Traceability and Modularity in Software Design

Roel Wieringa*
Department of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede, the Netherlands

Abstract

A software design specification consists of a number of documents that describe various aspect of the design at different levels of detail, that are linked in many ways. This paper shows how different designs may use different modularization criteria, and how documents describing these designs may be linked in a coherent way, even if the designs use techniques borrowed from structured as well as object-oriented analysis and design. Illustrations are taken from the meeting scheduler case study.

1 Introduction

In this paper, the term "design" is used in its most general sense of a decision that reduces uncertainty about a useful future artefact. In this sense, drawing up a list of desired external system requirements is a design activity just as determining a collection of internal components is. A design process delivers one or more documents that contain specifications of various aspects of the design, structured and linked in various ways. The documents are called **traceable** if it is easy to find related parts of the same or different documents [2, pages 191–192]. Traceability across different documents can be improved by giving these documents the same structure. Related parts in different documents are then given the same place in the structure. Another way to improve traceability is to visibly store links between related parts the documents (e.g. by cross-references). If this is done, these documents are called **traced**. A document that is traced is traceable, but the reverse is not true, because tracing is only one way of realizing traceability.

It is commonly stated in object-oriented methodology that object-orientation allows a seamless transition from analysis to architectural design and implementation, which enhances traceability of the design documents. Contrary to this claim, I argue in this paper that the structuring principles at different levels of software design, object-oriented or otherwise, lead to different design structures and hence to different document structures, that are harder to trace. In order to make these documents traceable, explicit links between related parts must be defined. Consequently, the aim of this paper is twofold: One, to show that the structuring principles at different levels of software design are essentially different, even in an object-oriented design, and two, to outline a set of links across different software design levels that can be used to improve traceability.

This is done by presenting a set of design levels and specification techniques for those levels that are borrowed from structured as well as object-oriented analysis and design. Putting on another pair of spectacles, this paper can therefore be viewed as an attempt to integrate structured and object-oriented software design. An important reason to attempt such an integration is that major defects can arise if software design is not properly embedded into systems design [6, 12]. Since system engineers routinely think in terms of functional decomposition [16], we must be able to trace software designs back to functional designs at the system level.

The presentation of the integrated structured/object-oriented design approach in this paper focusses on the issues of modularization (structuring a design) at different design levels and the definition of links across these levels. There are five or more design levels distinguished in this paper: requirements, external properties, conceptual decomposition, one or more implementation decompositions. These have been identified after a thorough analysis of structured and OO methods within a systems engineering framework [17]. In sections 2 to 4 the requirements to conceptual design levels are discussed in more detail. Due to lack of space, we ignore the mapping of a conceptual on an implementation decomposition. The specification links identified in this paper are identified by numbers between brackets. We only *mention* the links here. Precise definition by means of a metamodel is a topic

*Email: roelw@cs.utwente.nl. Work performed when at the *Vrije Universiteit*, Amsterdam.

Business objective: to schedule meetings.
Desired way of working:

R1 Determine a meeting date.

...

R1.7 Initiator requests excluded and preferred dates.

R1.8 Participant provides excluded and preferred dates.

R2 ...

Figure 1: Fragment of a requirements specification.

of current research [3]. Fragments of a specification of the Meeting Scheduler System (MSS) are used as illustrations. A full specification is available from ftp [18].

2 Requirements

I refer to the software system being designed as the System under Development (SuD). When a SuD is designed, there are always other systems being designed too. In general, software is embedded in hardware to form a larger system S, which in turn is embedded in a social system B, which I refer to as the business. The business has objectives, and people in B work together (so one hopes) to achieve these objectives. In the future situation, the people in B will interact with S and possibly other systems to achieve their business objectives. For example, the objective of the social environment B of the meeting scheduler system is to schedule meetings. The MSS is embedded in hardware to form a system S such that people in B can interact with S to schedule meetings with less overhead than currently exists.

I define a **requirements specification** as a description of the relevant business objective(s) and of the desired way of working to reach these objectives. A requirements specification must not refer at all to the SuD but only to the desired way of working and the business objectives: It only says *what* work is done in B and *why* this is done. Each requirement must be linked (1) to a business objective that motivates it.[1]

The structure of a requirements specification is determined largely by the structure of the desired way of working in the business. It may range from a simple hierarchical itemized list of tasks, as in figure 1, to an elaborate workflow model of a business. The requirements may be partitioned according to the relevant *business actors*, or *business departments*, or *workflows*, or *work procedures*, etc. See table 1 for a summary of modularization criteria. Whatever the modularization of requirements, there will be links (2) between requirements, that must be documented. In addition, we should add links (3) to various kinds of sources for the requirements [7]. (These are not shown in table 2.)

[1] The numbers refer to entries in table 2.

3 External properties

The SuD exists in order to allow its users to realize their requirements. An **external property specification** is a description of the desired externally observable properties of the SuD that should help users realize their requirements. These properties always consist of the ability of the SuD to engage in certain external interactions that have certain properties. To represent the system boundary, a **context diagram** can be drawn that represents the interactions between the system and its external actors. This is a classical structured analysis tool [5], recently reinstated by Jackson [10]. External interactions of a software system have a special property, discussed next.

3.1 Subject domain and system dictionary

The external interactions of a software system consist of the creation and deletion of symbol occurrences at the interface with the system. Each symbol occurrence is a physical item to which we assign a meaning. The part of the world referred to by these symbols is called the **subject domain** of the external interactions of the SuD. (Another term often used is Universe of Discourse.) To specify the desired interactions of the SuD, we must agree with the user about the meaning of these symbols. This is done by making a **subject domain model** and writing a **dictionary** that defines the meaning of the symbols by which the system communicates with its environment. The meaning of the symbols must be defined in terms of this subject domain model. A subject domain model always models the subject domain by representing a decomposition of it. We use class diagrams to represent this decomposition (figure 2). Other properties of the subject domain must be specified textually.

Note that different sets of external interactions may have different subject domains. Much of the design

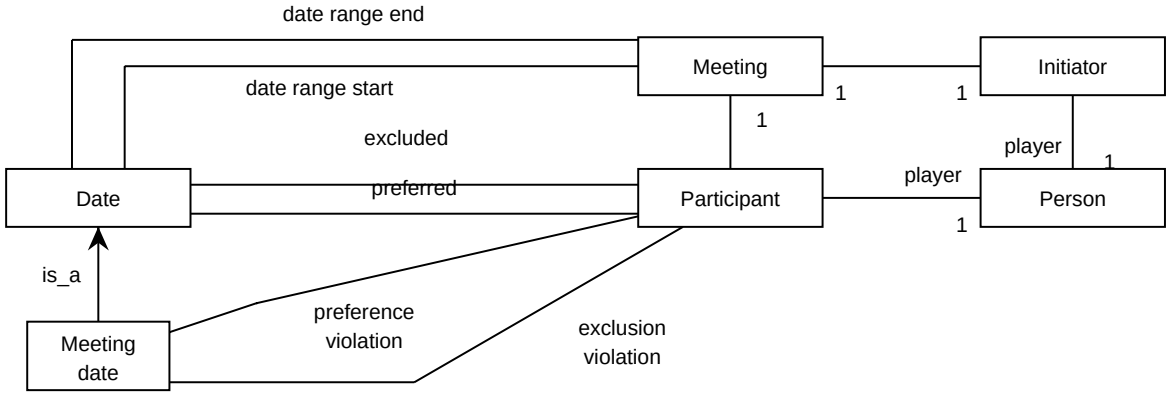

Figure 2: Subject domain model.

- **Excluded date**: Each *participant* may have a list of *dates* on which the participant cannot attend the *meeting*. Each *participant-date* pair identifies at most one excluded date for a particular *meeting*.
- **Exclusion set**: The set of *excluded dates* of a *participant*.

Figure 3: Dictionary entries. Terms in italics are defined elsewhere in the dictionary.

activity therefore consists of merging different subject domain views and negotiating a single dictionary across different views.

The system dictionary must define all class names (4) and association names (5) that occur in the subject domain model in clear language understandable by the user (the subject domain expert). Other relevant definitions, e.g. of some attributes (6) or operations (7), may also be entered. Figure 3 shows two dictionary entries. Writing the dictionary forces us to go back to the requirements specification and bring its terminology in agreement with the system dictionary (8), because the requirements usually use terms that refer to the subject domain.

The structure of the subject domain model is dependent upon the subject domain alone and is independent of the SuD. The structure of the dictionary is a web: like all dictionaries, the entries have many cross-references (9).

3.2 System functions

The **mission** of the SuD is the most general service that the SuD provides to its environment. It is the reason why the SuD should exist. Any external property specification should contain a specification of the mission of the SuD and relate this to business objectives (10). In addition, it may include a specification of the major responsibilities of the SuD, and a list of things that the SuD will not do [19, page 159], both related (11) to the mission (figure 5).

The rest of the property specification can be structured in a variety of ways (figure 1). It can be structured according to the external actors with which the SuD interacts, or according to the work procedures to be followed by these actors, or according to the functions offered, or according to the features supported, or to the kind of triggering event, to the kind of response desired, etc. [2, page 196]. This list only partly overlaps with the criteria for requirements structuring given earlier. Whatever the criteria used, each part of the property specification must be linked (12) to one or more requirements, and all parts must be linked (13) to the system mission. In the rest of this paper, I assume that the external property specification is organized according to external functions. An **external function** is defined here as a portion of the external SuD interactions that is of use for some actor in its environment.[2] External system functions refine (14) the responsibilities mentioned earlier in the specification of the system mission (figure 4). The refinement relationship between system mission, system responsibilities and system functions can be represented by a **function refinement tree** with the mission at its root and the functions at its leaves. Such a tree is really a traceability structure that represents links between various levels of refinement.

[2] This agrees with the concept of a *use case* [11].

System mission:

- The MSS will support the scheduling of meetings.

Responsibilities:

- Determine, upon request, a meeting date that satisfies the exclusion and preference constraints of the potential participants.
- ...

Exclusions:

- The MSS shall not support other aspects of organizing a meeting (such as payments, bookings etc.)

Figure 4: Example mission specification.

External functions:

- **Event 2.6**: Ask participants to give their excluded and preferred dates before a certain deadline.
  - **Meaning**: Initiator requests the MSS to ask each participant to give its excluded and preferred dates before a certain deadline.
  - **Desired response**: The MSS passes the question on to the participants, together with the deadline.
  - **Guard**: The meeting, initiator, partipants and date range must exist.

Figure 5: Example function specification.

Figure 5 gives an example function specification. In contrast to the requirements specification, a function specification is very much about the system. It treats the system as actor among other actors in its environment. Events may arise from external actors (15) defined in a context model, or from the passage of time (temporal events), and each system response goes to one or more external actors (16). The meaning of an event is described using terms defined in the dictionary (17) that refer to the subject domain or that are introduced by the function (18). The guard of the function gives a condition on the subject domain that must be true for the event to necessitate a response (19). Other information that can be added include pre/postconditions, a context diagram, etc. These are not given here. There will also be links (20) among functions, such as temporal dependency.

4 Conceptual decomposition

A **conceptual decomposition** of a software system is a decomposition into components that correspond to entities or activities in the external environment of the system. Because of this correspondence, traceability of the conceptual components to these parts of the external environment is optimal. This should also make the conceptual decomposition understandable to the user. The conceptual decomposition is the essence of what any implementation must do to realize the desired external functions. It corresponds to the *essential model* of structured analysis [14] and to the *specification model* of Syntropy [1]. The conceptual decomposition acts as an intermediary between external property specification and implementation-level decomposition, and should therefore improve traceability between these two design levels.

Structured analysis distinguishes three kinds of conceptual components: data stores, data transformations and state machines, which are closely coupled. By contrast, OO methods recognize only one kind of component, the object, that encapsulates data, data transformations and control. Because this reduces the number of spurious links within a decomposition, we use objects as components.

4.1 Component declaration

The essential decomposition of a software system is represented by a class diagram. Figure 6 gives an example. Since the class diagram technique is also used to represent a decomposition of the subject domain, it is easy to confuse a software decomposition model for a subject domain decomposition model and vice versa. To make absolutely clear that these are software components, the class names in the software decomposition model are suffixed with an _S.

There are different possible decomposition criteria for a software system. In **subject-domain-oriented decomposition**, conceptual software components correspond (21) to subject domain components (figure 6). This has good traceability to the subject domain and bad traceability to the external functions. In **functional decomposition**, software components correspond (22) to external system functions. This is the classical structured analysis decomposition

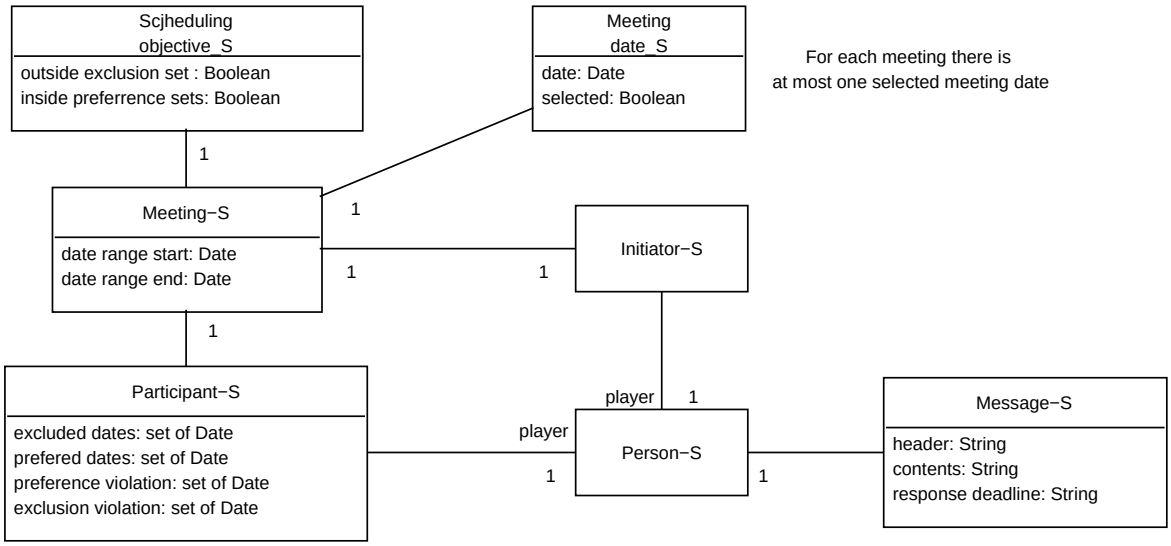

Figure 6: **Conceptual decomposition.**

criterion, with good traceability to external functions and bad traceability to the subject domain. We can also combine the two criteria, as in JSD [9] and Objectory [11], by distinguishing *surrogate objects*, that represent subject domain objects, from *function objects*, that implement software functions. Yet another decomposition criterion is **event partitioning**, according to which software components correspond (23) to external events [14]. Another decomposition criterion used in structured analysis is **device partitioning**, according to which software components correspond (24) to devices (actors) in the context of the system [19, pages 325, 517]. Still other partitionings are also possible. A judicious mix of decomposition heuristics should lead to a conceptual structure that is backward traceable to the external environment and forward traceable to the underlying implementation.

Whatever the decomposition criteria used, there are links (25) between components and external functions, whose meaning is that the external function is realized by the cooperation of one or more components. This can be represented by a **function decomposition table** such as the one shown in figure 7, with conceptual software object classes laid out horizontally and external functions laid out vertically. Each row of this table shows the objects in the mechanism by which a function is conceptually implemented. Each column represents the interfaces that an object needs to participate in these mechanisms. The table is a variant of the well-known traceability table of systems engineering [2, page 193] and the CRUD tables of Information Engineering [13].

4.2 Communication structure

Two important aspects of any decomposition is the relationship between different components in "space" (communication) and between the events of one component in time (behavior). These are treated next.

This can be represented by an undirected higraph [8] called a **communication diagram** (figure 8). The nodes represent the object classes of the conceptual decomposition (26) and the edges communication links [15]. These differ from the associations represented in the class diagram, because in the communication model they represent synchronous communications. Each communication links must agree with the class interfaces declared in the class diagram (27).

4.3 Component behavior

Each of the component objects has an interface through which it responds to events. When receiving an event, an object may change state and send out an action (which presumably is received as event by another object or by an external actor). This interface is declared in the class diagram and must agree (28) with the events and actions used in the behavior model and the communications represented in the communication model (29).

To keep matters simple, we assume that behavior is represented by an extended Mealy state transition diagram (STD), which is a directed labelled graph with an initial node. Each node represents a state and each edge a state transition. The edges are labeled by terms of the form event[guard]/updates & actions. The event is received along one of the communication links of the object, the guard tests the local object state and event

	Person_S	Participant_S	Initiator_S	Meeting_S	Message_S	Scheduling_ objective_S	Meeting_ date_S
2.6 ask dates					C		
2.7 give dates		U					

Figure 7: Function decomposition table for the two functions. The entries represent Create, Update, Read and Delete actions on objects.

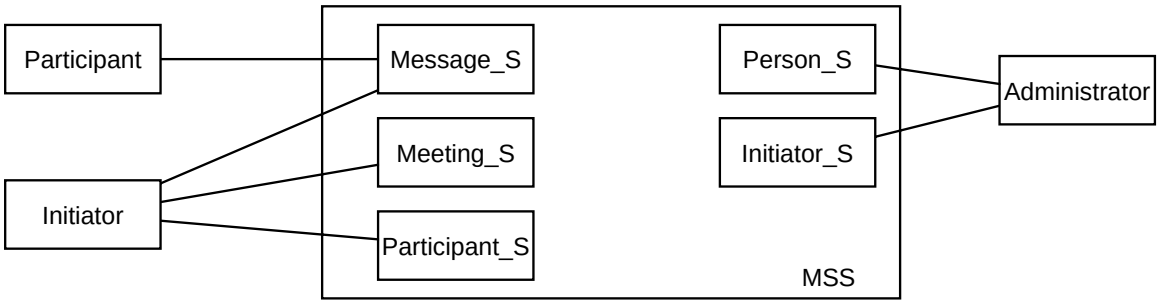

Figure 8: Communication model of the MSS conceptual decomposition.

parameters, the updates may change the state and the actions are messages sent out along the communication links of the object. Space limitations prevent a more detailed treatment. Each STD corresponds to a class in the class diagram (30), its local state is represented by the attributes declared in the class diagram (31), its events and actions must be declared in the class diagram (28) and they must agree with the communication links in the communication diagram (29).

5 Summary and Conclusions

Table 1 summarizes the partitioning criteria mentioned for the different design levels. The table is not exhaustive but gives an impression of the differences at different design levels. Because different criteria may use to different modularizations, this illustrates the claim that there are potential gaps in modularization between the different levels. Addition of an implementation level would show that for that level, yet other partitioning criteria apply, such as physical proximity to the source or destination of data, performance criteria and network topology. Clearly, we should not expect a seamless transition between the different levels, as is claimed by many object-oriented methodologists.

Because we cannot expect to have isomorphic design structures at different levels, we must use explicit links to maintain traceability across levels. Table 2 lists the links mentioned in the paper. To limit the size of the table, the business mission and business purposes have been lumped together as *business purposes* and system responsibilities have been omitted. The sources of requirements (stakeholders such as users or sponsors) have not been included either. In summary, the meaning of the links is as follows.

(1) Requirements are linked to business objectives.

(2) Requirements are linked to each other by means of various dependencies.

(3) Requirements are linked to their sources. (Not shown in the table.)

(4–7) The dictionary defines the meaning of class names, association names, attribute names and operation names.

(8) The terminology in the requirement specifications must agree with the terms defined in the dictionary.

(9) Dictionary entries may refer to each other.

(10) The system purpose must be motivated by the business objectives.

(11) The responsibilities of the system must be linked to the system purpose. (Not shown in the table.)

(12) Desired external functions must be linked to the requirements that motivate them.

(13) Desired external functions mut be linked to the system purpose.

Require-ments	Actors	Depart-ments	Workflow	Work procedu-res					
Proper-ties	Actors		Workflow	Work procedu-res	Functions	Features	Events	Respon-ses	
Con-ceptual compo-nents	Devices				Functions	Features	Events		Subject domain

Table 1: **Decomposition criteria.**

	Business purpose	Context model	Requirements specification	Subject domain model	Dictionary	System purpose	System function specification	Class model	Object behavior model
Business purpose									
Context model									
Requirements specification	1		2						
Subject domain model									
Dictionary			8	4, 5, 6, 7	9				
System purpose	10								
System function specification		15, 16	12	19	17, 18	11, 13	20		
Class model		23, 24		21			22, 25		
Object behavior model								28, 30	
Object communication model								26, 27, 31	29

Table 2: **Links between parts of a coherent specification.**

(14) Desired external functions must refine the responsibilities of the system. (Not shown in the table.) This creates a redundancy in the traceability links that must be kept consistent: An external function must be motivated by the system purpose, which in turn must be motivated by business objectives. An external function must also be motivated by a requirement, which in turn must be motivated by business objectives. Traveling these two chains of links must yield the same result.

(15) Each external function can be specified in event/response form, where the event arises from an external actor shown in the context model.

(16) Each response of a function goes to an external actor, modeled in the context model.

(17, 18) The specification of the meaning of an event can use terms defined in the dictionary. These refer to the subject domain or are introduced in the function specification itself.

(19) The guard of an external function must be defined in terms of the subject domain.

(20) There can be various links between external functions, such as temporal dependency.

(21) In a subject-domain-oriented decomposition, the components of a conceptual software decomposition correspond to the components of the subject domain.

(22) In a functional software decomposition, the components of a conceptual software decomposition correspond to external functions.

(23, 24) In event partitioning and device partitioning, software components correspond to external events or external actors, both represented in the context model.

(25) Components are linked to the external functions that they realize. This can be represented by a function decomposition table.

(26) The nodes in a communication diagram correspond to the nodes in a class diagram of the software decomposition.

(27) Each communication link in the communication diagram must agree with the class interface declared in the class diagram.

(28) The events and actions used in a state transition diagram model of a class must agree with the interface of the class declared in the class diagram.

(29) Each communication link in the communication diagram must agree with the events and actions used in the state transition diagrams.

(30) Each state transition diagram corresponds to a class declared in the class diagram of the software decomposition.

(31) The local state of the state transition diagram is defined by the attributes declared in its class, declared in the class diagram of the software decomposition.

Note that many link types are present in any model that uses these techniques, but that some links arise as the result of design decisions. This particularly concerns links (12) between requirements and external properties, and (21–24) between external properties and the conceptual decomposition. It is here that modularization decisions affect traceability of the design.

The above list illustrates that a coherent software specification is possible in which we use techniques borrowed from structured as well as object-oriented analysis and design. Using these links, cross-diagram consistency checks can be designed. To pursue this idea, current research includes the definition of a methodological framework called TRADE (Toolkit for Requirements and Design Engineering) that allows a precise definition of the links. The framework is currently populated with a number of simplified UML techniques for the external property and conceptual design levels, as well as other techniques to specify business and system purposes and express component communication other than by means of scenarios (as is done in the UML), and traceability tables. The links will be made prices in two ways. First, formal counterparts of the major techniques will be given and the links will be defined in terms of this formalization. Second, a metamodel for the techniques and their links will be given, that represents the formally defined links in a more intuitive way. We are working on a software toolkit called TCM [4] to support the use of these techniques. The goal is to make specification links explicit by means of TCM, to have it tolerate inconsistencies across links, and to provide support for going back and forth across design levels to make the documents mutually consistent. The metamodel will be used in TCM to design cross-diagram consistency checks.

Acknowledgements: This paper benefitted from comments made by Maarten van Steen and by the anonymous referees.

References

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

[2] A.M. Davis. *Software Requirements: Objects, Functions, States*. Prentice-Hall, 1993.

[3] F. Dehne and R.J. Wieringa. The Yourdon Systems Method and the Toolkit for Conceptual Modeling. Technical Report IR-414, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, December 1996.

[4] 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.vu.nl/~tcm.

[5] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press/Prentice-Hall, 1978.

[6] B. Duterte and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[7] O. Gotel and A. Finkelstein. Contribution structures. In *Second IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Press, 1995.

[8] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, 1988.

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

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

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

[12] G. LeLann. The arianne 5 flight 501 failure — a case study in system engineering for computing systems. Technical Report 3079, INRIA, December 1996. http://www.inria.fr/RRRT/RR-3079.html.

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

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

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

[16] K. Shumate. Structured analysis and object–oriented design are compatible. *Ada Letters*, 11(4):78–90, May/June 1991.

[17] R.J. Wieringa. A survey of structured and object-oriented software specification methods and techniques. Technical report, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1997. To be published, *ACM Computing Surveys*.

[18] R.J. Wieringa. Using the tools in TRADE, II: Specification and design of a meeting scheduler system. Technical Report IR-436, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, November 1997. ftp://ftp.cs.vu.nl/pub/roelw/97-TRADE02.ps.Z.

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