A Framework for Understanding the Integration of Design Methodologies*

Xiping Song
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540
song@scr.siemens.com

Abstract

Software researchers and practitioners have studied and used a number of approaches to integrating major design paradigms in order to improve Software Design Methodologies (SDMs). Software tool developers have developed tools to aid the integration and customization of existing SDM support tools. However, a framework for understanding and guiding various integration and customization processes is still lacking. Because of this users, even with tool support, often fail to systematically integrate SDMs and SDM support tools.

In this paper, we define a framework that can be used to understand various SDM integrations and customizations. Through this definition, we describe what kinds of integrations are useful, what difficulties are to be met, and how the integrity of the SDMs can be maintained.

1 Introduction

1.1 Motivation

Designing software systems for different application domains often requires the use of different Software Design Methodologies (SDMs). As a consequence, a large number of SDMs have been developed during the past two decades and used in software development. These SDMs often emphasize support for different phases in various software development lifecycles (e.g., some emphasize support for problem analysis phase while some emphasize support for system design phase). These SDMs have different strengths and weaknesses in supporting software modeling and documentation. These SDMs are different in their qualities. For example, some SDM may provide a more complete process while it provides less expressive design notations.

Because of this, the SDM community has suggested the integration of SDMs—taking the best characteristics from several existing SDMs to form one more comprehensive SDMs (e.g., Fusion [CAB+94]), or taking certain appropriate characteristics from various SDMs to form various domain specific SDMs.

In the past few years, many papers [War89, Ala88, Con89, Jal89, RBP+91, Boo91, YT90, Wie91, BC91, WPM90, Hei87]) have addressed SDM integration. We have found that these papers focused on discussing only the specific strategies to be used for integrating certain SDMs. Most of them (e.g, [Jal89, WPM90, Hei87]) focus on discussing how to integrate the object oriented design paradigm with the functional or data flow paradigm. Some of them (e.g., [BC91]) discuss how to integrate Jackson System Development (JSD)[Jac83] with Booch's Object Oriented Design method (BOOD) [Boo91]. Some of them present new SDMs (e.g., Object-oriented Modeling Technique (OMT)[RBP+91]) that are based upon an integration of the object oriented paradigm, the state transition paradigm and the data flow paradigm. In OMT [RBP+91], the object-oriented paradigm is used to define the static structure of a software system. The state transition paradigm is used to describe the dynamic behaviors of the system. The data flow paradigm is used to define the functions of the system. We found that many new Object Oriented Design Methods (OODs) ([SM88, SGME92, RBP+91])

*This research was supported by the Advanced Research Projects Agency, through ARPA Order #6100, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. This work is also sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1012. Support was also provided by the Naval Ocean Systems Center and the Office of Naval Technology.

use similar strategies to integrate these paradigms. In this paper we shall refer to this kind of integration, which integrates whole design paradigms or techniques, as high-level integration. We found that the SDM community has been focusing on this High-Level Design Method Integration (HLDMI), and has successfully addressed many of the issues involved.

In this paper, we define a framework that helps organize the understanding of HLDMI. More importantly, perhaps, we also use this framework to explain the Low-Level Design Method Integration (LLDMI) which integrates *parts* of the design paradigms and documentation techniques. We believe that, while HLDMI has been well addressed by design methodologies at a conceptual level, the actual implementation of the SDMs will often be explored by software designers. For example, designers may incorporate a notation coming from another SDM into BOOD to express the roles an object plays. Designers may integrate BOOD's measures for evaluating the design of a class into the Object Oriented Technique (OMT) [RBP+91].

There are a number of difficulties in achieving this LLDMI.

1. SDMs have evolved from sets of simple, informal programming guidelines into very complex information products. Integrating a new method into an SDM may cause potential conflicts, such as a conflict between the semantics of the notations and terminologies used in the different SDMs.

2. An integration of SDMs requires both broad and deep knowledge of the SDMs. As the SDMs are complex, it would be desirable for an SDM expert to do the integration. However, as the low-level integration is likely to be project-specific, and thus to be made by designers, their knowledge of the SDMs may be insufficient.

3. An integration process and framework are lacking. Though some work (e.g., [Pot89, SO94]) has addressed this issue, no comprehensive strategy has been defined and used to systematically customize and integrate SDMs. No technique or structure has been defined and used to record customizations and integrations.

These difficulties also render some SDM customization tools ineffective. These tools (often called Meta-CASE tools) support customization of of SDM support tool. However, tool vendors have found that, without an explicit guidance, the tool users are usually unable to to customize the SDM even though the tool provides many potentially useful capabilities (e.g., creating various notations and defining their semantics) [itMs92]. Therefore, a framework for understanding the various integrations and their relationships is essential for aiding designers in their integration efforts. Moreover, a process, possibly based upon this framework, that can be used to guide integration should be very useful for integrating SDMs.

1.2 Goals

In this paper, we define a framework that can be used to understand various kinds of SDM integrations. Through this framework, we identify and describe what kinds of integrations are useful. We also discuss the difficulties that can be expected and how to maintain the consistency of SDMs in carrying out the integrations.

To define this framework, we first analyze the structures of SDMs and then formalize these structures into a standard SDM model. Based upon this model, we define a framework for understanding various kinds of SDM integrations.

Section 2 defines the standard SDM model. Section 3 defines our framework for understanding SDM integrations.

2 Our Model of SDMs

In this section, based upon our analysis of a large number of SDMs, we define a standard model. This model characterizes the components of an SDM and the relationships among these components. This model is an enhanced version from a model we developed previously for classifying parts of SDMs [SO92a, SO92b].

This model is defined based upon our analysis of the following SDMs:

1. Jackson Systems Development [Jac83] (JSD),
2. Rational Design Methodology [PC86] (RDM),
3. Logical Construction of Programs [War76] (LCP),
4. Data Structured System Development [Orr77] (DSSD),
5. Structured Design [SMC74] (SD),
6. Ward/Mellor's Real Time SDM [WM85] (Ward),
7. Shumate's SDM [Shu91] (Shumate),
8. Booch's OOD [Boo91] (BOOD),
9. Jacobson's OOD [Jac87] (JOOD),
10. Rumbaugh *et. al.*'s OMT [RBP+91] (OMT),
11. Jalote's extended OOD [Jal89] (EOOD), and
12. Shlaer/Mellor's OOA [SM88] (SMOOA).

We selected these SDMs because they represent diverse approaches to software design. Three of them are based on data-flow modeling. Two of them are based on hierarchical data structure modeling. The last five are based on object-oriented modeling. Thus we believe that a model representing all of them, should be broad enough to represent most contemporary SDMs.

2.1 High-Level Components

Our previous experiences [SO92a, SO94] have indicated that entity-relation modeling is effective in characterizing SDMs. Thus, we used this modeling technique to define our SDM model (see Fig. 1). Our model defines an SDM as consisting of two major parts. The first part, as shown in the right part of Fig. 1, comprises *the high level* components of an SDM. As illustrated, an SDM defines or adapts a number of artifact models, and defines a design process which is suggested for use to develop the design artifacts. We define these entities as:

1. **Artifact Models**, which are structures and/or patterns for specifying and organizing design artifacts. These models define their components and the semantics of the relationships among these components. Using these models and following their underlying semantics, designers can effectively handle system complexity and construct artifacts to characterize the targeted problem domain or software systems. (e.g., the object model in OMT is an artifact model).

2. **Properties**, which are characteristics of design artifacts desired by designers in using the artifact models. An SDM is always aimed at producing artifacts that have some superior properties, (e.g., design artifacts should be easy to understand and modify).

3. **Principles**, which are concepts used to help in producing artifacts that have the properties desired by customers and designers. An SDM either justifies new design principles or adopts some existing principles as the basis of the SDM, (e.g., object-oriented SDMs use principles of information hiding and abstraction as their bases).

4. **Representations**, which are the means used for expressing the artifact models. They are aimed at improving the precision with which an artifact is specified and at improving the comprehensibility of artifacts. One artifact model could have multiple representations. They could be languages, sets of diagrammatic notations, (e.g., data flow diagram), etc.

5. **Processes**, which are sequences of steps an SDM suggests for designers to use in developing a software system using certain artifact models, (e.g., the design processes defined in JSD and OMT).

The type **representation** is further classified based on the level of formality with which a representation is defined in the selected SDMs.

1. **Structural representation**, which is a type of representation through which artifacts are captured in the form of diagrammatic notations. This type of representation is used to highlight important components and inter-component relations in a complex software design, improving the comprehensibility of the design. Examples include data flow diagrams, structure charts, etc.

2. **Mathematical representation**, which is a type of representation through which artifacts are captured using mathematical notations and mathematical operations that are performed on them. Examples include set and relation theories. (e.g., RDM uses relations to specify system functions).

3. **Linguistic representation**, which is a type of representation through which artifacts are captured in the form of statements in a defined language. Examples include English, various design languages, and templates (e.g., BOOD [Boo91] uses template to specify object)

2.2 Low-Level Components

The second part of the model, as shown in the left part of Fig. 1, defines *the low level* components of an SDM. We define these entities as:

1. **Model Components**, which are the components of the artifact model. They provide semantics for specifying and organizing descriptions/specifications of entities involved in software design activities (e.g., class in the object model of OMT). Model components are used in all SDMs to define the various structures and models of a software system.

2. **Criteria**, which are rules advocated for use by designers in deciding if an artifact is an instance of a model component (e.g., to decide if door is a class in designing an elevator control system). An SDM usually provides a few criteria that serve as

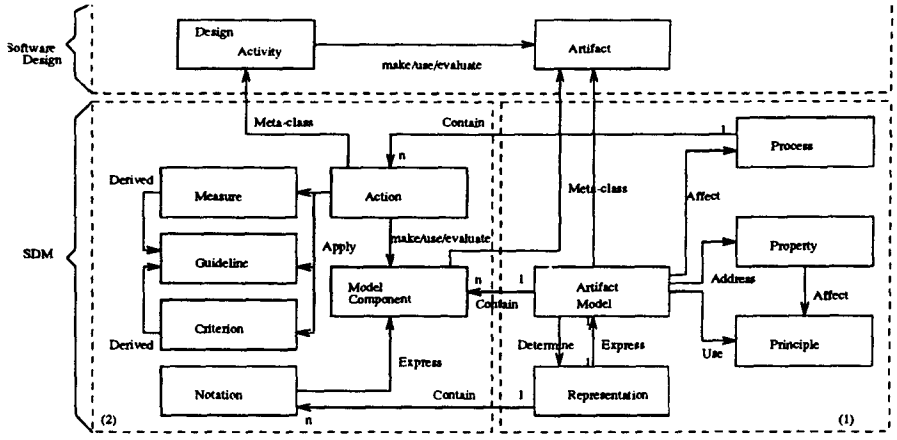

Figure 1: An Entity Relational Model of SDMs

the necessary conditions for deciding what model component an artifact is, (e.g., JSD defines the criteria for deciding a JSD entity. RDM provides a set of rules used to decide how to decompose a design document into a tree of module specifications).

3. **Guidelines**, which are concrete strategies, heuristics, or techniques advocated for use in identification and specification of artifacts and how they are to be structured into the artifact models. Guidelines are often described by giving *examples* of the model components. For instance, BOOD indicates that device, system, people, and location can be examples of objects. BOOD also suggests a guideline that using informal English analysis techniques [Abb83] can help in identifying objects.

4. **Measures**, which are quantifications with respect to some standard, or samples used for quantitative comparison or evaluation of the quality of artifacts that are structured in the artifact model. Some SDMs define measures to help quantify the degree to which various artifacts demonstrate desired properties, (e.g., Structured Design defines different bindings (e.g., functional, logical) and uses them as the basis for a measure of the cohesiveness of a program design).

5. **Notations**, which are means for expressing the artifacts that are identified and specified according to the model components. They are parts of the representation, (e.g., the rounded box is a notation in the object diagram of OMT).

6. **Actions**, which are physical and/or mental processing steps used for developing the artifacts that are structured in the artifact model. An action may create, modify or use an artifact. An action may also evaluate an artifact and then decide if it needs further development.

The type **action** is further classified based on the technical nature of the actions described in the selected SDMs. Note that the definitions in the decomposition that follows are aimed at capturing the actions that are described in various SDMs. They do not necessarily cover all fundamental design activities as are described in [Fre83]. The subtypes of **action** are defined as:

1. **Develop**, which is a high-level design action that is aimed at producing a major part of a complete design specification. It is often defined as a major development phase of an SDM, consisting of various kinds of design actions. Examples include developing the system specification in JSD.

2. **Modeling**, which is an abstraction action that is aimed at characterizing certain aspects of a system in order to aid the analysis and evaluation of a system design before its implementation. Modeling is often defined as a development phase of an SDM, consisting of various low-level design actions (e.g., the **Define** action). Examples include the modeling of the environment outside the system in JSD.

3. **Decompose**, which is the action that, according to certain artifact models, subdivides an artifact into small pieces so that the artifact can be more

readily understood, defined, and specified. Examples include decomposing a function in Structured Design (SD).

4. **Specify**, which is the action of elaborating the details of a design, often done in a descriptive manner. Examples include specifying implementation of a module in BOOD.
5. **Define**, which is the action of completely specifying the semantics of certain artifacts, often done in a declarative manner. Examples include defining the interface of a module in BOOD.
6. **Derive**, which is the action of constructing design artifacts from other previously developed artifacts by following some well-defined guidelines in SDMs. Examples include deriving a program from the data structure of its output in DSSD.
7. **Identify**: which is the action of finding artifacts that should be defined and specified as instances of the model components. Examples include identifying objects in BOOD.
8. **Select**: which is the action of choosing from, among a set of candidates, the ones that satisfy some given criteria. This action can be a part of an **Identify** action. Examples include selecting entities in JSD.

2.3 Relationships

Some of the relationships in Fig. 1 are quite self-explanatory and seem satisfactorily explained through the definitions of the method component types given earlier. Some other relations seem less clear and can benefit from further explaination such as the following:

1. Processes, artifact models and representations **contain** actions, model components and notations, respectively. Processes can be procedurally or functionally described sequences of actions. Models can be structures that define the relationships among various model components.
2. Actions **apply** certain concepts. Composition of an action and execution of its sub-actions are influenced by these concepts. For example, the action of identifying an object in BOOD **applies** the guideline for identifying the nouns in an informal problem description.
3. Properties **affect** the development of principles. For example, producing an easily changeable software design **affects** the development and uses of the principle of "information hiding".
4. Structures or models of an artifact **determine** representations used for expressing the artifact. For example, the object-oriented model **determines** the semantics to be supported by an object/class diagram (e.g., Booch's object-diagram).
5. Guidelines can be **derived** from criteria or measures. For example, the criteria for deciding what an object is (e.g., it must have an identity) can also be used to derive guidelines for identifying objects.

In the next section, we define a framework, based upon this SDM model, for understanding SDM integration.

3 Our Integration Framework

In surveying existing SDMs and their proposed integrations, we found that there are two primary approaches to the integration of SDMs. We call the first **function-driven integration**, in which new functional capabilities are added to an SDM. These capabilities are directly useful for modeling the problem and/or software systems, and for supporting a software development life-cycle. Integrating object oriented design and the Structured Design is an example of the function-driven integration.

We call the second approach **quality-driven integration**, which adds no new functional capability to an SDM, but which improves its quality or usability. For example, the SDM quality improved could be expressiveness and understandabilities [Kun83]. The usability of an SDM is concerned with how effectively an SDM can be used in designing a software system. Adopting a new, expressive representation to substitute for an old, less expressive one is an example of how such integration, which does not directly add capabilities, does nevertheless improve SDM effectiveness. In the next two sections, we define a framework for understanding the issues involved in carrying out these two kinds of integrations. We discuss what kinds of integrations have been done and what other kinds we expect to see.

3.1 Function-driven Integration

3.1.1 High-Level Integration

HLDMI integrates the high-level components of SDMs. design methodologies have proposed numerous function-driven, HLDMIs, which seem to fall into

the following categories.

Property Integration:

Motivation: Designers sometimes find that a particular SDM does not effectively support producing software systems that have certain desired properties. (e.g., ease of maintainence, software reusability).

Benefit: A proposed property-integrated SDM will be able to help in producing a software system that is superior with respect to these properties. This is generally done by designing the new SDM to incorporate the additional development phases.

Note: Support for producing software systems that have a desired quality may require the integration of new design principles and artifact models from other SDMs. This may require integrating new design representations and processes (see Fig. 1).

Principle Integration:

Motivation: 1) Sometimes SDM authors find that other SDMs have applied new and promising design principles. In order to compete with these SDMs, these authors enhance their SDMs to support these new principles. 2) Property integration can sometimes also lead to the principle integration (see Fig. 1).

Benefit: Supporting new design principles seems to be essential in assuring improvement in SDMs.

Note: Not all design principles used in software design are compatible with each other. The new design principles to be incorporated might conflict with principles already used in the SDM. For example, Shumate's SDM relies on the principle that functional modeling is right for problem analysis because functional modeling is the foundation for systems engineering [Shu90]. Therefore, introducing the object-oriented design principle that modeling the structure of an application problem is more essential than modeling system functions, will lead to a conflict with Shumate's method.

Artifact Model Integration:

Motivation: 1) Desire to enhance an SDM to support the modeling of additional major aspects (e.g., the behavioral aspect) of an application problem or software system. 2) Principle integration and property integration will sometimes lead to artifact model integration. 3) Some SDMs are only partial approaches (e.g., BOOD) to designing software systems, and they need to be integrated with other SDMs to help produce the products of other development phases, thereby producing a more complete product.

Benefit: The SDM will be able to support the modeling of an additional major aspect of the application problem or software system. The SDM can be used to support a broader segment of the overall software development life-cycle.

Note: 1) The integration must clearly define the relationships between the new model and the old models that have been used in the SDM. Each model should be taken as a separate view of the design artifact. For example, OMT describes clearly how the object model, dynamic model, and function model relate to each other. OMT describes what aspects each model will be used to characterize. 2) The integration must define relationships between the products of the different SDMs being integrated, i.e., how a product of one SDM, which supports one particular phase of a development life-cycle, can be used by another SDM to support another phase of the software development life-cycle.

Process Integration:

Motivation: Artifact model integration leads to process integration. Some SDM authors describe how steps in the old process are to be coordinated with steps in the process used for specifying the new artifact model. It is rare that process integration is the major motivation for SDM integration.

Benefit: Incorporating a new process into an old process appropriately can enable the two processes to provide guidelines to each other. For example, [Jal89] described how a functional decomposition process and an object decomposition process can guide each other. Incorporating a new process into an old process can also support a broader segment of the software development life-cycle of software development process life-cycle.

Note: This may cause the integrated design process to be highly concurrent and cooperative (e.g., [Jal89]). Thus, this can increase both the power and the complexity of the SDM process significantly.

Representation Integration:

Motivation: 1) Artifact model integration often leads to representation integration. A new representation is generally integrated into an old SDM along with the integration of the corresponding artifact model.

Benefit: It is often easier to integrate an existing representation than to invent a new one. The representation to be integrated is often already well known and has generally been evaluated as a vehicle for expressing the integrated artifact model. For example, the data flow diagram has been adapted into many SDMs which incorporate the data flow model.

Note: This may cause conflicts between the representations. However, normally, SDM authors separate the new model from the old models. Thus, the new model represented in the new representation can hardly have conflicts with the old representation. For example, OMT integrates the data flow and state transition models, and separates them from the object model. Each of these models has its own set of notations. For example, the object model uses the object diagram; the dynamic model uses the state diagram; although both representations employ the rounded box notation, the semantics of the rounded boxes depend on the representation within which they are used, and thus can be distinguished clearly.

3.1.2 Low-level Integration

As indicated earlier, we have shown that how the framework is constructed to understand HLDMI. In this section, we will show how the framework is constructed to understand LLDMI. Using our standard SDM mode, we view that LLDMI integrates the low-level components as shown in the left part of Fig. 1.

Model Component Integration:

Motivation: SDM comparisons illustrate the comparative weaknesses of an SDM. This often motivates the SDM authors to add new model components into the SDM. One approach to achieving this is to integrate the model components used in other SDMs into the old SDM.

Benefit: The SDM will be able to support the modeling of the additional aspects of application problem and software system.

Note: There might be reasons for the components were not provided originally in the old SDM. For example, most modeling formalisms for object oriented analysis do not support specification of the visibility of an object because specifying the visibility is often not required until the design phase. Thus, one should ensure that the added features are consistent with the ways in which the model will be used.

Criteria/Guideline/Action/Measure Integration:

Motivation: Model component integration can lead to integrations of criteria, guidelines, actions, and measures. As illustrated by Fig. 1, actions will use the corresponding criteria to decide artifacts which are instances of the added model components. The actions will use the guidelines to specify these artifacts, and use the measures to evaluate these artifacts.

Notation Integration:

Motivation:) Model component integration sometimes leads to notation integration.

Benefit: The SDM can have a more expressive and complete representation.

Note: This may cause conflicts as an integrated notation might be same with an old notation which denotes the different model component. For example, box represents different meanings in OMT and BOOD. This kind of conflict must be resolved before a notation can be adopted.

3.2 Quality-driven integration

The quality-driven integration does not incorporate any new components into the artifact model to model additional characteristics of the problem and system. Instead, the quality-driven integration is aimed at improving SDMs' quality and usability in supporting their existing features.

3.2.1 The High-Level Integration

Because not incorporating any new artifact model and not supporting any new property, the quality-driven integration does not integrate any new design principles. Thus, the high-level, quality-driven integration can be made only in the process and representation aspects (see Fig. 1). However, since an artifact model constrain its process and representation, the artifact model constrains the quality-driven integration in the process and representation. We believe that, the most

possible quality-driven integrations at the high-level are representation integrations, adopting more expressive notation.

Representation Integration:

Motivation: 1) An SDM may adopt other SDMs' representations to express a part or all of its artifact model. 2) An artifact model may need to be expressed in another kind of formality. For example, an SDM author may integrate a mathematical representation to express an artifact model (e.g., integrate the notations defined in [PM91] to express system functions).

Benefit: Increase the expressiveness, understandability, and formality of the representation.

3.2.2 Low-Level Integration

SDM books (e.g., [Boo91]) often include many low-level, quality-driven integrations. The SDM authors adopt examples, guidelines, measures, and actions from some other SDMs to improve the usability of their own SDMs.

The low-level, quality-driven integration can also be seen in the practice. Designers adopt examples, guidelines, measures, and actions from some other SDMs to customize the SDM that is to be used.

Guideline Integration:

Motivation: 1) Some other SDMs provide additional, useful and complete guidelines. 2) Examples provided in some other SDMs are more comprehensive and/or close to the application domain of the project to which the SDM is applied to (e.g., banking systems). 3) Examples provided in other SDMs use an implementation mechanism (e.g., programming language) which is similar to the mechanism that the SDM users use. These integrated guidelines are more directly helpful for the designers.

Note: Similar components in the different artifact models may still have differences. These differences may cause it inappropriate for designers to directly borrow and apply the guidelines and examples from other SDMs.

Measure Integration:

Motivation: An SDM may not provide any measure for evaluating the quality of an artifact (e.g., how well the interface of an object is defined). However, other SDMs may provide the measures for evaluating the similar artifacts. For example, Booch adopted measures *coupling* and *cohesion* of Structured Design into BOOD to measure the quality of objects.

Action Integration:

Motivation: Specifying an artifact entails designers to perform various kinds of actions (e.g., modeling, selection, as indicated by our decomposition (page 4)). However, an SDM may not define all these kinds of actions. For example, the Shlaer/Mellor Object-Oriented Analysis method(SMOOA) does not define a class/object selection process. However, users of SMOOA could adopt such a process from BOOD or some other SDMs (e.g., JSD) to improve the usability of SMOOA. This provides more detailed and complete design procedures for specifying artifacts.

Notation Integration:

Motivation: An SDM may not provide a notation for expressing a component of its artifact model. For example, BOOD defines what a derived object is, however, does not provide a notation for expressing the derived object. Integrating a new notation helps in expressing the model component and thus making the representation more complete.

Acknowledgements

The work described in this paper was carried out at Information and Computer Science Department of Univ. of California at Irvine. The author is very grateful to Prof. Leon Osterweil for reviewing an early version of this paper.

References

[Abb83] R. Abbott. Program design by informal English descriptions. *Comm. of ACM*, 26(11), 1983.

[Ala88] B. Alabiso. Transformation of data flow analysis models to object oriented design. In *oopsla88*, pages 335–353, San Diego, CA, Spet. 1988.

[BC91] A. Birchenough and J. Cameron. JSD and object-oriented design. In J. Cameron, editor, *JSP and JSD: The Jackson Approach to Software Development*, pages 293–303. IEEE Computer Society, 1991.

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

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION method.* Prentice Hall, 1994.

[Con89] L. L. Constantine. Object-oriented and structured methods: Towards integration. *American Programmer*, 7(2), August 1989.

[Fre83] P. Freeman. Fundamentals of design. In *Tutoral: Software Design Techniques*. IEEE Computer Society Press, Washington, DC, 1983.

[Hei87] M. Heitz. *HOOD: Hierarchical Object-Oriented Design for development of large technical and realtime software*. CISI Ingenierie, November 1987.

[itMs92] Discussions in the MetaCASE session. In *The 5th international workshop on computer-aided software engineering*, July 1992.

[Jac83] M. Jackson. *Jackson System Development*. Prentice-Hall International, 1983.

[Jac87] Ivar Jacobson. Object oriented development in an industrial environment. In *OOPSLA 87*, pages 181–191, Oct. 1987.

[Jal89] P. Jalote. Functional refinement and nested objects for object-oriented design. *IEEE Transaction on Software Engineering*, 15(3):264–270, March 1989.

[Kun83] C. H. Kung. An analysis of three conceptual models with time perspective. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 141–168. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.

[Orr77] K. T. Orr. *Using Structured System Design*. Yourdon Press, NY, 1977.

[PC86] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[PM91] D. L. Parnas and Jan Madey. Functional documentation for computer system engineering. Technical Report CRL Report No. 237, Communications Research Laboratory, McMaster Univ., Sept. 1991.

[Pot89] C. Potts. A generic model for representing design methods. In *Proceedings of 11th International Conference on SE*, pages 217–226, 1989.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[SGME92] B. Selic, G. Gullekson, Jim McGee, and I. Engelberg. Room, an object-oriented methodology for developing real-time systems. In *Proc. of 5th International Workshop on Computer-Aidied Software Engineering*. IEEE CS, July 1992.

[Shu90] Ken Shumate. Software specification and design: Structured analysis, object-oriented design and the transition. In *Tutorial of Tri-Ada 1990 - Software Specification and Design in Ada, Tri-Ada Conference*, Baltimore, MD, December 1990.

[Shu91] Ken Shumate. Structured analysis and object-oriented design are compatible. *Ada Letters*, 6(4):78–90, May/June 1991.

[SM88] Sally Shlaer and Stephen. J. Mellor. *Object-Oriented System Analysis—Modeling the World in Data*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM System Journels*, 13(2):115–139, 1974.

[SO92a] X. Song and L. J. Osterweil. A framework for classifying parts of software design methodologies. In R. W. Selby, editor, *Proceedings of the 2nd Irvine Software Symposium*, pages 49–68. IRUS, March 1992.

[SO92b] X. Song and L. J. Osterweil. Towards objective, systematic design-method comparison. *IEEE Software*, pages 43–53, May 1992.

[SO94] X. Song and L. J. Osterweil. Using meta-modeling to systematically compare and integrate modeling techniques. Available from the authors upon request, March 1994.

[War76] J.D. Warnier. *Logical Construction of Programs*. Van Nostrand Reinhold, New York, 1976.

[War89] Paul T. Ward. How to integrate object orientation with structured analysis and design. *IEEE Software*, March 1989.

[Wie91] R. J. Wieringa. Object-oriented analysis, Structured analysis, and Jackson system development. In *Proc. of IFIP working conf. on the object oriented approach in information systems*, Quebec City, Oct. 1991.

[WM85] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, New York, 1985.

[WPM90] A. I. Wasserman, P. A. Pircher, and R. J. Muller. The object-oriented structured design notation for software design representation. *Computer*, (3), March 1990.

[YT90] W. L. Yeung and G. Topping. Implementing jsd design in ada - a tutorial. 15(3):25–32, July 1990.