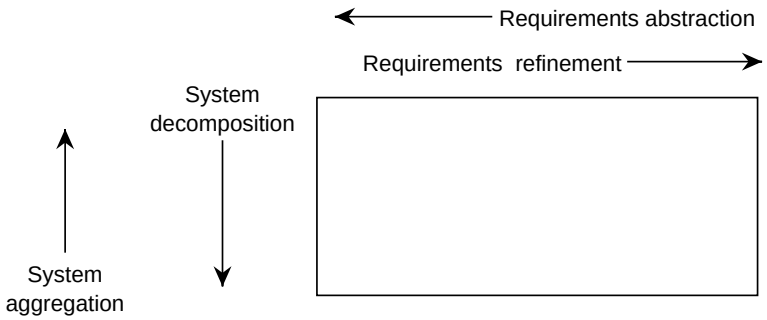

Figure 19.1 Harel & Pnueli's "magic square".

To sum up, our framework distinguishes two requirements and two architecture levels.

- **Business requirements** consist of desired properties of the business environment, such as a desired way of doing business, of working, of behaving, of playing.
- **System requirements** consist of desired properties of the system. We distinguish the desired *messages* between the system and its environment, its desired *behavior*, and its desired *communication* connections to its environment.
- The **essential architecture** is an implementation-independent architecture that realizes the system requirements.
- The **implementation architecture** is an architecture of implementation components, that maps the essential architecture to an implementation environment.

	Requirement 1	...	...	Requirement n
Component 1	X			X
...				
...	X			
Component m			X	

Figure 19.2 Traceability table.

The relationship between the system requirements and the different architecture levels can be represented by the rectangle shown in figure 19.1, called the “magic square” by Harel & Pnueli [HP85]. The upper left point in the square represents the system mission at the highest level of abstraction, and the upper right point represents the system requirements at the highest level of refinement. These include the messages that cross the system boundary and the behavioral and communication properties of these messages. In between these two extremes, we can find, for example, the required system functions and use cases. At the lower left point, we find the purpose of each low-level component of the system and at the lower right point, we find the detailed messages and protocols by which these components communicate with their environment. System engineering is a journey through this square that generally begins at the upper left point and ends at the lower right point. This journey may proceed in a top-down, bottom-up, inside-out or iterative manner. Discussion of engineering strategies is outside the scope of this paper.

Each point in the square identifies one level of aggregation and one level of requirements refinement. For each point, a **traceability table** such as outlined in figure 19.2 exists. This relates the requirements at one level of refinement to the parts of the system at one level of aggregation. For example, a traceability table can be used to show how external system functions are allocated to essential software objects. Each column would then show how a number of software objects collaborate to realize an external function.

Note that we can use traceability tables between any two different levels of abstraction: For example, to trace business requirements to system requirements, system requirements to essential architecture components, and essential architecture components to implementation components. Traceability tables are an important technique in systems engineering and they are a central technique in quality function deployment [L95]. For any non-trivial system, its traceability tables are too large to draw and we need software tools to store, manage and present this information. In section 5. I show how the UML techniques can be mapped to a traceability table that maps software requirements to software architecture.

Technique	Source	View	Abstraction level
Mission statement	YSM	External behavior	System requirements
Function refinement	IE		
Context/use case diagram	YSM / UML	External communication	
Event−response list	YSM	External messages	
Subject domain entity model	YSM	Subject domain decomposition	Subject domain architecture
Class diagram	UML	Essential decomposition	Essential architecture
Communication diagram	Shlaer−Mellor, SDL	Essential component communication	
Statecharts	UML	Essential component behavior	
Component diagram	UML	Implementation components (software)	Implementation architecture
Deployment diagram	UML	Implementation components (physical)	
Dictionary			

Table 19.1 Techniques in TRADE. YSM = Yourdon Systems Method [Y93], IE = Information Engineering [M89], UML = Unified Modeling Language [OMG97].

4. A TOOLBOX FOR REQUIREMENTS AND DESIGN ENGINEERING (TRADE)

In this section I show how some techniques from YSM can be combined with some of the UML techniques in a coherent systems engineering approach. Table 19.1 lists the techniques to be used. I call this list a Toolbox for Requirements and Design Engineering, or TRADE for short. More information about the source of the techniques in TRADE is given by Wieringa [W98a]. In section 5., I discuss requirements on the precise syntax and semantics of the techniques listed in table 19.1. Here, I informally illustrate most of these techniques by means of a case study in which software for a compact dynamic bus station is specified. Due to space limitations, component and deployment diagrams are omitted. The list of specifications given below does not represent the temporal sequence in which these specifications have been produced. It should not even be taken to suggest that they have been produced in any sequence: They have, in fact, been produced in parallel.

A bus station consists of a number of platforms at which busses can park so that passengers can enter and leave busses. At any point in time, most platforms of a bus

station are empty, which in densely populated areas is a waste of space. A *compact* dynamic bus station consists of a small number of platforms to which busses can be dynamically allocated so that, over time, the available space is used optimally. The software system to be designed is informed of the approach of all busses and has a database of preferred allocations of busses to platforms. It dynamically allocates an approaching bus to a platform that is expected to be available at the time the bus will enter the station, and informs the driver as well as waiting passengers about this allocation. The driver is then expected to drive to this platform and the passengers to walk to this platform to enter the bus. If there is unexpectedly no room at the platform for the bus, the bus will drive to a buffer area, from which it will drive to the platform as soon as there is room available. More requirements are given in the specification below. I will refer to the software system as the Bus Allocation System, or BAS for short. As part of a feasibility study in the application of formal description techniques, a specification of BAS was made in ASF/PSF, from which a prototype was generated [KVS96]. In order to make the structure of the ASF/PSF specification more explicit, we developed the model presented in this paper.

4.1 BUSINESS REQUIREMENTS

Even though table 19.1 does not contain techniques for the specification of business requirements, it is useful to illustrate what these requirements look like in this example. There are several stakeholders that play a role with respect to the system, including the bus drivers, the passengers, the bus company and the public in general, as represented by the municipality that maintains the roads. An analysis of their goals and needs would uncover such business requirements as the following:

- Passengers want to be informed in a timely manner about the approach of busses and the platform at which they can enter a bus.
- Drivers want to be informed in a timely manner about the platform they must drive to.
- The bus company wants efficient use of scarce platform space.
- The municipality wants to reduce the use of private cars and increase the use of public transport.

Requirements like these set the stage for design choices made in determining system requirements. The crucial step from business to system requirements is made by answering the following question: If this is what the business wants, then how can the system help the business achieve these goals? We now turn to the techniques in TRADE for specifying system requirements.

4.2 SYSTEM REQUIREMENTS

I simply work down the list of techniques shown in table 19.1. I should reiterate that this is *not* the sequence in which the specifications have been developed. They have been developed in parallel.

The mission of BAS is to dynamically allocate busses that approach a compact dynamic bus station to platforms. Its responsibilities are:

- To monitor the presence (arrival and departure) of busses at platforms;
- To respond to the approach of a bus by recomputing the optimal allocation of all busses to platforms;
- To announce the allocation to the driver in the approaching bus and to passengers waiting at platforms.

Figure 19.3 Mission statement for BAS.

Mission statement. A mission statement describes the required functionality of the system at the highest level of abstraction. It gives the reason for existence of the system, and links it to the business requirements. It summarizes the reason why some stakeholders in the business environment would pay money for acquiring the system. It consists of a one-sentence description of the functionality followed by a description of the main responsibilities of the system. Figure 19.3 gives a mission statement for BAS. Often, it is useful to additionally state a number of things that the system will *not* do. Although there are infinitely many things that the system will not do, the expectations of stakeholders may have to be managed by listing a few of those things explicitly.

Function refinement. A **system function** is a piece of external interaction of the system that is useful to some actor in the business environment of the system. Any execution of a function consists of a sequence of message exchanges with the business environment. In the UML, system functions are called **use cases.** Jacobson tries to distinguish use cases from functions by saying that functions are system interactions that might be good to have, whereas use cases are system interactions that are useful for particular classes of actors [JBR99, pages 5, 37]. This is a caricature of the definition of "function" in structured analysis, put up merely to maintain a fictional distinction between the use case approach and the functional approach to system requirements. Both structured analysis and object-oriented analysis start from desired system functions, both define them as external interactions useful for external actors, and both represent these actors in a diagram (the context and use case diagrams).

A function refinement tree is a hierarchic list of desired system functions, in which the system functions may be refined one or more times. The meaning of this refinement is that the functions $f_1, \ldots, f_n$ that refine a function f_0 jointly describe f_0 at a lower level of abstraction (a higher level of detail) and, conversely, that the desirability of f_0 explains the desirability of $f_1, \ldots, f_n$. Both levels of refinement refer to the same system at the same level of aggregation. In terms of Harel & Pnueli's "magic square", function refinement moves from left to right in the square but remains at one level of aggregation. Going to the more detailed level (to the right in the "magic square"), we

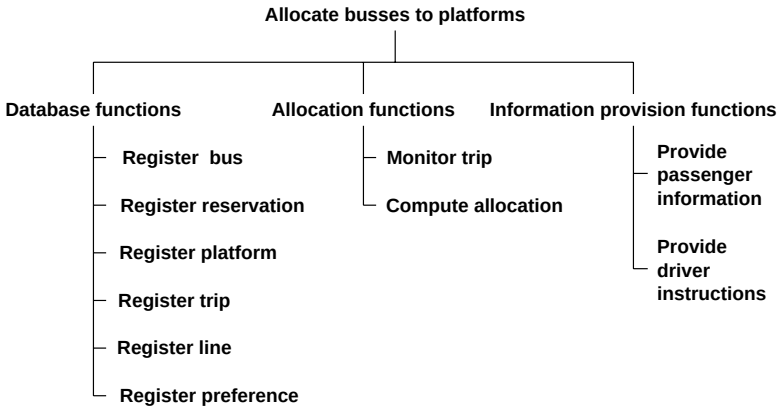

Figure 19.4 Function refinement tree.

answer the question *what* the system does. Going to the more abstract level (to the left in the "magic square"), we answer the question *why* the system does that.

A function refinement tree is a result of negotiation between the customer and the designer. There are many different ways to organize functions in a tree, that depend upon the perception of the functionality by the customer and the designer. What matters is that the function refinement tree bounds the discussion with the customer and acts as a means of communicating with the customer about desired functionality. The tree can be represented simply by an indented list of function names. If the tree is small, however, it may be represented by a tree diagram. Figure 19.4 shows a function refinement tree for BAS.

Context Model. A context model shows the way in which the system is embedded in its environment. The context diagram used in structured analysis shows the possible communications between the system and its external actors. The use case diagram of the UML does the same for use cases: For each use case (system function), it shows the external actors with which the system communicates when it is engaged in that use case. So the use case diagram is actually equivalent to a context diagram for a number of system functions. The UML provides several constructs that can be added to this basic picture, which I will not treat.

A non-trivial system may have dozens or even hundreds of use cases, too many to represent in one diagram. One can manage this complexity by maintaining a relationship with the function refinement tree. The root of the tree represents the system mission; this corresponds to the context diagram, which shows the external communications needed to realize this mission. The leaves of the tree represent elementary system functions, or elementary use cases in object-oriented parlance; a use case diagram provides a context diagram for all of these use cases. If there are too many elementary use cases to depict in one diagram, one can move up the tree until an

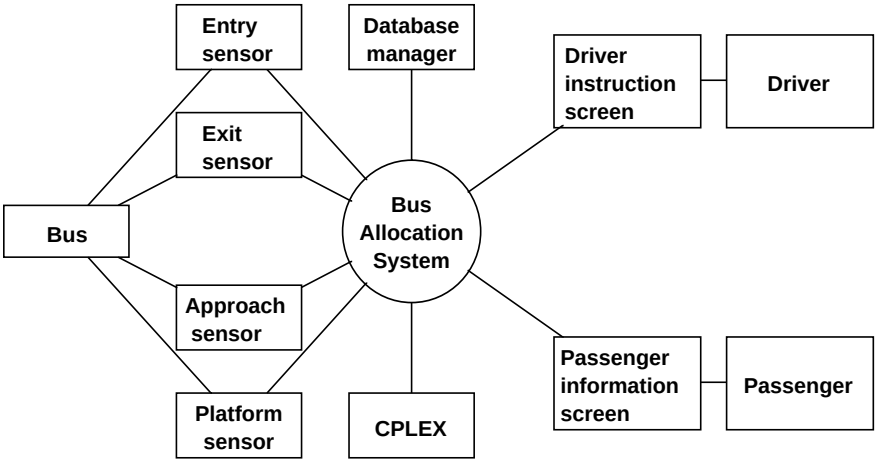

Figure 19.5 Context diagram of BAS.

abstraction level is reached with a manageable number of more abstract use cases, and draw a use case diagram at that level. Or one can just draw a context diagram for each use case separately without drawing a diagram that shows all of them.

Context diagrams and use case diagrams only show the actors with which the system communicates and omit the wider context of the system. As argued by Jackson [J95, page 35], it is actually very informative to include the wider context. The immediate context often consists of sensors and actuators, whereas the actors of interest, for whom the system performs a service, are one or two communication links removed from the system. Figure 19.5 gives a Yourdon-style context diagram for BAS, representing the system under design by a circle, external actors by rectangle and communication links by lines. CPLEX is a linear programming software library that performs the computations needed to optimize the bus allocation.

Event–response list. Figure 19.6 lists the events to which the system must respond, the stimulus by which the system discovers that the event has occurred, the source of this stimulus, the desired response of the system, and the function to which this event–response pair contributes. The arrival of an event and the production of a response are input and output messages of the system, respectively. Other properties of event–response pairs that can be listed are the destination of the response message, performance requirements, safety properties, ergonomic requirements, etc. Cross–referencing the list to the function refinement tree acts as a coherence check: Each event–response pair should contribute to a system function, and each system function should be realized by one or more event–response pairs. The event list is also cross-referenced to the context model, because the sources of stimuli in the event list as well as the destination of the responses, must be actors in the context model that have direct communication links with the system.

Event	Stimulus	Source	Desired response	Function
Scheduled arrival 15 minutes from now		Time	Update trip status to Expected.	Monitor trip
Scheduled arrival time has arrived		Time	Trip status becomes delayed. Update passenger information board.	Monitor trip Provide passenger information
Bus approaches	approach	Approach sensor	Update passenger information board. Update trip statusto Approaching.	Monitor trip Provide passenger information
Bus enters bus station	entry	Entry sensor	Update passenger information board. Instruct driver to drive to platform or buffer.	Monitor trip Provide passenger information Provide driver instructions
Bus arrives at platform	at platform	Platform sensor	Update trip status. Clear driver instructions.	Monitor trip Provide driver instructions
Bus departs from bus station	exit	Exit sensor	Clear passenger information board. Recompute position allocation.	Monitor trip Provide passenger information
Time to compute allocation (Every minute)		Time	Compute optimal allocation.	Compute allocation
Update database	update	Database manager	Update the database.	All database functions

Figure 19.6 Events to which the system must react, and their desired responses.

Subject domain. The messages received and sent by the system are messages about the subject domain. Figure 19.7 shows a simplified decomposition of the subject domain into entities and relationships. I use a subset of the UML class diagram syntax to represent entity-relationship diagrams. Boxes represent entity types and lines represent relationships, annotated with multiplicities. Attributes of entities can be listed in the entity type box.

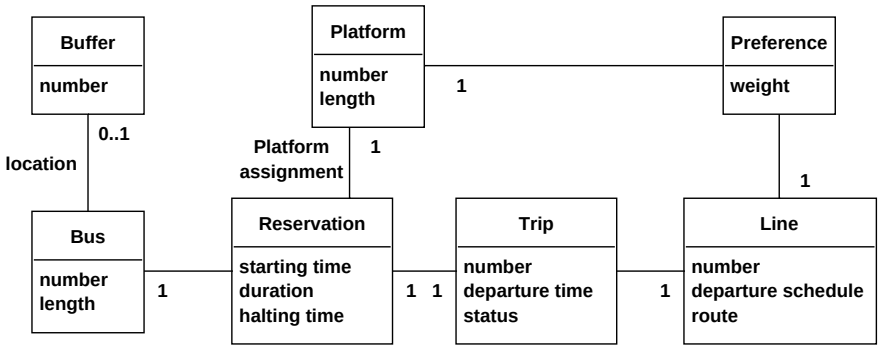

Figure 19.7 Class diagram of subject domain decomposition.

The figure shows that each trip belongs to one bus line and that a reservation relates a bus with a trip and a platform. Each line has a preference for a platform, which is indicated by a certain weight. Finally, a bus may (temporarily) be at the buffer.

Note that a subject domain model is not part of the system model. It is really part of the requirements dictionary, that documents the meaning of terms used in the requirements specification. The subject domain is the part of the business environment that is referred to by the messages that the system exchanges with its environment. A subject domain model represents a decomposition of this part of the business environment. The information about the subject domain represented by this model includes:

- Which classes of entities exist in the subject domain and
- what is the multiplicity of these classes and their relationships.

In addition, static and dynamic properties of the subject domain may be represented if this helps to understand the requirements of the system. The subject domain model must be cross–referenced to the event–response list, because the descriptions of the events and responses in that list can refer only to entities, relationships and attributes defined in the subject domain model.

4.3 ESSENTIAL ARCHITECTURE

Component classes. Figure 19.8 shows an essential decomposition of the system which shows the boundary between the software part of the system and the other parts. The software part duplicates the subject domain decomposition and adds software

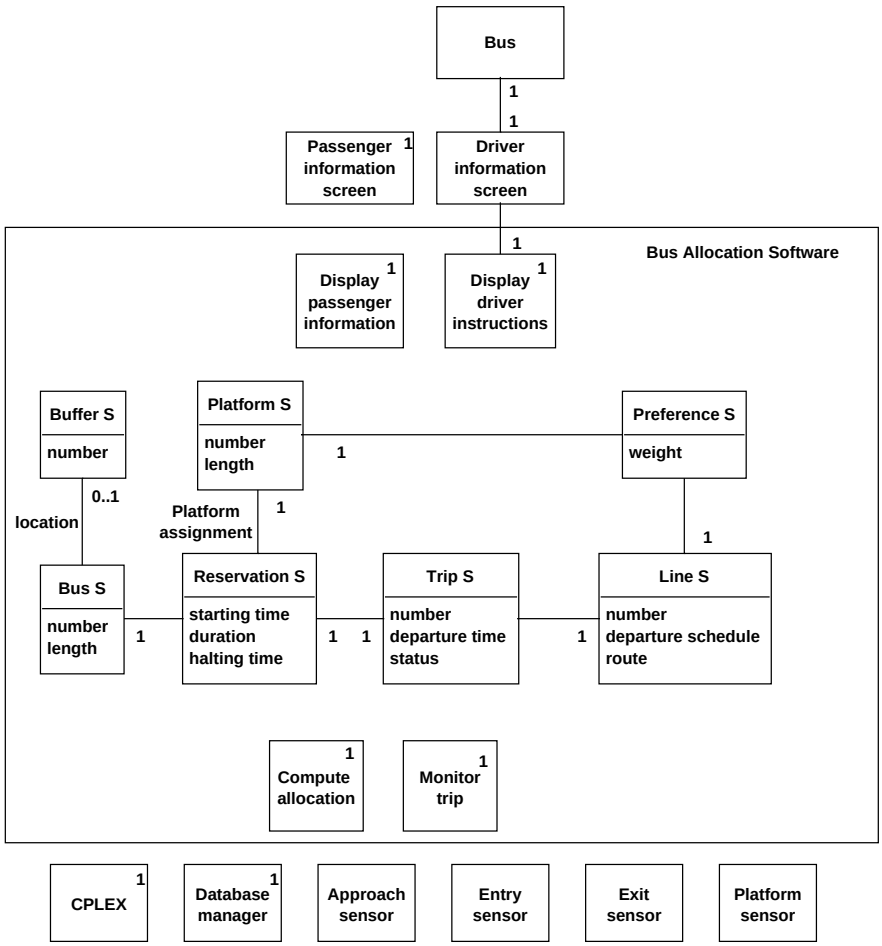

Figure 19.8 Class diagram of essential system decomposition.

objects that handle the desired system functions. The decomposition is represented by a UML class diagram. Two of the essential architectural software design guidelines mentioned in subsection 3.1 have been used: Identify software objects that represent subject domain entities and identify objects that perform required system functions. The class of a software object that represent a subject domain entity has been given the same name as the entity type, suffixed with an S.

The information provided by a class diagram the system architecture is the following:

- Which classes of objects make up the system, and
- what are the multiplicities of these classes and their relationships.

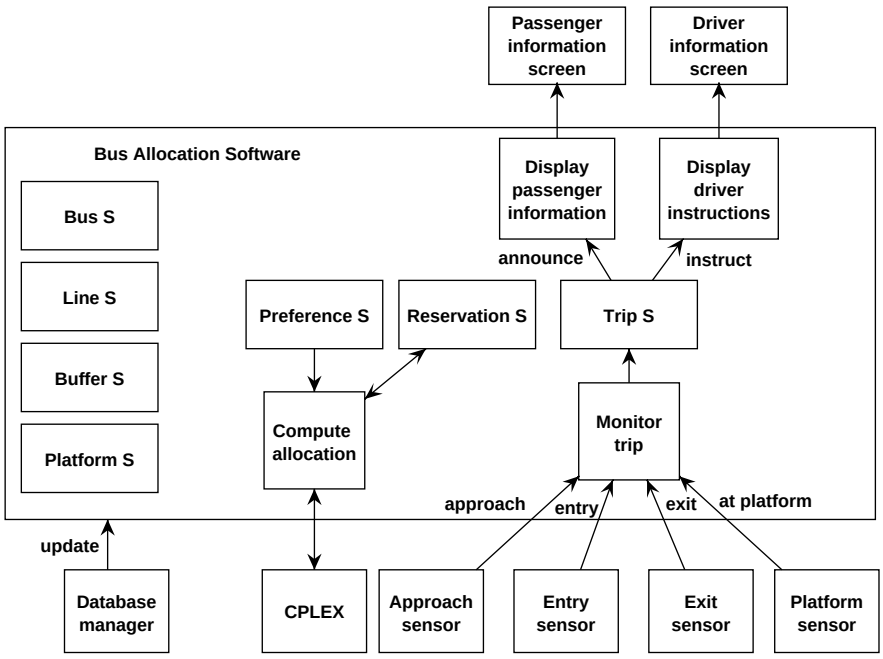

Figure 19.9 Essential communication diagram, showing the communications needed at the essential level.

Figure 19.8 shows that the function classes are singleton classes. The diagram also shows that there are many driver information screens (one per bus), but that there is one passenger information screen. Associations in the class model do not represent communication links but are used to represent multiplicity relationships. They are included only where they are needed to represent multiplicity information.

Depending upon the kind of subject domain and the desired system functionality, other information from the subject domain model can be used to design the software components, such as state invariants and behavioral properties of the software objects.

Component Communication. It is useful to represent communication between essential components by a separate **communication diagram**, consisting of boxes and arrows. The boxes represent object classes just as in a class diagram, possibly including multiplicities, attributes and operations. In contrast to class diagrams, however, all lines in a communication diagram are arrows, and they represent communication links. Figure 19.9 shows the communication diagram for BAS.

Our communication diagram is similar to communication diagrams used by Shlaer & Mellor [SM92], to block diagrams in SDL and to architecture diagrams used in the literature on software architectures [SH96]. It must be cross-referenced to the class diagram, because there, all the boxes from the communication diagram must be declared as object classes. It must also be cross-referenced to the context diagram,

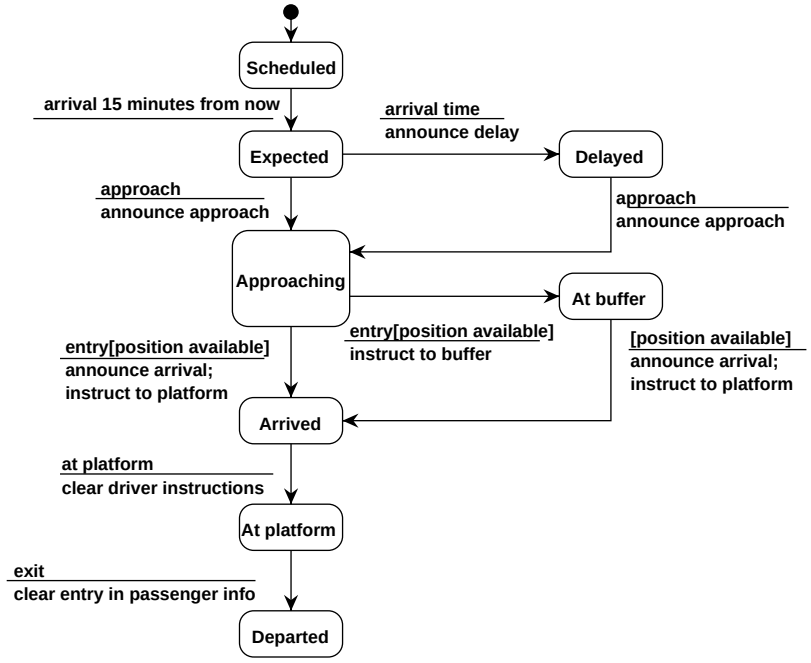

Figure 19.10 Statechart of the status of a trip.

because there, all the external communications of the communication diagram should appear.

Component behavior. Figure 19.10 represents the desired behavior of a trip by a simple and informal statechart. In an executable model, the events, guards and actions of the statechart should be specified in some executable language. The attribute status of a trip object represents the possible states in this diagram. The statechart must be cross-referenced with the communication diagram, because it must describe the behavior of instances of a class in the communication diagram, and the events and actions in the statechart must appear as communications of that class in the communication diagram.

5. USING UML TECHNIQUES IN SYSTEM ENGINEERING

The YSM function specification techniques in TRADE can be mapped in a simple way to Harel & Pnueli's "magic square":

- External system functionality at the highest level of abstraction (the upper left point in the "magic square") is represented by the *mission statement* and at the highest level of refinement (the upper right point of the "magic square") by the

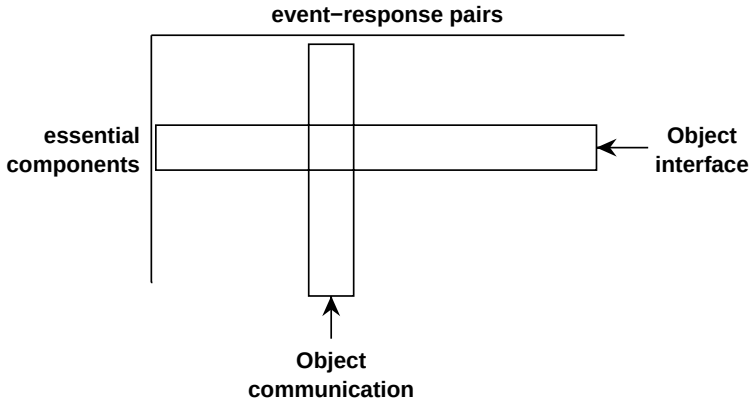

Figure 19.11 Mapping techniques to the traceability table.

event–response list. The two levels of refinement are related by the *function refinement list.*

The UML can be mapped to the traceability table format of figure 19.2; see also figure 19.11.

- The *class diagram* represents the classification and multiplicity of the components that appear in the leftmost column of the table.

- Each row of the table represents the interface of an object, that must be declared in the class diagram. The *statechart* of an object class constrains the behavior of this interface.

Each column of the table represents the operations of different objects that realize an event–response chain. The *communication diagram* shows the communication links needed for these collaborations. However, the communication diagram is not part of the UML. Instead, the UML defines two other kinds of diagrams to represent communication: sequence and collaboration diagrams. In their basic form, both diagrams can be used to represent a sequence of message exchanges between a finite (small) set of entities. These entities may be for example software objects, or processing resources in a network, or external actors exchanging messages with the system. Sequence and collaboration diagrams contain various constructs that allow one to specify control structures such as iteration, and to represent several independent processing sequences in different threads of control. In whatever form, these diagrams represent *possible* execution sequences in which a number of *individual* entities are involved. Their use as a specification technique is restricted to showing particular behaviors that the system must be able to perform. More properly, they can be used to document a particular implementation, for example, by illustrating a behavior pattern. For this reason, they are not included in table 19.1. Damm & Harel [DH99] have defined an extension to

sequence diagrams in which one can express the requirement that all possible behaviors of a system must conform to the diagram.

The *component* and *deployment diagrams* from the UML, not treated in this paper, show the communications between implementation components and physical resources. These should be related to the essential level components by another kind of traceability table. Finally, UML *activity diagrams* can be used to represent a network of activities between which temporal dependencies exist. These seem more suitable to representing workflows in organizations, i.e. business requirements, rather than system requirements. However, the UML is a mulit-purpose notation and Booch et al. [BRJ99] also recommend using activity diagrams for documenting operations of software objects.

The use of UML techniques in TRADE has implications for the semantics of these techniques when they are used this way. Class diagrams and statecharts are used in TRADE to represent software architecture at the essential level, where we can assume perfect technology. This means, for example, that actions do not take time and that all objects perform their tasks in parallel. A first version of such an essential-level semantics is presented elsewhere [WB98, BW99]. This semantics differs from the OMG semantics [UML99], in which actions take time, there are several threads of control and one message queue per thread which can receive signals exchanged by objects. The OMG semantics is clearly intended for and appropiate to what I call the implementation architecture. This is called the design model in the Unified Software Development Process [JBR99]. The use of C++ as action language in the executable UML models of Rhapsody [i-L99] confirms this, as does the outline of the executable statechart semantics given by Harel & Gery [HG97]. As pointed out in section 2., I do not claim that one of these semantics is "better" than the other. Rather, I claim that they are good for different methodological purposes.

If we describe an intended semantics of a design notation without making its intended methodological purpose explicit, as is done in the OMG documentation, then we leave the user of the notation in the dark about the use of the constructs whose meaning has been defined. Conversely, if we omit a description of the semantics of a notation from a description of the method for which this semantics is intended, as Jacobson et al. [JBR99] do, then we leave the reader in the dark about the meaning of the constructs she uses. To repeat Wittgenstein once more, meaning is use.

As pointed out before, the UML user guide by Booch et al. [BRJ99] lists so many possible completely different uses for each of the UML diagrams, that there is no single semantics that corresponds to all of these uses. This places the UML semantics proposed by the OMG into perspective as one among a variety of possible semantics, suitable to different methodological purposes. The OMG semantics is suitable for documenting an object-oriented program by an executable model that closely reflects the implementation. The TRADE semantics is another possibly useful semantics, suitable for documenting the architecture and behavior of a system required at the essential level, independent from its implementation. This resembles very much the semantics of Statemate models [HP98].

6. CONCLUSIONS AND FURTHER WORK

This paper gave a framework for system and software design that separates requirements from architecture, distinguishes business from system requirements, and essential from implementation architecture. System requirements are separated into required messages, required behavior and required communication. TRADE is a collection of techniques to specify system requirements and essential software architecture, that includes techniques from YSM for requirements specification and techniques from the UML for essential architecture specification. This entails an essential-level semantics for the used UML constructs, that differs from the intended OMG semantics. It was shown how this collection of techniques can be used in a systems engineering approach, that emphasizes the embedding of software in its system environment and of the system in its business environment and that explicitly maintains traceability between these different levels.

Our current work includes the elaboration of the formal semantics for a UML class and behavior diagrams that is appropriate for the essential modeling level, and application of the UML to a number of case studies to validate this semantics. After validation, the execution semantics will be implemented in TCM [DW96].

References

[AKZ96] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice-Hall, 1996.

[BF90] B. S. Blanchard and W. J. Fabrycky. *Systems Engineering and Analysis*. Prentice-Hall, 1990.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from protocol Specification*. Prentice-Hall, 1991.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BW99] J. Broersen and R.J. Wieringa. A logic for the specification of multi-object systems. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Formal methods for Open Object-Based Distributed Systems*, pages 387–398. Kluwer, 1999.

[CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[DH99] W. Damm and D. Harel. LSC's: Breathing life into message sequence charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Formal methods for Open Object-Based Distributed Systems*, pages 293–311. Kluwer, 1999.

[D98] B.P. Douglas. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[DW96] F. Dehne and R.J. Wieringa. Toolkit for Conceptual Modeling (TCM): User's Guide. Technical Report IR-401, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1996. http://www.cs.utwente.nl/~tcm.

[G93] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley, 1993.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985. NATO ASI Series.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[i-L99] i-Logix. *Rhapsody Reference Guide, Version 2.1*. i-Logix Inc., 1999.

[J83] M. Jackson. *System Development*. Prentice-Hall, 1983.

[J95] M. Jackson. *Software Requirements and Specifications: A lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JCJO92] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Prentice-Hall, 1992.

[KVS96] A.S. Klusener, S.F.M. van Vlijmen, and A. Schrijver. Compact dynamisch busstation. Technical Report CS-N9601, Centrum for Wiskunde en Informatica, May 1996.

[L95] W.M. Lamia. Integrating QFD with object-oriented software design methodologies. In *The Seventh Symposium on Quality Function Deployment*, pages 417–433. QFD Institute, 1995. http://www.gfdi.org/.

[M89] J. Martin. *Information Engineering*. Prentice-Hall, 1989. Three volumes.

[MP84] S. M. McMenamin and J. F. Palmer. *Essential Systems Analysis*. Yourdon Press/Prentice Hall, 1984.

[OMG97] OMG. *Unified Modeling Language: Notation Guide, Version 1.1*. Object Management Group, 1 September 1997. http://www.omg.com.

[SH96] M. Shaw and D. Harlan. *Software Architecture*. Prentice-Hall, 1996.

[SM92] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, 1992.

[UML99] UML Revision Task Force. *OMG UML Specification*. Object Management Group, march 1999. http://uml.shl.com.

[WB98] R.J. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class– and behavior diagrams. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *ICSE98 Workshop on Precise Semantics for Software Modeling techniques*, pages 129–151, 1998. Report TUM-I9803, Technische Universität München.

[W98a] R.J. Wieringa. Postmodern software design with NYAM: Not yet another method. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering*, pages 69–94. Springer, 1998. Lecture Notes in Computer Science 1526.

[W98b] R.J. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, December 1998.

[W71] L. Wittgenstein. *Philosophische Untersuchungen.* Suhrkamp Verlag, 1971.

[Y93] Yourdon Inc. *Yourdon™ Systems Method: Model-Driven Systems Development*. Prentice-Hall, 1993.

7. ABOUT THE AUTHOR

Roel Wieringa is professor of Information Systems at the University of Twente, the Netherlands. His research interests include design methods for software systems that integrate techniques for data-intensive and behavior-intensive systems, and the integration of formal description techniques with diagram-based techniques. In this context, he has studied the application of algebraic and logical techniques to information system specification. His current research interests are the application of temporal and real-time logics in information system specification. Together with J.-J. Ch. Meyer he edited a book on *Deontic Logic in Computer Science*. He wrote a book on *Requirements Engineering: Frameworks for Understanding* (Wiley, 1996), in which a number of classical software engineering methods are analyzed in a systems engineering framework.