

# UC Irvine

## ICS Technical Reports

### Title

Comparing software design methodologies through process modeling

### Permalink

<https://escholarship.org/uc/item/3rd1c0hs>

### Author

Song, Xiping

### Publication Date

1992

Peer reviewed

Notice: This Material  
may be protected  
by Copyright Law  
(Title 17 U.S.C.)

ARCHIVES

Z  
699  
C3  
no. 92-48  
C.2

**Comparing  
Software Design Methodologies  
Through  
Process Modeling**

Xiping Song  
University of California  
Irvine, California 92717

May 1992

**Technical Report No. 92-48**



UNIVERSITY OF CALIFORNIA

Irvine

Comparing Software Design Methodologies Through Process  
Modeling

A dissertation submitted in partial satisfaction of the  
requirements for the degree Doctor of Philosophy  
in Information and Computer Science

by

Xiping Song

Committee in charge:

Professor Leon J. Osterweil, Chair

Professor Richard N. Taylor,

Professor Richard W. Selby

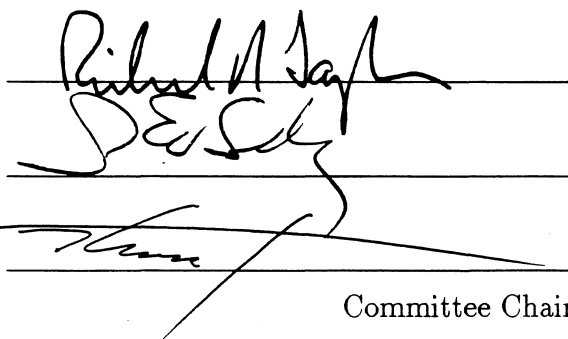
1992

©1992

Xiping Song

ALL RIGHTS RESERVED

The dissertation of Xiping Song is approved,  
and is acceptable in quality and form for  
publication on microfilm:



Richard A. Jaffe  
D. S. Song

Committee Chair

University of California, Irvine

1992



## Dedication

To my family

Juan Yu,

Vivian M. Song

To my parents and brother

Shipu Song

Donquan Chen

Liping Song





# Contents

List of Figures . . . . .	vi
List of Tables . . . . .	viii
Acknowledgements . . . . .	ix
Curriculum Vitae . . . . .	xi
Abstract . . . . .	xiv
<b>Chapter 1 Introduction . . . . .</b>	<b>1</b>
1.1 Comparing Software Design Methodologies . . . . .	1
1.2 Informal Comparisons . . . . .	3
1.3 Quasi-formal Comparison Approaches . . . . .	5
1.4 Motivations for Objective Comparison . . . . .	9
1.5 Strategies for Objective Comparison . . . . .	10
<b>Chapter 2 Problem Definitions and Research Goals . . . . .</b>	<b>13</b>
2.1 The Primary Goal of the Research . . . . .	13
2.2 Secondary Goals of the Research . . . . .	15
<b>Chapter 3 Our Comparison Approach: CDM . . . . .</b>	<b>20</b>
3.1 Step 1: <i>Build_Process_Model</i> . . . . .	21
3.2 Step 2: <i>Classify_Components</i> . . . . .	25
3.3 Comparison of Design Methodologies . . . . .	28
<b>Chapter 4 Experiment 1: Comparing JSD With BOOD . . . . .</b>	<b>32</b>
4.1 Introduction to HFSP . . . . .	33
4.2 Use of CDM . . . . .	35
4.3 Evaluation of CDM . . . . .	57
4.4 Suggested Improvements . . . . .	63
<b>Chapter 5 Supports Needed For CDM . . . . .</b>	<b>65</b>
5.1 Required Supports . . . . .	65
5.2 Why Evolutionary Development . . . . .	67
5.3 Evolutionary Development of BF and MF . . . . .	69
5.4 The BF and MF . . . . .	73

<b>Chapter 6</b>	<b>Experiment 2: Comparison of SDMs</b>	<b>89</b>
6.1	Goals and Design of the Experiment	89
6.2	Step 1: Build Process Models	92
6.3	Step 2: Classify Components	110
6.4	Comparison of BOOD with RDM	118
6.5	Comparison of JSD with SD	125
6.6	Compare DSSD with SD	132
6.7	Compare LCP with DSSD	140
6.8	Evaluation	143
6.9	Status	153
<b>Chapter 7</b>	<b>Assessment of the CDM-based Comparisons</b>	<b>156</b>
7.1	Goals of the Experiment	156
7.2	Design of the Experiment	158
7.3	Comparing the Comparisons	160
7.4	Summary	168
<b>Chapter 8</b>	<b>An Encoding of the CDM Process</b>	<b>173</b>
8.1	Program Structure	175
8.2	Compare SDM Procedure	176
8.3	Definition of the CDM Package	182
8.4	Implementation of the CDM Package	187
<b>Chapter 9</b>	<b>Conclusion and Future Work</b>	<b>195</b>
<b>References</b>		<b>199</b>

# List of Figures

3.1	<i>Compare_Design_Methodologies</i> (CDM), a data flow diagram modeling a SDM comparison process . . . . .	21
3.2	Part I of the <i>Base_Framework</i> : A Model of the Software Design Life-cycle (MSDL). (The diagram inside the broken lined box is a data flow diagram.) . . . . .	26
3.3	Part II of the <i>Base_Framework</i> : types at the top-level of Method Component Type Hierarchy(MCTH) . . . . .	27
3.4	The artifact structure for summarizing the differences . . . . .	31
4.1	A model of JSD specified in HFSP . . . . .	37
4.2	A model of BOOD specified in HFSP . . . . .	39
4.3	Decomposition of the MSDL's component Solution Model Domain .	40
4.4	Classifications of the JSD and BOOD components under the Solution Model Domain . . . . .	44
4.5	The code of the action for identifying <i>Model_Processes</i> . . . . .	47
4.6	The model of defining <i>Model_Processes</i> . . . . .	48
4.7	The model of specifying <i>Function_Process</i> and <i>System_Function</i> .	50
4.8	The domains manipulated by a JSD/BOOD integrated design process	56
5.1	Evolution process of a <i>Evolution_Target</i> . . . . .	69
5.2	Definitions of the factors affecting evolution of BF . . . . .	70
5.3	Definitions of the factors affecting evolution of an MF . . . . .	71
5.4	Evolutionary development processes for Base Framework and Modeling Formalism, represented as a data flow diagram. The numbers labeled on edges indicate a scenario of evolving the BF. . . . .	73
5.5	Part of BF: Definitions of the top-level types in Method Component Type Hierarchy (MCTH) . . . . .	77
5.6	Part of BF: Definitions of the top-level types in Method Component Type Hierarchy (MCTH)(cont.) . . . . .	78
5.7	MCTRM: Method Component Type Relation Matrix . . . . .	79
5.8	A Model of the Software Design Life-cycle (MSDL) . . . . .	80
5.9	Definitions of the Problem Model Domain framework . . . . .	81
5.10	Definitions of the Solution Model Domain framework . . . . .	82
5.11	Definitions of the Documentation Domain framework . . . . .	83
5.12	The current version of MF . . . . .	86

5.13	A template for specification of a model of design methodology . . .	87
5.14	Definitions of the template components . . . . .	88
8.1	Static Structure of the CDM Process Program . . . . .	175

# List of Tables

2.1	Summary of the research goals and contributions . . . . .	18
4.1	Summary of the differences between JSD and BOOD . . . . .	54
6.1	<i>Concepts</i> classified within MCTH . . . . .	112
6.2	<i>Artifacts</i> classified under MCTH . . . . .	113
6.3	<i>Representations</i> classified within MCTH . . . . .	114
6.4	The JSD and BOOD <i>Actions</i> classified within MCTH . . . . .	115
6.5	Classification of <i>artifacts</i> under the Problem Model Domain . . . . .	116
6.6	Classification of the <i>artifacts</i> under the Solution Model Domain . . . . .	117
6.7	Classification of the artifacts under the Document Model Domain . . . . .	118
6.8	Summary of the differences between the RDM and BOOD components	124
6.9	Summary of the differences between the JSD and SD components . . . . .	133
6.10	Summary of the differences between SD and DSSD . . . . .	139
6.11	Summary of the differences between LCP and DSSD components . . . . .	143



# Acknowledgements

I am, of course, very deeply indebted to my advisor, Prof. Leon J. Osterweil. He has supported me, inspired me, taught me during my Ph.D study. He also has greatly encouraged me to pursue this topic. He often carefully reviews my work, revises my writing, and provides excellent advice on my research. Without his support and guidance, it would have been impossible for me to finish my dissertation.

I also wish to thank other ICS software faculty members, particularly, Professors Richard W. Selby and Richard N. Taylor. Their comments have been very useful to this research.

I am very grateful to Mr. Grady Booch and Mr. John Cameron for reviewing my comparisons between Booch's OOD and JSD. Their comments helped in improving this work and their support encouraged this research.

I am very grateful to Dr. Robert Balzer and Prof. Alain Lewis, who served on my Ph.D candidacy exam committee. Their comments and questions helped me in shaping the topics of this research.

I am grateful to the Institute of Electrical and Electronic Engineers, Inc. and to the IEEE Computer Society for permission to reprint sections from the following articles:

"Towards Systematic, Objective Design-Method Comparisons". *IEEE Software*, May, 1992. pp. 43-53, Xiping Song and Leon J. Osterweil. Portions of this paper appear in Chapters 1, 2, 5 and 6.

"Comparing Design Methodologies Through Process Modeling," *Proceedings of the First International Conference on the Software Process*, IEEE Computer Society, Redondo Beach, Ca., October 21-22, 1991, pp. 29-44. Xiping Song and Leon Osterweil. Portions of this paper appear in Chapters 1,2,3 and 4.

Finally, I would like to thank my sponsors. My research was supported by the Defense Advanced Research Projects Agency, through ARPA Order #6100,



Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. My 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.

**Xiping Song**

Department of Information and Computer Science  
University of California  
Irvine, California 92717-3425  
phone: (714) 856-0902 (Home)  
phone: (714) 856-4047 (Office)  
email: (Internet) song@ics.uci.edu

**Personal Data**

Home Address: 4825 Verano Place, Irvine, CA 92715

**Education**

Ph.D., University of California, Irvine, CA, Computer Science  
Advisor: Leon J. Osterweil, (expected May 1992)  
M.S., University of Colorado, Boulder, Co., Computer Science, May 1988  
M.S., Beijing Polytechnic University, Beijing, China,  
Computer Science, June 1984  
B.S., Beijing Polytechnic University, Beijing, China,  
Computer Science, June 1982

**Research Interests**

Software engineering process  
Software analysis and design methodologies  
Software environments  
Software engineering data base  
Software version and configuration management

**Academic Experience**

9/88-present: Research Assistant, University of California, Irvine, CA.  
6/87-9/88: Research Assistant, University of Colorado, Boulder, CO.  
9/85-6/87: Teaching Assistant, University of Colorado, Boulder, CO.  
9/82-8/85: Lecturer, Beijing Polytechnic University, Beijing, China.

**Industrial Experience**

6/88-9/88: Research Associate, Software Engineering Program, Sci.& Tech Div.,  
US WEST Advanced Technology, Boulder, CO.  
6/86-1/88: Software engineer, UniData, Denver, CO.

**Memberships in Professional Societies**

Association for Computing Machinery,  
ACM: SIGSOFT,  
Institute of Electrical and Electronics Engineers,  
IEEE Computer Society.

## Published Journal and Refereed Conference Papers

- Xiping Song and Leon J. Osterweil.  
“Towards Systematic, Objective Design-Method Comparisons,”  
IEEE Software, May, 1992. pp43-53.
- Xiping Song and Leon J. Osterweil.  
“A Process-Modeling Based Approach to Comparing and Integrating Software Design Methodologies,”  
To appear in *Proceedings of the Fifth International Workshop on CASE*.  
IEEE Computer Society, July 6, 1992, Montreal, Quebec, Canada
- Xiping Song and Leon J. Osterweil,  
“Comparing Design Methodologies Through Process Modeling,”  
*Proceedings of the First International Conference on the Software Process*,  
IEEE Computer Society, Redondo Beach, Ca., October 21-22, 1991, pp. 29-44.
- Stanley M. Sutton, Jr., Hadar Ziv, Dennis Heimbigner, Harry E. Yessayan, Mark Maybee, Leon J. Osterweil, and Xiping Song,  
“Programming a Software Requirement-Specification Process,”  
*Proceedings of the First International Conference on the Software Process*,  
IEEE Computer Society, Redondo Beach, Ca., October 21-22, 1991, pp. 68-89.
- Xiping Song and Leon J. Osterweil  
“A Framework for Classifying Parts of Software Design Methodologies”  
*Proceedings of the Second Irvine Software Symposium*, Richard W. Selby Eds.  
The Irvine Research Unit in Software, Irvine, Ca., March, 1992,

## Papers Submitted for Journal and Conference Publications

- Xiping Song and Leon J. Osterweil.  
“CDM: A Process-Modeling Based Approach for Comparing Design Methodologies”.  
*IEEE Transactions on Software Engineering*.

## **Presentations**

- “Comparing Design Methodologies Through Process Modeling”  
*The First International Conference on the Software Process*, IEEE Computer Society, Redondo Beach, Ca., October 21-22, 1991, pp. 68-89.
- “A Framework for Classifying Parts of Software Design Methodologies”  
The Second Irvine Software Symposium, Irvine, Ca., March, 1992,

## **Selected Technical Reports**

- “Version Control and Configuration Management System”  
Xiping Song  
Technical report, US West Advanced Technologies, 1988.
- “DEBUS: A Software Design Process Program”  
Xiping Song and Leon Osterweil.  
Arcadia technical report, UCI-89-02, 1989.
- “A Survey of Process Program Design Formalisms”  
Xiping Song and Leon Osterweil.  
Arcadia technical report, UCI-90-16, 1990.
- “REBUS: A Requirement Specification Process Program”.  
Xiping Song, Mark Maybee, Leon Osterweil and Dennis Heimbigner  
Technical report, ICS, Univ. of California Irvine. UCI-90-17.
- “An Experiment on CDM—A Method for Comparing and Choosing Design Method”  
Xiping Song and Leon Osterweil.  
Arcadia technical report, UCI-91-11, 1991.



# Abstract of the Dissertation

## Comparing Software Design Methodologies Through Process Modeling

by

Xiping Song

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1992

Professor Leon J. Osterweil, Chair

Recently, the importance of consolidating existing software engineering approaches and concepts has been well recognized by the software engineering community [Boa90]. We believe that study of Software Design Methodologies (SDMs) is an excellent place to start. To achieve this, we must be able to objectively and systematically compare SDMs.

Quite a number of SDMs have been developed and compared over the past two decades. An accurate comparison aids in codifying, enhancing and integrating SDMs. However, after analyzing the existing comparisons, we found that these comparisons are often based largely upon the experiences of the practitioners and the intuitive understandings of the authors. Consequently, these comparisons are subjective and affected by application domains. We also analyzed a number of comparisons which use quasi-formal approaches to comparing SDMs. We found that these comparisons are often based on hypothesizing features required by the design process and software design problems. In order to compare SDMs more scientifically, in this thesis we introduce a systematic approach (called CDM (Comparing Design Methodologies)) to objectively comparing SDMs. We hope that using CDM will lead to precise, explicit and complete comparisons.

CDM is based on modeling SDMs and classifying their components (e.g., guidelines and notations). Modeling SDMs entails decomposing them into components. The classification of the components illustrates which components address similar design issues and/or have similar structures. Similar components then may be further modeled to aid in understanding more precisely their similarities and differences. The models of the SDMs are also used as the bases for conjectures and conclusions about the differences between the SDMs.

Two key components required by CDM are 1) a fair *Base Framework* (BF) to classify parts of SDMs and a comprehensive *Modeling Formalism* (MF) to model all these parts. In this thesis we address these two problems by suggesting an evolutionary strategy for developing such a BF and MF. Then we present the BF and MF we have developed using this strategy, and demonstrate how they have been and can be used. Further we evaluate the BF and MF based on their applications and suggest how they might be enhanced. In doing this, we intend to illustrate that increasingly fair BFs and MFs can be developed by using this development strategy.

We believe that this sort of iterative evolutionary development of key framework and modeling formalisms is consistent with the ways in which more mature scientific disciplines operate. Thus, we hope that this effort indicates a way in which software engineering can begin to grow into a mature scientific discipline. Further, we suggest that this evolutionary development of BFs and MFs should be a community-wide activity.

In this thesis we demonstrate this approach by using it to compare six SDMs (JSD [Jac83], Booch's Object Oriented Design (BOOD) [Boo86], RDM [PC86], SD [YC79, SMC74], LCP [War76], and DSSD [Orr77]). We compared our SDM comparisons against other comparisons obtained using other approaches. The results of this comparison demonstrate that *process modeling* [Ost87, KH88] is valuable as a powerful tool in analysis of software development approaches. Besides, the SDM comparisons result, we obtained through this effort are by themselves valuable for understanding software design activities and SDMs.

# Chapter 1

## Introduction

### 1.1 Comparing Software Design Methodologies

Effectively designing large-scale and complex, yet reliable and high performance, software has motivated research into a more systematic approach to designing software systems. Such an approach, usually called a *software design methodology* (SDM), describes and justifies a collection of *design methods*. A design method assists designers by providing the rules specifying 1) what design decisions are to be made, 2) how to make and denote/organize them, and 3) in what order they should be made. The SDM chooses its methods to complement one another, along with rules for applying them. *Method components* are parts of a design method. Concepts, artifacts, measures, guidelines, rules-of-thumb, notations and procedures are examples of method components.

Various SDMs (e.g. JSD [Jac83], OOD [Boo86], SD [YC79], RDM [PC86]), describe, at least superficially, different approaches for designing software. For example, Freeman [Fre83] has identified five kinds of design methods used in SDMs—top-down design, outside-in design, inside-out design, bottom-up design



and most-critical-component-first design. Thus, questions such as the following could be asked:

- What different design issues do they address?
- Which of their components are actually aimed at similar design issues?
- What are the relations among those method components?
- Is there some way to merge them, integrating the best characteristics of each?

Objectively answering those questions should help in

- enhancing our understanding of existing SDMs by discovering their weaknesses and false assumptions, (as Cameron notes [CCW91], “the commonly accepted differences may in fact not be real, and the real differences may be quite different from the apparent differences.”)
- devising a structure for recording SDM knowledge by discovering SDM’s common characteristics, (e.g., a generic structure for representing a variety of object-oriented design methodologies).
- evaluating the SDMs by discovering their differences, (e.g., with an understanding of the differences, one might be able to identify the application domains for which an SDM is suitable), and
- integrating the SDMs by discovering compatibility between SDMs, (e.g., being able to merge design specifications which are specified in different SDMs by different organizations and design teams [CCW91]).

In the last two decades, a fair number of attempts have been made to answer such questions. In the next two sections, we describe those attempts and analyze their limitations.

## 1.2 Informal Comparisons

Some of those attempts [Ber81, Ber78, Was80, YT86, PT77, Gri78] used an informal approach to compare SDMs.

The paper [Ber78] summarizes concepts used in structural analysis and design. Then, it briefly describes a number of SDMs (i.e., functional decomposition, data flow design (the Yourdon method), and data structure design (JSP)). For each SDM, it describes experiences in using the SDM, and lists its advantages and disadvantages. Finally, it presents a design example to show the differences between the Yourdon method and JSP.

The paper [Was80] first describes a software development life cycle and a set of underlying concepts (e.g., modularization, abstraction) used in SDMs. Then, it summarizes a number of SDMs (e.g., SADT, SREM, USE, JSP, DREAM). For each SDM, it describes some of its advantages, disadvantages and application domains.

The papers [YT86, PT77, Gri78] use a similar strategy to compare SDMs. The paper [YT86] emphasizes comparing distributed software design methodologies. The paper [PT77] provides fairly complete summaries of the features provided by various SDMs (e.g., Yourdon, JSD, LCP) and the supports provided by their advocates. The paper [Gri78] emphasizes comparing the underlying concepts used by the SDMs.

By analyzing these comparisons, we found that they are usually aimed at helping software practitioners 1) to intuitively understand a number of SDMs, 2) to roughly understand the strengths/weakness of each SDM, and 3) to choose an SDM for designing a software system. Consequently, these comparisons often rely

on 1) the subjective experiences of the practitioners, 2) the intuitions of the authors who make the comparisons, and 3) informal analysis of the SDMs.

Based on the experiences of practitioners, some of these comparisons [Ber78, PT77] describe how widely and successfully a methodology has been utilized in real life software projects. For example, Bergland [Ber78] states: "I have seen several success stories which praise the ease of doing Data Flow Design but they also pointed out the high overhead associated with passing all that data from one 'ear' to the other 'ear' of their structure diagram." However, Peters and Tripp comment: "The process (of Data Flow Design) seems deceptively simple but when attempts are made to use it, difficulties are encountered. For example, consistently identifying transformations of data is not easy to do." From this example, we can see that these assessments on data flow design seem to be contradictory. Why?

The reason, we believe, is that these comparisons are affected largely by differences in application domains and project personnel. In addition, this kind of comparison does not show precisely *why* a part of the methodology is praised or criticized. As a result, it is difficult to evaluate such comparisons.

The levels of understanding of different authors also vary; different authors may have a better understanding of those method components that appear to be more important to them. Consequently, different authors may emphasize different method components. Thus, many comparisons between the methodologies tend to be incomplete and biased. An author often, based on his/her understanding, expects a comparison result and then deliberately selects some way to show it. Thus, sometimes, one design methodology is shown to be more appropriate for designing a given software example than another methodology. For example, the McDonald's

Frozen Food Warehouse example [Ber78, Jac75] is used to show that JSP is better than SD (Structured Design). However, Peters [PT77], in giving another example, points out that the assumptions of JSP could be invalid at times. Thus, these efforts fail to systematically and completely compare design methodologies. Consequently, the comparison results often vary from author to author.

Most unfortunately, in these comparisons a framework for classifying design issues and a type hierarchy for characterizing method components are lacking. Therefore, method components are not systematically organized, typed, and classified by the design issues they address. As a result, these comparisons often fail to show *how* and *why* some components *should/could* be compared and thus the completeness of these comparisons often can not be evaluated.

In summary, previous SDM comparison efforts are inadequate because they:

- are affected by project personnel and application domains.
- fail to show an explicit and formal basis for drawing a conclusion.
- are difficult to be evaluated independently by others.
- are often are not precisely and explicitly described.

### 1.3 Quasi-formal Comparison Approaches

Some other comparisons use quasi-formal approaches to compare SDMs. These comparisons are aimed at helping software practitioners as well as software researchers to 1) understand the substance of the general software design activity and or a particular SDM, 2) to more precisely and comprehensively understand

the strengths/weaknesses of each SDM. Quasi-formal comparison approaches (e.g., those used by [WFP83a, Kun83, Bra83, Oli83, ABC<sup>+</sup>91]) can be divided, as Sol [Sol83] suggests, into five categories. Sol describes these approach categories as:

1. One may describe an idealized methodology and evaluate other methodologies against this frame of reference. *Then the problem remains how to develop such an ideal.*
2. Another approach is to distill a set of important features in an inductive way from a number of methodologies. The methodologies can be compared against this yardstick. *Evaluation depends very heavily on the subjectivity in scoring the various methodologies against the framework and on the relative weight given to a feature.*
3. A third approach is to formulate a-priori hypotheses on a (partial) ordering of features, and to try to derive a possible framework from the empirical evidence in a number of methodologies. *The difficulty in this approach lies primarily in the formulation of hypotheses.*
4. Quite another approach is to define a meta-language as a vehicle for communication and as a frame of reference in which various methodologies can be described. The attractiveness of this approach is that implicit, contextual features as well as the process aspects of a methodology can be made explicit. *However a meta-language may have a limited expressive power. It also may blind us for specific features of some methodologies.*
5. Finally, a contingency approach tries to relate features of methodologies to contingencies in applying this methodology in specific problem situations.

The paper [WFP83a] surveys a large number of SDMs and evaluates these SDMs against a number of frameworks. This paper defines a model of the software development life cycle and uses it to examine the coverage of these SDMs. This paper also evaluates these SDMs from other aspects, e.g., technical concepts supported, methodology applicability, etc. However, the evaluation largely depends on the claims of the authors of the SDMs. This paper primarily takes comparison approach 1.

The paper [Oli83] describes a framework that consists of five levels of abstraction: external, conceptual, logical, architectural and physical. This paper uses certain templates as a sort of meta-language to describe the SDMs and thus to aid the analysis of the SDMs. This paper uses, to some extent, comparison approaches 2 and 4.

The paper [Kun83] derives a set of features from analyzing the objectives of the SDMs. This paper examines SDMs from the following aspects: 1) understandability, 2) expressiveness, 3) processing independence 4) checkability and 5) changeability. Then the author analyzes three different modeling approaches against these five aspects.

The paper [Bra83] analyzes a number of SDM comparisons and, based on its analysis, suggests a very high level framework for comparing SDMs. The items in the framework are selected according to the author's understanding of the SDMs and experiences in comparing SDMs. Those items are 1) origin and experiences, 2) development process, 3) model, 4) iteration and tests, 5) representation means, 6) documentation, 7) user orientation and 8) tools and prospectives. This paper primarily takes comparison approach 2.

The paper [ABC<sup>+</sup>91] analyzes a variety of object-oriented SDMs from three points of view—1) concepts, 2) notations, and 3) processes. It summarizes the object-oriented SDMs from these points of view, listing their key concepts, notations and processes. Then, based on this summary, it compares the SDMs. This comparison uses, to some extent, comparison approach 4.

In analyzing these comparisons, we found that they are usually based on 1) firm and well-known understandings of widely recognized method components (e.g., data flow diagrams), 2) what the authors claim for their SDMs (e.g., by using a questionnaire [WFP83b]), and 3) understandings of the authors of the comparisons (e.g., applying an SDM to an example and analyzing the application). Using method 1 can often lead to a fairly objective comparison because the method components are usually well understood. However, using methods 2 and 3 may lead to some controversial results (e.g., Does JSD [Jac83] provide better guidance for identifying an entity than BOOD [Boo86] does for identifying objects, as concluded in [BC91]?). We believe that one reason for this is that comparisons made in these two methods do not rest upon an explicit and formal basis (as does, for example, a proof of a theorem in mathematics) that enables independent evaluation of the comparisons themselves (e.g., evaluation of the completeness of the comparisons).

Besides, Approaches 1, 3 and 5 rely on formulated hypotheses (e.g., Approach 5 (e.g., [Wie91]) relies on hypothesizing a problem situation) rather than an analysis of existing SDMs. This could hinder one from objectively and systematically comparing SDMs. Although approaches 2 and 4 do not rely on formulating hypotheses, they have their own problems which need to be coped with (e.g.,

they need to have a comprehensive specification language). Thus, these comparisons which use quasi-formal approaches still achieve only limited objectivity in comparing SDMs.

## 1.4 Motivations for Objective Comparison

We believe that these previous comparisons all have value. They can help, at least to some extent, software practitioners to learn, choose and use SDMs, and software researchers to deepen their understanding of SDMs as well. However, two growing interests in the software engineering community are motivating work on more objective and more systematic comparisons.

The first interest is aimed at consolidating software engineering concepts and approaches. Pointing out that current software engineering approaches are often “slippery and many-sided”, the report of the recent US National Research Council’s Computer Science Technology Board (CSTB) workshop [Boa90] concludes: “... *progress will be made if the vast array of existing and emerging knowledge can be codified, unified, distributed, and extended more systematically*”. To achieve this in the area of SDM study, we must seek objective and systematic comparisons of SDMs. Otherwise, the codification and unification of SDMs, which will rely on the comparisons, would be less likely to be recognized and thus rarely used. Moreover, such codification and unification will not be effective for making progress in software engineering.

The second interest is aimed at developing process-centered software design environments [TBC<sup>+</sup>88, SO89]. As such environments are often aimed at strongly



supporting software designers in using various SDMs, its development will probably require developers to have a more precise and objective understanding of similarities and differences between SDMs. A precise and objective comparison is expected to help in building a software development environment that is most effective in supporting software designers who are using those SDMs. Such comparisons will probably also help the effective integration of SDMs and their support tools.

## 1.5 Strategies for Objective Comparison

In pursuit of objective and systematic comparison of SDMs, we first observed how such comparisons are made in other scientific disciplines. Analysis and comparison are activities at the heart of most scientific fields, including biology and chemistry. In biology, animals are systematically and objectively studied using comparisons of their organs and their inter-organ relations. Usually, organs (e.g., eyes) are classified by their functions (e.g., vision). From such classification, organs (e.g., eyes of various animals) having the same or similar functions can be identified and then compared. To study the differences in how they achieve these functions, one compares their structures (e.g., shape) and their relations to other organs (e.g., the brain). For a detailed comparison of certain organs, one can expect to need to identify and study the parts of these organs.

We believe that an objective and scientific comparison of SDMs should similarly be based on comparisons among method components and inter-component relations. Components should be classified by their functions (what problems they

address) and characterized by their structures. However, SDMs and their components often are not explicitly and rigorously defined, and are much less precisely understood than animals and their organs. Thus, it is desirable to model these SDMs in such a way as to make sure that their components are more explicitly and rigorously defined. Further, this certainly requires having modeling techniques, modeling formalisms and a set of strategies about how to apply these techniques and formalisms.

In searching for such techniques and formalisms, we looked into *process modeling* [Ost87, KH88, SO91], a research area that studies software process, (i.e., those activities, such as SDM-guided design activities, involved in software development and maintenance). Its general strategy is to formally specify software processes with the aid of more classical software specification techniques (e.g., data flow diagrams and regular expressions) in the hope of understanding these processes better. A number of Software Process Modeling Formalisms (SPMFs), (e.g., HFSP [Kat89] and SDA [Wil88]), a set of conventions for specifying software processes, have also been developed in this area to more rigorously and explicitly model software processes. As *process modeling* supports the rigorous and explicit descriptions of static software processes and structures of their components, we think that process modeling should help us cope with this problem.

We also believe that it is well founded to use process modeling techniques to support modeling of SDMs based on the following observations. A *design process* is a type of software development activity that adapts an SDM in response to local factors, and uses it to devise software artifacts that are to satisfy specific software requirements. Thus, a design process can perhaps be viewed as an execution of an instantiation of an SDM. Conversely, an SDM can perhaps be viewed as a generic

and static process definition<sup>1</sup>. Thus, as an SPMF supports the modeling of software process definitions, it should be plausible to use process modeling techniques to model SDMs.

Based on these motivations, observations and foundations, we attempt in this research to use process modeling techniques to pursue objective and systematic comparison of SDMs. In the next chapter, we will more precisely define our research goals.

---

<sup>1</sup>Moreover, when formalized in a programming language, an SDM would then be a *process program* [Ost87, SHO90]

# Chapter 2

## Problem Definitions and Research Goals

In this research we will primarily tackle only one problem. This problem, as discussed in the last chapter, is about how to apply process modeling techniques to pursue objective and systematic comparisons of SDMs. Thus, the topic of this thesis is to study how to compare SDMs more scientifically, rigorously and precisely by using process modeling techniques. This research also has some subgoals and is aimed at making a number of different contributions to the software engineering community. In the following sections, we will define the problem and the research goals in detail.

### 2.1 The Primary Goal of the Research

In this research, we pursue objective comparisons among SDMs. Such comparisons should have the following characteristics:

- SDMs should be completely compared at a certain abstract level. No feature of an SDM being compared is to be hidden in order to favor it over another SDM against which it is being compared.
- All comparison results should have a formal and explicit basis, which describes why and how the result is arrived at.
- All comparison results should rely on the analysis of the SDMs. The analysis should (if possible) use well-recognized analysis techniques.

In this research, we also pursue systematic comparisons of SDMs. Such comparisons should have the following characteristics:

- The comparison process should be systematic. The process should systematically apply principles, guidelines, notations, etc. By this, we hope that subjectivity in comparing SDMs can be reduced.
- The comparison results should be organized systematically. They should be represented in rigorous and well-organized notations.

In this research, we also pursue more precise and explicit comparisons of SDMs. Such comparisons may more precisely answer such questions as 1) what components of various SDMs fall into the same class? 2) what are detailed differences among the components in a single class? and 3) what are the relations among the components?

To tackle these problems, we will define and study an approach to comparing SDMs. The approach will apply process modeling techniques and can help us to be able to make comparisons having the characteristics described above. In this research, we will lay down a solid foundation for this approach. This will allow

this approach to be further developed into a complete and comprehensive SDM comparison methodology. We will also describe its major steps, artifacts and the representation means it will employ.

## 2.2 Secondary Goals of the Research

In addition to the primary goal, this research also has some secondary goals.

### 2.2.1 Development of a Framework and Modeling Formalism

In this research, we will develop 1) a framework for classifying parts of SDMs and 2) a formalism for modeling SDMs. In this thesis, we often refer to the framework as *Base\_Framework* (BF) and the formalism as *Modeling\_Formalism* (MF). Those two will be used as the key facilities to support our comparison approach.

In recent years, the importance of developing frameworks for classifying software engineering knowledge and vehicles for specifying this knowledge has been well recognized by the software engineering community. Pointing out that current software engineering approaches are often “slippery and many-sided”, the report of the recent US National Research Council’s Computer Science Technology Board (CSTB) workshop [Boa90] concludes: “... *progress will be made if the vast array of existing and emerging knowledge can be codified, unified, distributed, and extended more systematically*”. To achieve this goal, the report suggests: “ *What is needed is a way to define and discuss the ‘parts’ of software engineering, the specification of each, and a conceptual framework within which to place them*”.

We strongly agree with these statements. Further, we believe that study of *Software Design Methodologies* (SDMs) is an excellent example of an area in which the above suggestion should be carried out. Existing SDMs (e.g., [Jac83, Boo86, Orr77, PC86]) are defined informally in plain English and their components are often not explicitly and rigorously defined. A framework for classifying these method components and a vehicle for specifying the components are still lacking. Thus, achieving those two research goals will probably contribute here.

In addition, we will study how to develop BF and MF, describing strategies for developing the BF and MF. We believe that it is very important to make the BF and MF development process explicit because it can be used to guide our research activities and can probably be adopted to develop frameworks and formalisms for studying other software engineering approaches (e.g., requirement engineering methodologies, various evaluation methodologies).

### **2.2.2 Evaluation of Software Process Modeling Formalisms**

In this research, we will also explore the value of *process modeling* [Ost87, KH88], a research area that studies software process, (i.e., those activities, such as methodology guided design activities, involved in software development and maintenance). Its general strategy is to formally specify software processes with the aid of more classical software specification techniques in the hope of understanding these processes better. Researchers in this area have developed a number of Software Process Modeling Formalisms (SPMFs), (e.g., HFSP [Kat89] and SDA [Wil88]), sets of conventions for specifying software processes. However, few

of them have seen more than limited application, and thus they have not been thoroughly evaluated. Besides, most existing experiments in using process modeling techniques have focused on development of process-centered software development environments. Thus, the application area of process modeling is still far from being thoroughly explored.

In this effort, we will adopt SPMFs as modeling formalisms to model SDMs and use their models for the SDM comparisons. Thus, this provides an excellent chance to further evaluate these existing SPMFs and widen the application area of process modeling.

We expect that the evaluation of process modeling will be carried out through studying the answers to the following questions:

- What SDM aspects can an existing SPMF help to characterize explicitly?
- What SDM aspects can an existing SPMF not help to characterize explicitly?
- What benefits can we gain from these explicit characterizations?
- How can effective process modeling be a help in comparing SDMs?
- What are the limitations of process modeling techniques in comparing SDMs?
- What aspects of SDMs do we desire to model, that are not supported by existing SPMFs. How should SPMFs be enhanced to support the modeling of these aspects?

### **2.2.3 Comparisons of Software Design Methodologies**

In this research we will compare a number of SDMs as an experiment in validating our comparison approach. This experiment will produce comparisons



Summary of Goals and Contributions		
Goals	Contributions	
	Area	Brief description
1. Developing an SPM-based comparison approach	SDM	Help to choose, evaluate enhance SDMs, etc.
1.1. Developing a BF	SDM	Systematically codifying, distributing SDMs.
1.2. Developing a MF	SDM	Define and understand SDM.
1.3. How to develop the BF and MF	SDM SE	Define an approach for consolidating SDM and SE.
1.4. Evaluating MF	SPM	Improving SPMF.
1.5. Comparing SDMs	SDM	Understanding and integrating certain SDMs.

Table 2.1: Summary of the research goals and contributions

among SDMs. We believe that these comparisons will be more explicit, precise and objective than previous SDM comparisons, and should help in understanding the SDMs and the software design activity in general. More specifically, this research may help in understanding composition of the existing SDMs and strategies they have used to guide software design activities. This may show what techniques have been repeatedly used in different SDMs, however to tackle different design problems.

Table 2.1 summarizes our research goals and planned contributions.

In the next chapters, we will describe our approach for comparing SDMs. We expect that developing, using and evaluating this approach can help us to tackle the above-described problems.



# Chapter 3

## Our Comparison

### Approach: CDM

In this chapter, we present a comparison approach we use to compare SDMs. We expect that the development, use and study of this approach will serve as a vehicle for us to achieve our research goals. The initial development of the approach will help us to formulate our research hypotheses. The use of the approach will help to evaluate the approach and will possibly produce the desired comparison results. The study of the approach will help to evaluate process modeling techniques in aiding comparisons of SDMs.

As process modeling supports the rigorous and explicit descriptions of static software processes and the structures of their components, this approach starts by modeling SDMs. With explicitly defined SDM models, we are then able to identify, classify, and compare method components. The data flow diagram in Figure 3.1 is a model of *Compare\_Design\_Methodologies*(CDM), the comparison process we suggest to compare two SDMs. A box in the figure denotes a data object used in CDM and an ellipse denotes a step of CDM. The label attached to a directed edge

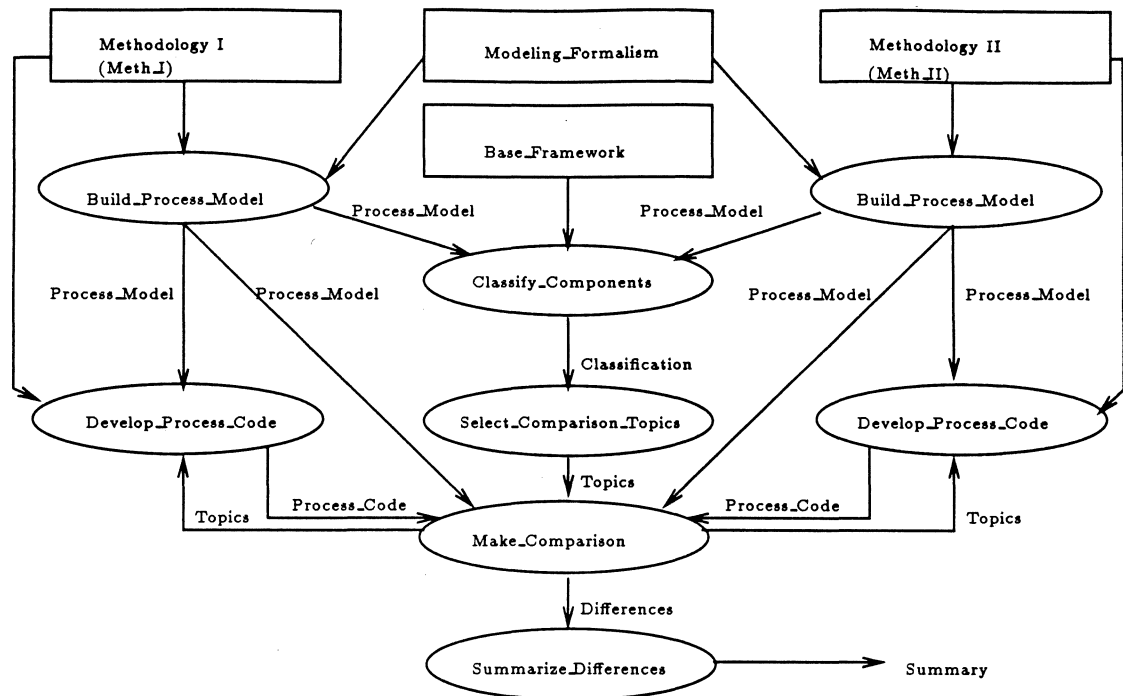


Figure 3.1: *Compare\_Design\_Methodologies(CDM)*, a data flow diagram modeling a SDM comparison process

shows the data flow between the steps. In the following sections we will describe those steps and the data flowing through them.

### 3.1 Step 1: *Build\_Process\_Model*

#### 3.1.1 Objectives

The first step in CDM (Fig. 3.1) is to develop a model, a more formalized description, of each of the two SDMs to be compared—*Meth\_I* and *Meth\_II*. Thus, this step produces two artifacts—the process models of *Meth\_I* and *Meth\_II*. We hope that in doing this we can effectively decompose an SDM into components,

which include design artifacts (design objects and inter-object connections), activities (the order of execution of design steps and their inputs and outputs). We also hope that in doing this, we can make the types of these components and their relations more explicit. Those types and relations will be used to guide the comparisons and aid the analysis of SDMs.

Because an SDM can be very complex, it is important that its model at higher abstraction levels be compact and clear, yet complete enough for further refinement. Since in this step it has not been decided which method components to compare, it is desirable to avoid specifying details that might be irrelevant to future comparisons. Thus, this step is analogous to the modeling and design of a software system, which emphasize the development of the architecture of the system.

As we indicated earlier that an SDM can be viewed as a process definition, it is plausible to apply an SPMF as a *Modeling\_Formalism* to model the SDM. The SPMF chosen must be capable of modeling the characteristics matching those described in the last two paragraphs.

### 3.1.2 Issues

We have identified three important issues that are related either to the feasibility or to the applicability of CDM. If CDM is to be used applied successfully, one must understand and address these issues.

**Issue 1:** One problem that may be encountered in modeling an SDM is that an SDM probably contains some components (e.g., measures, guidelines and rules-of-thumb) that lack precise semantics and therefore cannot be modeled precisely. Some of those components could be defined more precisely and rigorously along with the improvement of the SDMs. Some of them, however, as Cameron [CCW91] noted, inherently cannot be defined more precisely. These components often are what will allow flexibility in using the SDM.

Our strategy in coping with this is to completely model the SDMs at higher abstraction levels. By doing this, we can at least highlight all the key components in the model. Then, from such a model, and given informal descriptions, we can identify those components and aspects that might be more amenable to detailed and lower-level precise modeling with existing SPMFs. For example, at a higher abstraction level, we can model that an SDM has certain principles and some design activity uses some of these design principles. Then, at a lower abstraction level, we can examine whether the principles can be effectively modeled.

We believe that the comparisons, which are based on partial but rigorous models and complementary informal descriptions, should still be more precise and explicit than comparisons based solely on informal descriptions of the SDMs. Moreover, we anticipate that, with further development of SDMs and SPMFs, more components will be amenable to increasingly precise modeling. This belief is the foundation for our strategy.

**Issue 2:** Another problem is how to ensure that the *Modeling Formalism* is comprehensive enough to capture SDMs adequately. Usually, any given SPMF will be more capable of precisely and effectively specifying certain aspects of a

given software activity than others would. Thus, a comparison based on models specified in one particular SPMF will be more effective in showing differences in certain aspects, but may be relatively less effective in other aspects. Thus, any arbitrarily selected SPMF will help in making certain limited comparisons, but it should be expected that these comparisons may be incomplete and potentially misleading.

For example, a model specified by a functional SPMF (e.g., HFSP [Kat89]) may more clearly indicate the input/output domains of a design activity than would a rule-based SPMF (e.g., Marvel [KF87]), which could be more capable of modeling the design criteria. (e.g., criteria for selecting an entity in JSD). Therefore, a comparison based on models in HFSP would show the differences in the input/output domains of the design activities rather than the differences in the design criteria. Conversely, in a rule-based paradigm, the differences in the criteria, rather than in the input/output domains, would be shown clearly. Thus, overall SDM comparisons, which are made based on models specified in only one formalism (e.g., HFSP or Marvel), should be expected to be at least somewhat misleading.

These observations indicate why it is desirable to use a number of SPMFs, chosen to complement each other. Doing so should help to more completely model an SDM and therefore reduce the chance of obtaining a misleading comparison. Thus, this research will address the issue of finding appropriately complementary SPMFs, which will be discussed in chapter 5.

**Issue 3:** Quite another issue is how to validate a model of an SDM (i.e., examine whether the model is an accurate characterization of the SDM at least



with respect to the aspects being modeled). We suggest that two strategies can be used: 1) reviewing the model against definitive publications describing the SDM, and 2) soliciting comments from the authors of the SDM. By doing the first, we hope to ensure that the model captures the SDM as presented by the publications. By doing the second, we hope to eliminate the modeler's possible misunderstandings of the SDM as presented by the publications. Thus, we expect that a model validated using both these strategies should be sufficiently accurate and thus can ensure that the comparison is accurate. Note that, since our work focuses not on evaluating SDMs but rather on differentiating among SDMs, it does not seem necessary to use an experimental approach to validate SDM models.

## 3.2 Step 2: *Classify\_Components*

### 3.2.1 Objectives

Having identified the method components, one next considers classifying the components (see Fig. 3.1) within a comparison framework (*Base\_Framework* of Fig. 3.1). Such a classification is used to identify the overall differences/similarities between SDMs, and to guide the selection of comparison topics. Therefore, the classification under *Base\_Framework* should show which method components address the same or similar issues. Further, it should show *how* and *why* certain components *should/could* be compared.

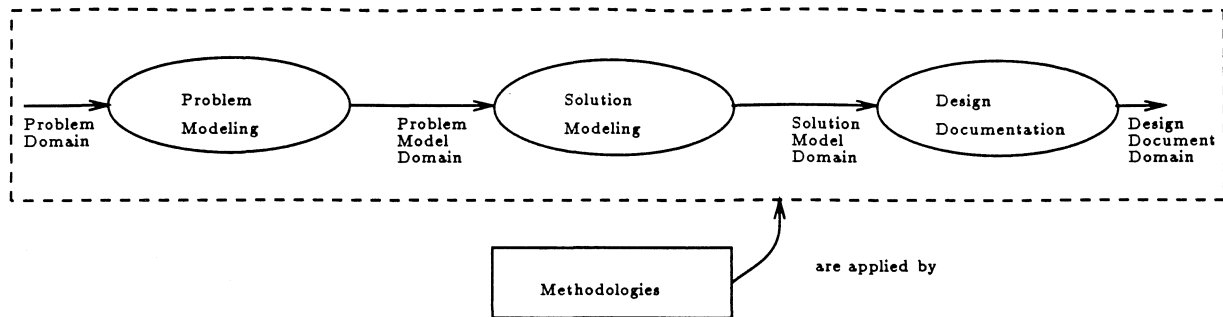


Figure 3.2: Part I of the *Base\_Framework*: A Model of the Software Design Life-cycle (MSDL). (The diagram inside the broken lined box is a data flow diagram.)

### 3.2.2 A Prototype Framework

In our research we have identified and utilized a prototype framework that consists of two parts: a Model of the Software Design Life-cycle (MSDL) and a Method Component Type Hierarchy (MCTH). MS DL enables us to functionally classify components (i.e., classify components by the issues they address) while MCTH enables us to characterize the structure of the method components.

MSDL, as Fig. 3.2 shows, consists of three sub-processes which, by applying design methods, transform the elements from one domain to another. These sub-processes can be further decomposed. Thus, for example, the Solution Model Domain is decomposed in Fig 4.6. MCTH, whose top-level types are defined in Fig. 3.3, provides the basic types that can be used to characterize the parts of these methods. The description of the framework presented here is used only to ease the description of CDM. A more comprehensive version of the framework will be described in Section 5.4.2.

- **Concept:** includes 1) understanding of the general characteristics of software problem domains and the problems in designing software; 2) general principles for coping with these problems; 3) concrete strategies or criteria that guide the design of software and that cope with these problems.
- **Artifact:** a description involved in the design process. The structure and role of an artifact in a methodology are probably affected by the related *concept*. An artifact could be represented in one or a combination of a number of forms such as computer program, diagram or templated text.
- **Representation:** a means for representing design *artifacts*, (e.g., document templates and design/modeling languages). A representation should provide expressive notations with rigorous semantics to aid in specifying design *artifacts*.
- **Action:** one or more physical and/or mental behaviors used in design. An action may create or modify a design *artifact*.

Figure 3.3: Part II of the *Base\_Framework*: types at the top-level of Method Component Type Hierarchy(MCTH)

---

### 3.3 Comparison of Design Methodologies

In the previous sections, we have discussed the first two steps of our approach (see Fig. 3.1). In this section we discuss the last three steps—Select Comparison Topics, Develop Process Code and Make Comparison.

#### 3.3.1 Step 3: *Select Comparison Topics*

##### Objectives

In this step, we will identify the method components to be compared. This selection of method components will guide succeeding comparisons.

##### Criteria and Guidelines

Generally, two criteria can be used in selecting components for comparisons: 1) they should be comparable, and 2) a comparison between them should help in showing key differences between SDMs.

Selecting components to be compared also requires guidelines for determining which components *can* be compared. Classifications should indicate which components address similar design issues and have comparable structures, (e.g., an *action* could be compared with another *action* but not with a *representation*), and should aid in selecting comparison topics. For example, based on the definition of MCTH (Fig. 3.3), we may use a guideline like the following to select topics for comparison:

The classification may illustrate that two *concepts* address similar design issues. If so, one can select these two *concepts* for comparison, and begin to trace the *artifacts* supporting the *concepts*, the *representations* representing the *artifacts* and the *actions* creating or modifying the *artifacts* (which are specified in process models) and eventually, find the *artifacts*, *representations* and *actions* that can be compared.

### 3.3.2 Step 4: *Make\_Comparison and Develop\_Process\_Code*

#### Objectives

After deciding to compare two given components, one compares the models expressing those two components in order to understand their concrete differences. Since a process model (which is analogous to an architectural design in our approach) must capture only an overall view of an SDM, it is not sufficient for identifying detailed differences between method components. Thus, it might be necessary at times to develop a model characterizing those details most relevant to the comparison to be made. These more detailed process models are similar to detailed designs and we refer to them as *process code* to distinguish them from those that capture the higher level view of an SDM.

#### Aspects to Compare

CDM might be more effective in helping with the comparison of certain aspects of SDMs than with the comparison of some other aspects. Based on some

criteria [RS78] for classifying SDMs and our experiences [SO89], we anticipated that comparing process code would aid in identification of differences in:

- **Inter-component dependency**, what other components a component depends upon, which may illustrate different usages and characteristics of the components;
- **Degree of human involvement**, the need for human intelligence in performing design actions, which may indicate how much the action could be automated or systematically applied in practice;
- **Development procedure**, the order in which the design actions are to be performed; and
- **Scope of issues**, the scope of the design issues the SDM addresses.

In the next chapter, we will present an experiment that uses CDM to compare SDMs.

### 3.3.3 Step 5: *Summarize Differences*

Summarizing differences identified is aimed at providing readers with an overview of the differences between the SDMs compared. This summary should be organized around the *Base\_Framework* and the comparison topics selected. For example, it should show what differences have been identified under the Problem Model Domain, etc. By doing this, the differences can be appropriately emphasized and therefore better understood. This summary should help in indicating the differences between the components with respect to the aspects (e.g., those described in the last section) with which the comparisons are concerned. This

<b>The MSDL component</b>			
<b>Component</b>	<b>Aspect 1</b>	<b>Aspect 2</b>	<b>...</b>
<i>A</i>	Comments	Comments	...
<i>B</i>	Comments	Comments	...

Figure 3.4: The artifact structure for summarizing the differences

summary should help directly in analyzing the functions of the method components, providing an aid in alternating the components in an SDM and integrating SDMs.

Fig. 3.4 defines an artifact structure we suggest for use in CDM to summarize the differences between SDMs. In the table of Fig. 3.4, *A* and *B* are two method components that have been compared, which have the same method component type, and address the same issues about **The MSDL component**. The **Aspects** are the aspects from which the comparison is being made (e.g., Inter-component dependency). The **Comments** then indicate the differences between *A* and *B* with respect to these aspects.

# Chapter 4

## Experiment 1: Comparing JSD With BOOD

In this chapter we describe an experiment that demonstrates how CDM was used to compare JSD [Jac83] and BOOD [Boo86].

The primary reasons for choosing JSD and BOOD are: 1) as we believe that they are neither dramatically different nor very similar, thus, their comparison should not be an extreme case, 2) it seems to us that they share many characteristics with many other SDMs (e.g., a variety of Object-Oriented SDMs [RBP<sup>+</sup>91, Jac87]).

First, we introduce the SPMF used to model JSD and BOOD, and discuss the reasons for using it. Second, we present a brief overview of JSD and BOOD with descriptions of their models. Third, we illustrate how their components can be classified under the *Base\_Framework*, and how similar components can be compared based on these models. Last, we evaluate the application of process modeling for comparing SDMs.



## 4.1 Introduction to HFSP

We specified models of JSD and BOOD in an SPMF called the Hierarchical and Functional Software Process (HFSP) formalism [Kat89]. In HFSP, a software process is modeled as an *activity* defined in the form:

$$A(x_1, x_2, \dots, x_n | y_1, \dots, y_m);$$

Execution of  $A$  is performed functionally, and does not refer to or change any global object.  $x_1, x_2, \dots, x_n, y_1, \dots, y_m$  are called *attributes* of  $A$ , defining the input and output domains. The activity  $A$  might be decomposed into some subactivities in the form:

$$A \Rightarrow A_1, A_2, \dots, A_k \text{ Where } E;$$

The set  $E$  of *attribute definitions* specifies how to prepare inputs to the subactivities and how to get the result of the main activity  $A$  when the subactivities  $A_j$  come up with their execution results.  $E$  contains the attribute definitions for:

1. input attributes of subactivities  $A_1, \dots, A_k$  and
2. output attributes of the main activity  $A$ .

Every attribute definition is of the form

$$a = f(a_1, a_2, \dots)$$

where  $a$  is the attribute to be defined,  $a_1, a_2, \dots$  are other attributes in the decomposition, and  $f$  is an auxiliary pre-defined function. These dependencies among the attributes determine the order in which the subactivities might be performed.

HFSP supports the modeling of a software process as a mathematical function. However, HFSP is less satisfactory in that it does not directly provide mechanisms for modeling the structures of artifacts and the conditions for activating/terminating a software development/maintenance activity or orchestrating its subactivities.

HFSP seems to be a plausible formalism to use to elucidate the characteristics of these SDMs (see the first two paragraphs of Section 3.1.1) because:

1. HFSP allows description of an SDM through a hierarchy of functional abstractions.
2. In HFSP, a design action can be rigorously defined as a function mapping some artifacts (i.e., input attributes) to other artifacts (i.e., output attributes). The dependency relations among the attributes of an activity and those of its subactivities can also be rigorously defined by defining the mappings between them. In HFSP, the input/output domains of the activity can be explicitly indicated at the time of both its definition and use.
3. The declarative property of HFSP should help us concentrate our attention on modeling the static properties (i.e., functions) of a design action. HFSP does not support description of the conditions under which the action will be activated or terminated. This should be a weakness in modeling dynamic characteristics of software processes. However, we believe that, in our application, this should be advantageous since this can help us in focusing on modeling SDMs rather than their instantiations/enactions.

Since HFSP is a powerful aid to modeling the functions of design action and design action is a basic type of method component, we focus on modeling the functions of

JSD and BOOD design actions. The weaknesses (e.g., weak support for modeling artifacts) of HFSP could affect our models and comparisons. We will discuss this issue after showing the comparison.

## 4.2 Use of CDM

### 4.2.1 Step 1: Build Process Models of JSD and BOOD

#### JSD and its Process Model

JSD consists of two major steps—*Develop\_Spec* and *Develop\_Impl*. In this overview we introduce only the former and correspondingly describe its model (Fig. 4.1).

As Jackson believes that the model of the *real-world* outside the system is more fundamental to the structure of the system than the required functions of the system, the first step of JSD, *Model\_Reality* (Fig. 4.1(e)), is to model the real-world in terms of *entities* and *actions*. An *entity* must exist *outside* the system, must perform or undergo *actions* in a significant *time-ordering*, and must be uniquely named. An *action* is regarded as taking place at a point of time, must take place in the world *outside* the system, and cannot be decomposed further into subactions. *Identify\_Entity\_Action*, (Fig. 4.1(e)(1)), provides a list of the *entities* and *actions*. Then, taking this list as input, *Draw\_Entity\_Structure*, (Fig. 4.1(e)(2)), specifies the life-cycle of each *entity*, called *Entity\_Structure* or *Real\_World\_Process*, as a regular expression [WG84] of action occurrences.

At the end of *ModelReality*, attention is shifted toward modeling the system. *ModelSystem* (Fig. 4.1(f)) has three sub-steps: 1) identify *ModelProcesses* that comprise the real-world/system interface and simulate *RealWorldProcess*; 2) describe how to *connect* (i.e., to communicate) a *RealWorldProcess* to a *ModelProcess*; and 3) derive the algorithmic structure of a *ModelProcess* from the action occurrences specified in the corresponding *RealWorldProcess*. The *connection* through which a *RealWorldProcess* communicates with a *ModelProcess* can be built by either of two mechanisms: a receiving *DataStream* or an inspecting *StateVector*. At the end, *ModelSystem* produces a document, *InitSysSpecDiagram*, describing the model of a system interface.

### **BOOD and Its Process Model**

BOOD is based on information-hiding and abstract data types. It emphasizes identifying and specifying the system component *objects* that *may* correspond to real-world components. An *object* must have *state*, *operations* it performs and undergoes, and restricted scopes for viewing other *objects* and for being viewed. The *state* of an *object* is defined by the value of the *object* plus its sub-objects. Therefore, an *object* is something that exists in time and space, and may be affected by the executions of the *operations* of other *objects*.

BOOD, as Figure 4.2 shows, consists of two types of actions: identifying the system components (i.e. *IdentifyObject* and *IdentifyOperations*) and specifying these components (i.e. *EstablishVisibility*, *EstablishInterface* and *EstablishImplementation*).

---

(a)  $JSD(Real\_World|Design\_Spec) \Rightarrow$

- (1)  $Develop\_Spec(Real\_World\_Desc|System\_Spec\_Diagram)$
- (2)  $Develop\_Impl(System\_Spec\_Diagram|System\_Impl\_Diagram)$
- (3) **Where**  $Real\_World\_Desc = Interview(Users, Developers, Real\_World),$
- (4)  $Design\_Spec = union(System\_Spec\_Diagram, System\_Impl\_Diagram);$

**Second\_Level:**

(b)  $Develop\_Spec(Real\_World\_Desc|System\_Spec\_Diagram) \Rightarrow$

- (1)  $Develop\_System\_Model(Real\_World\_Desc|Init\_System\_Spec\_Diagram)$
- (2)  $Develop\_System\_Func(Init\_System\_Spec\_Diagram|System\_Spec\_Diagram);$

**Third\_Level:**

(c)  $Develop\_System\_Model(Real\_World\_Desc|Init\_System\_Spec\_Diagram) \Rightarrow$

- (1)  $Model\_Reality(Real\_World\_Desc|Real\_World\_Model)$
- (2)  $Model\_System(Real\_World\_Model|Init\_System\_Spec\_Diagram);$

(d)  $Develop\_System\_Func(Init\_System\_Spec\_Diagram|System\_Spec\_Diagram) \Rightarrow$

- (1)  $Define\_Func(Init\_System\_Spec\_Diagram|System\_Function, Function\_Process)$
- (2)  $Define\_Timing(Init\_System\_Spec\_Diagram, System\_Function|Timing)$
- (3) **Where**  $System\_Spec\_Diagram =$   
 $is\_composed\_of(Init\_System\_Spec\_Diagram, System\_Function, Function\_Process, Timing);$

**Fourth\_Level:**

(e)  $Model\_Reality(Real\_World\_Desc|Real\_World\_Model) \Rightarrow$

- (1)  $Identify\_Entity\_Action(Real\_World\_Desc|Entity\_Action\_List)$
- (2)  $Draw\_Entity\_Structure(Entity\_Action\_List|Entity\_Structure\_List)$
- (3) **Where**  $Real\_World\_Model = is(Entity\_Structure\_List),$
- (4)  $Real\_World\_Process = is(Entity\_Structure);$

(f)  $Model\_System(Real\_World\_Model|Init\_System\_Spec\_Diagram) \Rightarrow$

- (1)  $Identify\_Model\_Process(Real\_World\_Model|M\_Proc\_Name\_List);$
- (2)  $Connect(Real\_World\_Model, M\_Proc\_Name\_List, Data\_Stream, State\_Vector|Connection\_List)$
- (3)  $Specify\_Model\_Process(Connection\_List, Real\_World\_Model, M\_Proc\_Name\_List|Model\_Process\_list)$
- (4) **Where**  $Init\_System\_Spec\_Diagram = is(Model\_Process\_List);$

Figure 4.1: A model of JSD specified in HFSP

---

We validated these two models by using both strategies we described earlier (Sec. 3.1.2). We extensively reviewed them against the definitive publications [Boo86] and [Jac83]. We also solicited comments from their authors (i.e., G. Booch and J. Cameron). J. Cameron thinks that our models are basically accurate while G. Booch indicates that our BOOD model was essentially accurate, except for its omission of two components: the timing diagram and the state transition diagram. Using their comments, we reviewed again the models against the publications on the SDMs. We found that Booch's comments are based on a more recent publication ([Boo91]). As our comparison is intended to be an experiment in using CDM rather than a definitive model of either SDM, we did not extend the model to cover these two components because we think that the current BOOD model is an accurate representation based on the original publication [Boo86].

As required by CDM (see Section 3.1.1), these two models highlight the *artifacts* (HFSP attributes) and describe the order for executing the *design actions* (HFSP activities). These models also completely and rigorously define the functions of the *design actions*. Although these models are still incomplete since they fail to model other types, (e.g., *representation* and *artifact*) of the method components and other aspects of the components, (e.g., what criteria a *design action* should apply), we think that they already convey enough information for us to make some comparisons and thereby to demonstrate/evaluate CDM.

#### **4.2.2 Step 2: Classify the Components of JSD and BOOD**

In this step, we first classify the components of each SDM and then merge these two classifications together. In doing this, we hope to show more clearly the

- 
- (a)  $\text{BOOD}(\text{Req\_Spec}|\text{Design\_Spec}) \Rightarrow$
- (1)  $\text{Identify\_Object}(\text{Req\_Spec}|\text{Objects, States})$
  - (2)  $\text{Identify\_Operations}(\text{Req\_Spec, Objects, States}|\text{Operation})$
  - (3)  $\text{Establish\_Visibility}(\text{Req\_Spec, Objects, States, Operation}|\text{Visibility})$
  - (4)  $\text{Establish\_Interface}(\text{Visibility, Objects, States, Operation}|\text{Interface})$
  - (5)  $\text{Establish\_Implementation}(\text{Interface}|\text{Implementation})$
  - (6) **Where**  $\text{Design\_Spec} = \text{is\_composed\_of}(\text{Interface, Implementation});$
- Second Level:**
- (b)  $\text{Identify\_Object}(\text{Req\_Spec}|\text{Objects, States}) \Rightarrow$
- (1)  $\text{Identify\_Nouns}(\text{Req\_Spec}|\text{Nouns})$
  - (2)  $\text{Identify\_Concrete\_Object}(\text{Req\_Spec, Nouns}|\text{Concrete\_Object})$
  - (3)  $\text{Identify\_Abstract\_Object}(\text{Req\_Spec, Nouns}|\text{Abstract\_Object})$
  - (4)  $\text{Identify\_Server}(\text{Req\_Spec, Nouns}|\text{Server})$
  - (5)  $\text{Identify\_Agent}(\text{Req\_Spec, Nouns}|\text{Agent})$
  - (6)  $\text{Identify\_Actor}(\text{Req\_Spec, Nouns}|\text{Actor})$
  - (7)  $\text{Identify\_Class}(\text{Req\_Spec, Agent, Server, Actor, Concrete\_Object, Abstract\_Object}|\text{Class})$
  - (8)  $\text{Identify\_Attributes}(\text{Objects}|\text{States})$
  - (9) **Where**  $\text{Objects} = \text{union}(\text{Concrete\_Object, Abstract\_Object, Class, Agent, Actor, Server})$
- (c)  $\text{Identify\_Operation}(\text{Req\_Spec, Object, States}|\text{Operation}) \Rightarrow$
- (1)  $\text{Identify\_Suffered}(\text{Req\_Spec, Object, States}|\text{Operation\_Suffered})$
  - (2)  $\text{Identify\_Required}(\text{Req\_Spec, Object, States}|\text{Operation\_Required})$
  - (3)  $\text{Defining\_Time\_Order}(\text{Req\_Spec, Operation}|\text{Time\_Order})$
  - (4)  $\text{Defining\_Space}(\text{Req\_Spec, Operation}|\text{Space})$
  - (5) **Where**  $\text{Operation} = \text{union}(\text{Operation\_Suffered, Operation\_Required})$
- (d)  $\text{Establish\_Visibility}(\text{Req\_Spec, Objects, States, Operation}|\text{Visibility}) \Rightarrow$
- (1)  $\text{Specify\_Object\_See}(\text{Objects}|\text{Objects\_See})$
  - (2)  $\text{Specify\_Object\_Seen}(\text{Objects}|\text{Object\_Seen})$
  - (3) **Where**  $\text{Visibility} = \text{union}(\text{Objects\_See, Object\_Seen})$
- (e)  $\text{Establish\_Interface}(\text{Visibility, Object, States, Operations}|\text{Subsystem, Interface}) \Rightarrow$
- (1)  $\text{Derive\_Module}(\text{Object}|\text{Module})$
  - (2)  $\text{Specify\_Attr}(\text{States, Module}|\text{Attributes})$
  - (3)  $\text{Specify\_Proc}(\text{Operations, Module}|\text{Procedures})$
  - (4)  $\text{Specify\_Visibility}(\text{Visibility, Module}|\text{Visibility\_Spec})$
  - (5) **Where**  $\text{Subsystem} = \text{is\_in\_term\_of}(\text{Module}),$
  - (6)  $\text{Interface} = \text{is\_composed\_of}(\text{Attributes, Procedure, Visibility\_Spec});$

Figure 4.2: A model of BOOD specified in HFSP

---

- 
- **Interface Model:** Describes the way in which real-world events interact with system components.
  - **Communication Model:** Describes the mechanism through which system components can communicate with each other.
  - **Data Model:** Describes the data structure used to realize the system.
  - **Entity Model:** Describes the system in terms of the system components and the operations they may perform and/or undergo. A system component must 1) exist in time and space, and 2) perform and/or undergo operations.
  - **Transform Model:** Describes how a desired system output can be computed. This may entail identification and elaboration of system programs.

Figure 4.3: Decomposition of the MSDL's component Solution Model Domain

---



potential of the classification, which indicates the intersected and complementary parts of the two SDMs.

Fig. 4.4(a) and (b) show the classifications under the Solution Model Domain described in Fig. 4.3. In these figures, an ellipse denotes a framework component while a box denotes a method component. The line connecting two components of the same kind denotes the *has-subclass* [KM85] relation, (e.g., in Fig. 4.4(b), the method component *Server* is connected with another method component *Object*. Thus, *Object* has *has-subclass* relation with *Server*, which means that *Server* defines a subset of the set defined by *Object* and therefore *Server inherits* the properties of *Object*). The line between two different kinds denotes an *is-addressed-by* relation which means that the method component addresses some issues raised by the framework component.

Through these figures, one can identify key method components, (they are usually at high levels of the *has-subclass* hierarchy), and thereby identify which components address which issues. For example, one can understand that *Server* as a subclass of *Object* should address some issues *Object* addresses—about modeling the Solution Model, more specifically, about modeling the Entity Model.

Fig. 4.4 (c) shows that *Model\_Process* addresses the issues concerned with modeling the Entity Model. In the following, we justify this to demonstrate how the rest of this functional classification can be similarly justified.

Fig. 4.1(f)(4) shows that

*Init\_System\_Spec\_Diagram = is\_in\_terms\_of(Model\_Process).*

This illustrates that model processes comprise the system interface and hence they are the components of the system. Since a model process simulates a JSD entity which must exist in the real-world and perform and/or undergo JSD actions, a model process should not be a mere input/output mapping and should perform and/or undergo operations. Therefore, it should be appropriate to view *Model\_Process* as addressing issues about modeling the Entity Model.

We also validated these classifications by using strategies similar to those used for validating the SDM models. Because the models and classifications have been carefully validated using CDM, we believe that they should not be viewed simply as our personal vision of the SDMs and their functions.

Having separately classified BOOD and JSD components, we merge these two classifications to identify which method components address similar issues. In doing so we provide guidelines for selecting components to be compared. Since the frameworks utilized are the same, we can do this by moving the method components of one classification into another, while keeping the *is\_addressed\_by* relations unchanged. By doing this, we get Fig. 4.4(c), which indicates that:

- BOOD does not explicitly address the issues of modeling the communication between system components (i.e., *Objects*). In contrast, JSD provides *Data\_Stream* and *State\_Vector* as two ways to model communication among system components (i.e., *Model\_Processes*).
- BOOD does not explicitly address the issues of modeling the interactions between system components and related events outside the system. In contrast, the JSD notions of *Connection*, *State\_Vector* and *Data\_Stream* address this issue.

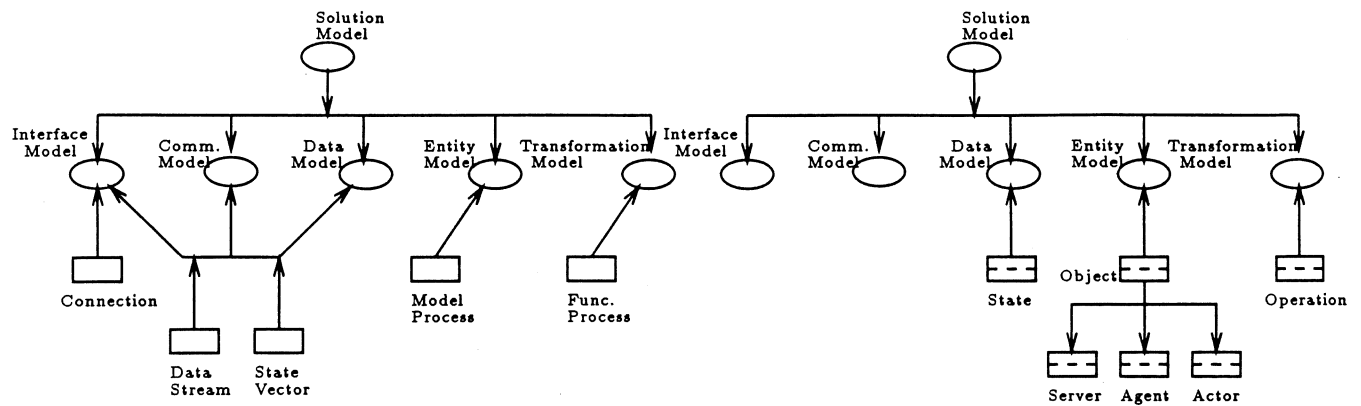
- *Model\_Process*, *Function\_Process* and *State\_Vector* or *Data\_Stream* in JSD should respectively address issues similar to those addressed by *Object*, *Operation* and *State* in BOOD.

We can now see that this classification can be used to identify components that address similar issues. However, we also can see that this classification does not illustrate how these components are similar or different. This is exactly the reason why we continue the comparison, taking this classification as a road map for identifying the components that should be compared further.

### 4.2.3 Step 3: Select Comparison Topics

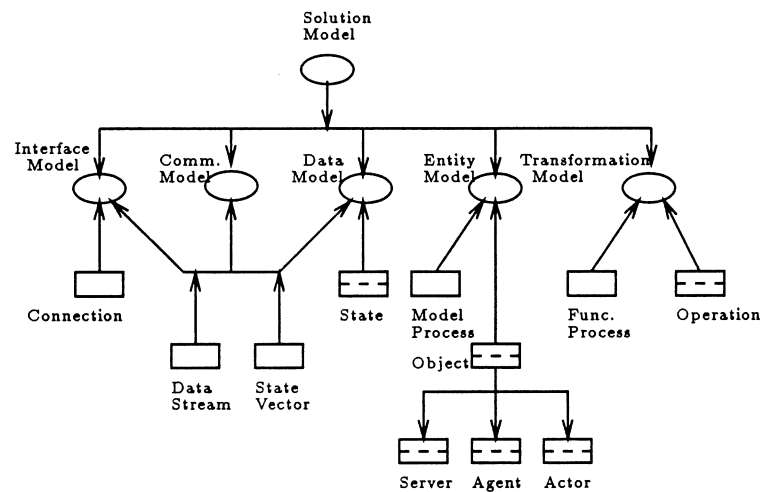
Fig. 4.4(c) indicates that JSD's *Model\_Process* and BOOD's *Object* address similar issues. Thus, we consider them comparable. In addition, since they describe the fundamental structures (e.g., sequential process structure) of the design artifacts, we believe that it should be important to compare them. Using a similar rationale, we should also compare JSD's *Action* with BOOD's *Operation*, and JSD's *State\_Vector* with BOOD's *State*.

Applying the guideline suggested in section 2.4.1, we compare the correspondingly related components of same type. As our JSD and BOOD models focus only on defining the functions of the design actions, we focus on comparing these actions.



(a) Classification of the JSD components

(b) Classification of the BOOD components



(c) Merged classification

Figure 4.4: Classifications of the JSD and BOOD components under the Solution Model Domain

## 4.2.4 Step 4: Make Comparisons and Develop Process Code

In this section we compare the design actions identifying and specifying the components chosen to be compared. By doing this, we hope to demonstrate how one can, based on the models of the SDMs, find the differences/similarities in the aspects described in section 2.4.2.

### Comparing *Object* with *Model\_Process*

#### Comparing Identification Actions

The process models (Fig. 4.1(f)(1) and Fig. 4.2(b)) help us in understanding the following:

- **Differences in inter-artifact dependency:** By comparing the inputs of *Identify\_Model\_Process* in JSD with *Identify\_Object* in BOOD we see that developing a *Model\_Process* depends on *Real\_World\_Process* and developing an *Object* depends on *Req\_Spec*. To understand if they are actually different or not, we analyzed the activities (Fig. 4.1(e)(1)(2)) producing *Real\_World\_Process* and tried to identify the activity producing *Req\_Spec*. As a result, we found that no activity defined in BOOD is used to define *Req\_Spec*. By doing so and by checking the informal description of BOOD, we understand that *Real\_World\_Process* (i.e., *Entity\_Structure*) is a well-defined JSD artifact as opposed to *Req\_Spec* which is not well-defined in BOOD. (Note that the models of the artifacts (e.g., *Req\_Spec*) should help

us in deciding this. But since HFSP does not support modeling artifacts, we have to check the informal description of *Req\_Spec* in BOOD)

- **Different need for human involvement:** Fig. 4.2(b) indicates that *Identify\_Object* consists of a number of subactivities. Fig. 4.2(b)(9) indicates that each of those activities identifies a particular kind of object. Thus, BOOD provides guidelines for identifying *Object*. In addition, based on common knowledge, we understand that deciding if a noun represents an object is a human process. In contrast, Fig. 4.1(f)(1) does not show *how* to identify a *Model\_Process*. This motivates us to specify the details of *Identify\_Model\_Process*.

HFSP provides strong help for describing *what* artifacts are produced rather than *how* they are to be produced by a design action. Thus, it seems that a formalism that can complement this should be used to specify *Identify\_Model\_Process*. As Ada [Uni83] can be used to procedurally describe processes in detail, we choose to use Ada-like notations to code this action (Fig. 4.5). Fig. 4.5 illustrates that each JSD entity should correspond to a model process that supports the whole life-cycle of the entity (note that the procedure parameters are consistent with their uses in Fig. 4.1(f)(1)). Therefore, in contrast to *Identify\_Object*, a human guided process, the model processes could be mechanically identified based on the given entities.

Coding *Identify\_Model\_Process* seems also to support arguments we made in section 3.1.2 and 3.3.2. In those sections we suggested that any single existing SPMF is probably not sufficient for precisely and effectively modeling every aspect of an SDM. However, using a number of complementary selected SPMFs should reduce this problem. We also noted that an SPMF capable of modeling details

---

```
Procedure Identify_Model_Process(  
    r_processes: IN entity_structure_list;  
    names      : OUT m_process_name_list) Is  
    -- entity_structure_list and m_process_name_list  
    -- are defined in the JSD data type definitions.  
Begin    For i IN r_processes Loop  
        names[i] := i.entity_name;  
        -- we assume that an entity_structure  
        -- has a field called entity_name.  
    End Loop;  
End Identify_Model_Process.
```

Figure 4.5: The code of the action for identifying *Model\_Processes*

---

of the process is sometimes necessary for understanding the differences between method components. Recognizing this, our research approach is to start by taking one plausible SPMF (HFSP) as a start, to then iteratively identify what method components and what aspects of those components need to be modeled, and to then correspondingly identify what additional SPMFs should be used. In the next chapter we will discuss the development of such a modeling formalism.

### Comparing Specification Actions

Fig. 4.6 and 4.2(b) and (c) illustrate:

---

```

Specify_Model_Process(Connection_List, Real_World_Model, M_Proc_Name_List|Model_Process_List) ⇒
Translate_Structure(Real_World_Model, M_Proc_Name_List|Init_Model_Process_List)
Add_Connection(Connection_List, Init_Model_Process_List|Model_Process_List)

```

Figure 4.6: The model of defining *Model\_Processes*

---

- **Different need for human involvement:** specifying a *Model\_Process* from *Entity\_Structure* and *Connection* is a fairly mechanical process (procedural descriptions of *Translate\_Structure* and *Add\_Connection* should show this clearly). In contrast, since *Identify\_Operation* contains subactivities each of which identifies one kind of *Operation*, we can view *Identify\_Operation* as a guided human process. Since *Operation* is a part of *Object*, the process of specifying object must also be a human process.
- **Differences in scope:** Fig. 4.6 shows that *Connection* is an input to *Specify\_Model\_Process* and *Add\_Connection* is a subactivity of the same action. Since *Connection* addresses the issues of communication between the system and the real-world (note that this entails our understanding of the informal description of the meaning of *Connection*), we can see that a *Model\_Process* must be specified to describe how it communicates with a *Real\_World\_Process*.
- **Different development procedure:** Fig. 4.2 illustrates that in BOOD, the order (i.e., *Time\_Order*) in which the *Operations* are executed is specified as part of the specification of *Operations* and therefore as a part of *Object*. In contrast, Fig. 4.1 does not illustrate that *Specify\_Model\_Process*



requires one to specify the order in which the *Function\_Processes* can be performed. However, after reviewing a larger part of the model, we found that *Entity\_Structure* defines the order in which the JSD actions are performed. Since this order constrains the order in which the *Function\_Processes* might be performed, we see that the two SDMs both address this issue but use different procedures. Note that HFSP makes it harder to determine this as it is a functional, not procedural, modeling formalism.

This example shows how one could identify differences in scope and development procedures. The strategy used here is to compare the control flows and the subactivities. If finding that a subactivity of activity *A* addresses an issue activity *B* does not address, one may conclude that there are differences in the issues that activities *A* and *B* address. However, one should not immediately conclude that the two corresponding SDMs have such a difference since the SDM containing activity *B* may address this issue through other activities. Thus, more of the model of this SDM might have to be checked. If one finds that the issue is indeed addressed through other activities, one may conclude that the two SDMs differ in development procedures. This suggests a needed refinement to the comparison process CDM.

### **Comparing Operation with *Function\_Process***

In JSD, a *Function\_Process* is defined to achieve the outputs the customers desire. The model (Fig. 4.7) shows the procedure for adding the *Function\_Processes* to the *Model\_Processes*. This procedure shows that *System\_Functions* and *Function\_Process* are defined in *Model\_Processes*. For each *Model\_Process*,

---

```
Procedure Define_Func(Spec : IN Init_System_Spec_Diagram,
                    Output: OUT System_Function,
                    Func   : OUT Function_Process) Is
Begin
  For Model_Process in Spec Loop
    For Action in Model_Process Loop
      1) Define_Its_Func(Action, Func, Output);
         -- function processes that support actions directly.
    End Loop;
      2) Add_Other_Func(Model_Process, Func, Output);
         -- function processes that support a model process.
    End Loop;
      3) Add_Other_Func(Spec, Func, Output);
         -- function processes that support a function
         -- to be achieved by mutilple model processes.
    End;
End;
```

Figure 4.7: The model of specifying *Function\_Process* and *System\_Function*

---

and then each of its actions, certain *System\_Function* and *Function\_Process* will be defined. After that, some additional *Function\_Process* might be defined to achieve some additional *System\_Functions* which depend on a number of actions which may be embedded in different *Model\_Processes*. JSD suggests that *Function\_Process* should be chosen based on the outputs the customers desire. Based on description given in [Jac83], we modeled a procedure (Fig. 4.7) for adding the *Function\_Processes* to the *Model\_Processes*.

## Comparing Identification Actions

- **Different need for human involvement:** The identification of a *Function Process* depends on the customer desired outputs and the actions simulated by the *Model\_Process*. Therefore, some of them (Fig. 4.7(1)) could be identified by these actions. However, others (Fig. 4.7(2)(3)), which could be about how to produce the outputs, may require human experience and intelligence. In contrast, the *Operations* of BOOD are totally identified by applying guidelines.

## Comparing Specification Actions

- **Differences in inter-artifact dependency:** *Operation* is an action an object requires or undergoes, and thus is encapsulated inside the object definition. Similarly, a *Function\_Process* could respond to an *Action* required or suffered by an *Entity* (specified in 1) of Fig. 4.7). However, in contrast, a *Function\_Process* could also be directly activated by the *Actions* in a number of the model processes (specified in 2) and 3)). Thus, a *Function\_Process* may not depend on any particular model process.
- **Differences in scope:**
  - A *Function\_Process* must have outputs (Fig. 4.7(1)). In contrast, an operation may or may not have an output, which is not explicitly defined in BOOD.
  - JSD requires specification of the time duration for performing a *Function Process* (see JSD model(d)(2)). BOOD does not explicitly address this issue.

## Comparing State with State Vector

JSD and BOOD both fail to define the actions that are used to specify *State\_Vector* and *State*. However, JSD model(f)(2) and BOOD model(b)(8) help us to understand:

- **Differences in scope:** *State\_Vector* is an alternative notion mainly for building the *Connection* between the *Model\_Process* and *Real\_World\_Process*. In contrast, *State* of *Object* is introduced only from the aspect of recording the internal state of the object, which is a distinct characteristic of *Object*. Since internal state can be used for communication, we think that *State* is a more general notion.
- **Differences in procedures:** JSD specifies in more detail when to specify *State\_Vector*. However, since *State* and *State\_Vector* should be specified respectively when *Object* and *Model\_Process* are specified, the difference is relatively small.
- **Differences in inter-artifact dependency:** design of a *State\_Vector* depends on the *Real\_World\_Process* with which the *Model\_Process* communicates. In contrast, *State* of an *Object* is a more general notion that is defined in BOOD as not depending on any concrete artifacts other than the *Object*.

### 4.2.5 Step 5: *Summarize\_Difference*

Table 4.1 summarizes the differences between the similar components of JSD and BOOD. This summary is a complete description of the differences that are

identified by comparing the design actions of JSD and BOOD. Comparisons which might be made from other aspects, like criteria or artifact composition, may provide additional evidence that supports the differences we have identified here, and/or help to reveal other differences.

This table summarizes that JSD provides strategies for devising the system interface and system components; it addresses issues of communications between real-world activities and system components; and it suggests a way to define system functional requirements in terms of system components.

In contrast, BOOD does not provide a detailed strategy (e.g. when) for how to model the system interface and the communications between real-world activities and system components. BOOD assumes that system functions already are or will be defined by some other activities, and thus provides no strategy for describing those functions. However, unlike the JSD entity, which in most cases represents a thing that exists in the environment using the system, the BOOD object, depending on an unrestricted *Req-Spec*, could correspond to a component in the environment using the system as well as one in the support environment. Thus, BOOD should help in identifying and designing the interactions of the system interface with the support environment.

#### **4.2.6 Integration of JSD and BOOD**

Here we discuss briefly how one might integrate these two SDMs. By doing this, we hope to demonstrate that comparisons resulting from using CDM are directly effective in aiding the integration of SDMs.

<b>Comparisons with Respect to the Solution Model Domain</b>				
<b>The Entity Model</b>				
<b>Component</b>	<b>Dependency</b>	<b>Scope</b>	<b>Need for human</b>	<b>Proc.</b>
<i>Model _Process</i>	<i>Entity_Structure</i> well defined	communication between the real-world activities and system components	mechanical  here	time order not defined
<i>Object</i>	<i>Req_Spec</i> not defined	system components, which may not communicate to real world activities.	guided	specify time order
<b>The Transformation Model</b>				
<b>Component</b>	<b>Dependency</b>	<b>Scope</b>	<b>Need for human</b>	<b>Proc.</b>
<i>Function_Proc.</i>	customer and <i>Model_Process</i>	outputs and time delay	mechanical/ guided	N/A
<i>Operation</i>	<i>Object</i>	time delay and outputs may not be specified	guided	N/A
<b>The Data Model</b>				
<b>Component</b>	<b>Dependency</b>	<b>Scope</b>	<b>Need for human</b>	<b>Proc.</b>
<i>State_Vector</i>	<i>Real_World_Proc.</i> <i>Model_Process</i>	communication	no guideline is provided	N/A
<i>State</i>	<i>Object</i>	recording internal state of object	no guideline is provided	N/A

Table 4.1: Summary of the differences between JSD and BOOD

Based on the summary and analyses above, we directly get the following hints for integrating JSD and BOOD:

- One may derive a BOOD object from a JSD model process (but this does not mean that every object must be derived from a JSD model process). This object provides the text to be executed by the model process. An operation on the object provides the text to be executed by a JSD function process embedded in the model process. The state definition of the object can define the data structure of the state vector of the model process.
- The order of executing the embedded JSD function processes, which is constrained in the definition of the corresponding *Entity\_Structure*, guides the specification of the time order of the operations of the object.
- Timing constraints on the JSD function processes will affect the implementation of the supporting operations in the object.
- BOOD can be used to identify and design the system components that provide the services to the JSD function processes.
- All the objects can still be documented in the format as BOOD suggested originally.

In a combined JSD/BOOD process (Fig. 4.8), one can use JSD strategies 1) to model the problem (by modeling the entity structure), 2) to define the system interface (by developing model processes), 3) to elaborate the system functions (by adding JSD function processes and specifying their functions), and then using these as guidelines to define the BOOD objects and to document these objects in the BOOD format. In this way, JSD and BOOD can complement each other. The issues which BOOD fails to address, can be coped with by applying

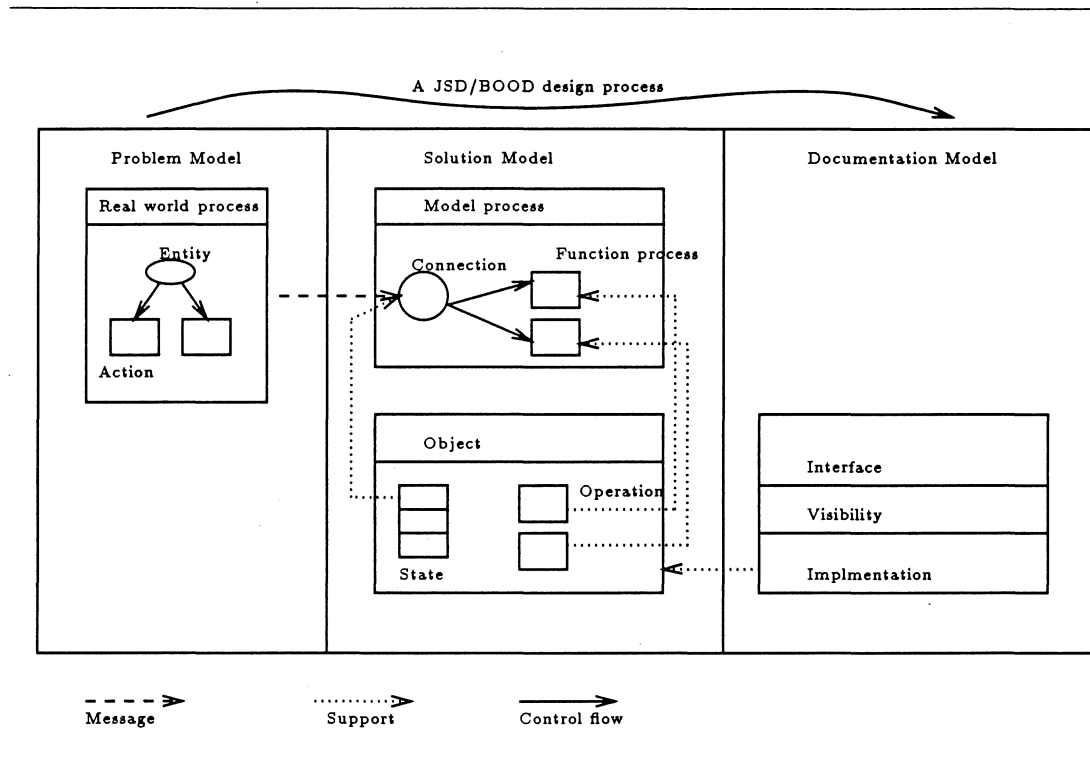


Figure 4.8: The domains manipulated by a JSD/BOOD integrated design process



JSD strategies. On the other hand, the issues which JSD fails to address (e.g. designing/documenting static programs and identifying the system interface with the supporting environment), can be addressed by the BOOD strategies.

Although strategies for integration of JSD and BOOD have been suggested before [EHZAG89, BC91], we believe that our suggestions based on the systematic comparisons using CDM, are more complete and explicit.

### **4.3 Evaluation of CDM**

To evaluate CDM and therefore to achieve our research goals, we must discuss the following two issues in the next two sections: 1) How effectively does process modeling help in comparing SDMs? 2) How effectively can CDM overcome the problems of previous comparison efforts (as described in Section 1.2 and 1.3)?

#### **4.3.1 Evaluating the Application of Process Modeling**

What has process modeling provided to aid in comparing SDMs? Based on the lessons learned during this experiment, we will address this question by answering three questions: 1) What aspects of a process does HFSP help to characterize explicitly? 2) What benefit do we gain from the characterizations? and 3) What do we desire to model, that is not supported by HFSP? Answering the first two questions would help us in understanding what modeling formalisms should be used to analyze what aspects of SDMs. Answering the last question would help us

in understanding what other modeling paradigms might need to be incorporated in an SPMF.

To answer the first question, we observe that the process models (Fig. 4.1 and 4.2):

- **Characterize software activity in a more *rigorous* and *explicit* way:**

The meaning of a software activity is explicitly described by 1) its name, 2) the meaning of its inputs and outputs, 3) the meaning of its subactivities, and 4) the control flow of its subactivities (for example, see Fig. 4.1(f), where *Model\_System* is rigorously defined as a function that maps *Real\_World\_Model* to *Init\_System\_Spec\_Diagram*. The descriptions of the subactivities (see Fig 4.1(1)(2)(3)) which explicitly indicate their inputs and outputs, help us to understand the meaning of *Model\_System*.)

- **Characterize the relations between the inputs/outputs of the activity and those of its subactivities:** For example, Fig. 4.1(a)(4) explicitly indicates that the output of JSD is an aggregation of the outputs of JSD's subactivities *Develop\_Spec* and *Develop\_Impl*.

To address the second question, we found that these characterizations aid us in comparing:

- **Development procedures or scopes of design activities:** The explicit models of control flow and subactivities can help to identify potential differences in SDM aspects (e.g., see section 3.5.1. for a comparison between the actions used in specifying *Model\_Process* and *Object*). However, we found that models are not by themselves sufficient to demonstrate these differences.

One illustrative example is that the JSD model does not express the meaning of *Connection*. Therefore, in our comparisons, we have to analyze the informal description of *Connection*.

- **Dependencies among artifacts:** Explicit definitions of input/output domains of design activities can help to show dependencies between design artifacts (as inputs and outputs of the activities), and thereby may help to show that two artifacts depend on different artifacts. Since no global variables exist in an HFSP model, the model should explicitly show all the potential dependencies. However, knowing whether two artifacts truly depend on different artifacts still requires us to understand more precisely all these artifacts. This may require that we model the design actions producing these artifacts, model these artifacts themselves, (so, HFSP may need to be extended to support modeling artifacts), or analyze the informal descriptions of these artifacts and the design actions. The way in which we checked to see if *Entity\_Structure* differs from *Req\_Spec* is an example of this.
- **Need for human involvement:** in our comparison, we found that the SDMs use three mechanisms to guide identifications of design artifacts: 1) the rules for deriving one kind of artifact from another kind of previously defined artifact, (e.g., derive *Model\_Process* from *Real\_World\_Process*), which usually require no human involvement, 2) criteria for deciding what an artifact is, (e.g., the criteria for deciding an *Entity\_Structure*), whose application often requires human involvement, and 3) the classification (or other kinds of decompositions) of a kind of artifact, (e.g., in BOOD, *Object* as a kind of artifact can contain other kinds of objects like *Server* and *Agent*). Our models in HFSP seem particularly helpful in expressing the third mechanism.

The attribute definition shows that an output of the activity is a union of the inputs of its subactivities and thus, the descriptions of these subactivities describe how to get the output of the activity. Since our models in HFSP do not model the procedural aspect of a design activity and the criteria for deciding an artifact, they do not help much in identifying the first two mechanisms.

To address the last question, we found that HFSP has some limits (described below) which are common to other formalisms, (e.g., SDA [Wil88]). These hinder us from effectively comparing other aspects of design actions.

- HFSP is incapable of expressing *which design representation (rendering)*, as designated by an SDM, should be used by a design *action* to specify the design *artifacts*. For example, JSD recommends that *Draw\_Entity\_Structure* should use a *representation Structure\_Diagram* to specify *Entity\_Structure*. Although one may express the designated *representation* as an input to the action, HFSP does not directly support an explicit distinction between this input as a *representation* and other kinds of inputs, (e.g., *artifact*).
- HFSP is incapable of indicating and characterizing the *design criteria* as a design *action* that should apply. For example, we would like to 1) more rigorously express the criteria for determining an entity, (e.g.,  $Entity = \{x | PerformAction(x) \vee \dots\}$ ), 2) indicate that *Identify\_Entity\_Action* applies these criteria. Due to these limits, our JSD and BOOD models in HFSP would not help much in comparing this aspect of design criteria and design actions.

### 4.3.2 Advantages and Limitations of CDM

Can CDM effectively overcome the problems of previous comparison efforts? What are the limitations of CDM? Following, we discuss some advantages of CDM over previous comparisons:

- **The models of SDMs show the bases for conclusions:** In contrast to previous comparisons, which usually describe an author's understanding rather than his/her reasoning, CDM explicitly shows at least a large part of the comparison process and a rationale for drawing conclusions. (For example, identification of a *union* relation is used as a basis for concluding that an activity is a guided human process.)
- **The comparison result can be evaluated:** Since the comparison process and rationales are explicitly shown, a comparison result can be evaluated by evaluating the comparison process and rationales.
- **CDM can be more systematically applied:** In contrast to previous comparisons which are often carried out in an *ad hoc* way, CDM suggests a way to more systematically compare SDMs. Consequently, the comparison results should be less dependent on their authors. For example, 1) the classifications guide the selection of comparison topics; 2) CDM suggests some systematic ways for identifying certain kinds of differences. This characteristic is very important for objectively comparing/evaluating software processes.
- **The comparison results should be more objective:** CDM prevents possibly misleading comparison results caused by comparing design examples/projects, since design examples are not involved in these comparisons.

- **The comparison results should be more explicit and precise:** 1) With the support of a framework that precisely defines the issues involved in software design, the results should more explicitly and precisely indicate which issues an SDM addresses. For example, the comparison explicitly shows that BOOD does not address issues of specifying the Interface Model. 2) The result explicitly shows the differences in development procedures; it shows precisely where in the development procedures the same issue is addressed by the SDMs. For example, our comparison indicates that in JSD the order constraint is defined in *Draw\_Entity\_Structure* while in BOOD a similar issue is addressed in *Identify\_Operation*.

Following, we discuss some limitations or difficulties in using CDM:

- **A model of an SDM still cannot completely describe the SDM.** Thus, comparisons based solely on the models may lead to some biased results. Therefore, in CDM, one must carefully analyze both the models and the informal descriptions of the SDMs to minimize these biases.
- **The fundamental ideas behind an SDM are hard to specify rigorously.** For example, one idea behind JSD is that a model of the system can set a context for defining system functions. This idea is very difficult to specify rigorously in any existing SPMF. Thus, CDM with current SPMF support seems to be powerless to expose the differences between these ideas.

It should be noted that our work is aimed at identifying the differences between the SDMs rather than evaluating SDMs. Thus, it does not address what application domains an SDM might be good or bad for. It does not address how easily and effectively the SDMs can be applied in practice. However, we believe that a complete and explicit comparison should significantly help to do these.

## 4.4 Suggested Improvements

In this experiment we have applied CDM to comparing JSD and BOOD. We have classified their components under all domains, made a number of interesting comparisons. However, we understand that using CDM to compare other SDMs (e.g., RDM [PC86]) should suggest the need for adjustment of our framework and CDM, and indicate needed improvements to the SPMFs. With an improved framework and SPMFs, CDM should be more effective.

To improve SPMFs and the models of SDMs, we expect to

- Identify the routines, primitives, and notations that are effective for describing design processes. By using those with semantics that are more precisely defined, a model of an SDM should convey information more precisely and explicitly. Thus, the comparisons will rely more on the models of SDMs. For example, *Identify\_Noun*, which is used in both JSD and BOOD, could be a routine. *Identify* and *Define* could be two primitives to be instantiated to describe some specific design actions, (e.g., *Identify\_Entity\_Action* and *Define\_Func*). A notation (e.g., @) may help to distinguish criteria from other inputs (*Select\_Entity(..., Noun, @Criteria|Entity\_List)*);
- Identify which language paradigms are effective for modeling which kinds of method components and which aspects of method components. Up to now, our research has indicated that functional, procedural and object-oriented (*has-subclass*) paradigms are effective for modeling design actions and understanding the organization of method components. We speculate that a rule-based paradigm should be effective for modeling design criteria

and guidelines. Modeling design *concepts* is very important in order to enable CDM to be used to compare the substance of SDMs.

In the following chapters, we will describe our efforts in developing *Modeling Formalism* and *Base Framework*. The *Modeling Formalism* to be developed will be more comprehensive than HFSP and thus can model more aspects of an SDM. The *Base Framework* will be more carefully reviewed and defined. Then, using the *Modeling Formalism* and *Base Framework*, we compare four more SDMs to further evaluate CDM, process modeling technologies, and the classification framework for comparison of SDMs.





# Chapter 5

## Supports Needed For CDM

### 5.1 Required Supports

Note that CDM, which adopts strategies similar to comparison Approaches 2 and 4 of Section 1.3, requires two important supports. The first one is a *Base\_Framework* (BF) that is fair enough to enable a more complete and objective classification of method components. A BF that hides some method components of SDMs and their features could hinder one from objectively assessing the SDMs. A BF should also allow method components to be classified objectively. Note that CDM does not itself provide a strategy for providing and evaluating a BF.

The second is a *Modeling\_Formalism* (MF) that is comprehensive enough to capture all major aspects and components of SDMs. The BF will help to classify method components objectively, thereby enabling more objective assessment of SDMs. The MF will help in specifying valid models of SDMs, thereby preventing us from being unfairly blind to specific features of SDMs. Again, CDM does not provide any comprehensive strategy for choosing these MFs. CDM only suggests using the SDM models specified in an MF to arrive at a more objective classification

of method components. Thus, we must develop this support to facilitate the use of CDM.

In Chapter 3, we have briefly described a prototype BF. We used this BF to explain the ideas of CDM and to help in carrying out the first experiment described in Chapter 4. In Chapter 4, we also used a SPMF, HFSP, as a prototype MF to help in carrying out the first experiment. We believe that those two supports should be sufficient for demonstrating the basic ideas of CDM. However, as we explained earlier (e.g., HFSP supports only function modeling), we also believe that they are insufficient in support of CDM for more objectively comparing SDMs.

In this chapter we describe an evolutionary development strategy to be used to develop BF and MF. In using this strategy, we develop a BF based on our analysis of a set of selected SDMs and we develop an MF under guidance of the BF. Conversely, we use the SDM models specified in the MF to evaluate the completeness of the classifications of BF in order to aid the evaluation of BF. In this chapter we describe a BF and MF we have been developing using this strategy, (the MF is also enhanced based on our evaluations of HFSP). We also demonstrate how the MF is used to model SDMs and how the BF is used to classify method components. Then, we evaluate the completeness, objectivity, and effectiveness of the BF.

It is very important to note that successfully developing such a BF and MF would contribute not only to the comparative study of SDMs, but also perhaps to the study of various other software development processes (e.g., the requirements specification process, or the configuration management process). Moreover, as [Boa90] indicates, "A unifying model [e.g., a fair BF] would not necessarily be

of immediate use to a system builder. But it would be a tool for academic analysis that could, in turn, yield structures and tools to a practitioner.”.

## 5.2 Why Evolutionary Development

Developing a BF that is sufficiently large and detailed to classify method components of a large variety of SDMs is a very difficult task, because software design activities cover an extremely wide range of issues and can be viewed from various perspectives.

Analyzing frameworks which have been proposed previously to help quick summarization of features of SDMs, Brandt's framework [Bra83] includes 1) origin and experience, 2) development process, 3) model, 4) iteration and tests, 5) representation means, 6) documentation, and 7) user orientation. The framework defined by Wasserman and Freeman [WFP83a] includes 1) methodology applicability, 2) technical concepts supported, 3) work-products and representation schemas, 4) quality assurance methods and 5) usage aspects by methodologies. Olive's framework [Oli83] includes 1) external, 2) conceptual, 3) logical, 4) architectural and 5) physical modeling. More frameworks can be found in [Kun83, BFL<sup>+</sup>83, Fre83, Gri78, PT77, BRS83] to help in understanding the difficulties for developing a BF.

Identifying an MF that is capable of modeling all major aspects of an SDM is also a very difficult task. The broad and complex nature of design activities as indicated above causes an SDM be a complex product. An SDM must incorporate *various types of components* to deal with a broad range of design issues. For example, Brandt's framework implies that an SDM must contain definitions of

modeling techniques, development procedures, documentation standards, pictorial notations, measurement techniques, and evaluation criteria. Further, these definitions are often not isolated but rather typically closely related in various ways. This diversity requires various modeling paradigms and notations to ensure the validity, precision and understandability of models of these definitions.

Considering these difficulties, we think that evolutionary development is perhaps the only way for us to arrive at a suitable BF and MF. In using this approach, we construct an initial MF or BF based on the features and characteristics of some major method components of a few selected SDMs, and then extend the BF and MF, depending upon the demands of modeling or classifying more method components. This seems to ensure that the BF will be sufficiently fair to classify method components and to ensure that the MF will be comprehensive enough to model SDMs. This allows more realistic evaluation of the BF than extensively reviewing it against a large number of method components. This also minimizes the problems of accommodating new aspects of SDMs that need to be modeled.

We expect that BF will not evolve to a mature stage in a short time (e.g., three years) but rather over a long term (e.g., eight years). We expect a similar situation for MF. However, we believe that a more systematic evolution of BF and MF should greatly shorten the time the evolution would take and save many potentially duplicated efforts.

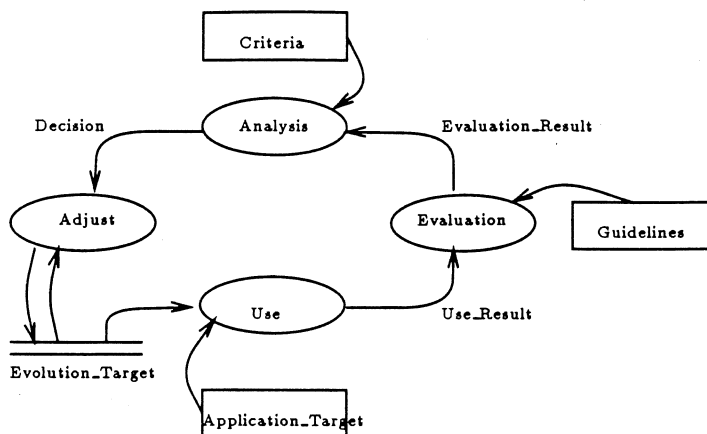


Figure 5.1: Evolution process of a *Evolution Target*

### 5.3 Evolutionary Development of BF and MF

To evolve BF and MF more systematically, we must precisely and concretely define the factors that affect the evolution processes. Fig. 5.1 describes a model of these processes and indicates these factors. As the figure shows, *Evolution Target* is to be used on *Application Target*, and is then to be evaluated in following *Guidelines*. Then *Analysis* will decide, depending on *Evaluation Results* and applying *Criteria*, if *Evolution Target* needs to be adjusted. The more *Application Targets* that *Evolution Target* is applied to, the more mature will an *Evolution Target* be.

Based on this evolution model, we defined the factors for developing BF and MF, respectively in Fig. 5.2 and 5.3. Those factors emphasize only improvement upon the completeness of the BF and MF.

In order to more easily evaluate the *Evolution Targets*, we must find a strategy to evolve them. The discussions in the second and third paragraphs of the last section imply that there is a strong connection between the BF (i.e., various

- 
- *Evolution\_Target: Base\_Framework* (BF).
  - *Application\_Target*: Method components (explicitly modeled and/or highlighted in *Process\_Model*).
  - *Use\_Results: Classification* as the output of *Classify\_Components* of CDM.
  - *Guidelines*: For components for which there is no place in BF examine where they can be appropriately placed. More specifically, identify what issue an existing method component addresses but which has not been incorporated into the BF. Identify what structure an existing method component has, which has not been incorporated into the BF.
  - *Evaluation\_Results*: The descriptions of those issues which have been addressed but which are not in the BF and the structures which have been used which are but not in the BF.
  - *Criteria*: The necessary conditions leading to augmenting or refining the BF are 1) existence of a method component which cannot be appropriately classified and 2) identification of a portion of BF which cannot effectively distinguish the key differences of the method components which are classified within this portion.
  - *Decison*: A specification of what adjustments need to be made.

Figure 5.2: Definitions of the factors affecting evolution of BF

---

- 
- *Evolution\_Target: Modeling\_Formalism* (MF).
  - *Application\_Target*: SDMs to be compared.
  - *Use\_Results: Process\_Models* of the SDMs, as the output of *Build\_Process\_Model* of CDM.
  - *Guidelines*: 1) Examine what aspects and components of the SDMs the MF is incapable of modeling. 2) Evaluate how effectively the models specified in the MF support classifications of method components.
  - *Evaluation\_Results*: Descriptions of those aspects and method components that the MF is incapable of modeling or whose models are not rigorous and explicit enough to support their classifications.
  - *Criteria*: The necessary conditions leading to adjusting the MF are: 1) new aspects or components; 2) the models specified in the MF, which are not sufficiently precise and explicit. Determination of whether or not sufficient is primarily based on the tradeoff between using existing formalisms that can be readily adapted to the MF to effectively overcome the weaknesses or creating new one.
  - *Decison*: A specification of what adjustments need to be made.

Figure 5.3: Definitions of the factors affecting evolution of an MF

---



types of components) and MF (i.e., various modeling paradigms). We think that our development strategy should make use of this connection. We exploit this connection to define an evolutionary development strategy (shown in Fig. 5.4).

Here, we focus on describing only how we will develop the BF using this strategy. First, based on analysis of SDMs, we select a set of SDMs (*Selected\_SDMs*) and then construct an initial version of a BF. Second, based on the BF, we define an initial version of an MF. Third, we do *Build\_Process\_Model* and *Classify Components* (they are explained in Sec. 3.1 and 3.2). Fourth, we evaluate *Classification* using *Process\_Models*, examining which method components cannot be classified within the BF. Fifth, we analyze reasons for this (*Analysis*). Sixth, we adjust the BF (*Adjust*) to improve the BF (we expect the improvement to be effected, in most cases, through extending the BF.). Repeating these steps (3,4,5,and 6) until the BF becomes stable, we may then *add* (step 7) more SDMs to *Selected\_SDMs*, which may restart this evolution process over.

In the next section we present the MF and BF we have developed so far. We intend to indicate that continually evolving them using this strategy should lead to a fair BF and a sufficiently comprehensive MF. Further, we believe that this sort of iterative evolutionary development of key frameworks and modeling formalisms is consistent with the ways in which more mature scientific disciplines operate. Thus, we hope that this effort indicates a way in which software engineering can begin to grow into a more mature scientific discipline. While we consider the proposed CDM, BF and MF to be very important, we view our proposed evolutionary strategy for developing them to be even more significant.

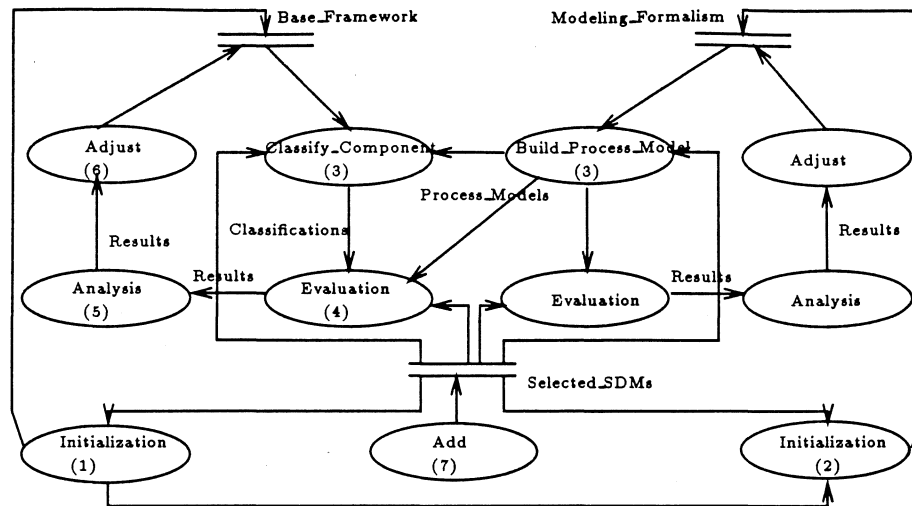


Figure 5.4: Evolutionary development processes for Base Framework and Modeling Formalism, represented as a data flow diagram. The numbers labeled on edges indicate a scenario of evolving the BF.

## 5.4 The BF and MF

### 5.4.1 The Selected SDMs

As Fig. 5.4 shows, in order to begin the evolutionary development process, we must select an initial set of SDMs for which the MF and BF can be initialized, used, and evaluated. We believe that at the initial stage of evolution, all the SDMs selected should be general purpose and yet based on diverse approaches. We think that issues and structures deduced from such SDMs are more likely to be shared with other SDMs, and this is likely to expedite the identification of a comprehensive set of fundamental issues that method components must address and basic structures they often take. Based on this rationale, we

selected JSD [Jac83], BOOD [Boo86], RDM [PC86], DSSD [Orr77], LCP [War76] and SD [YC79, SMC74].

#### 5.4.2 *Base\_Framework* (BF)

There are a number of ways of developing a framework for classifying method components—1) based on a software development life-cycle (e.g., [Was80]); 2) based on the concepts used in software design (e.g., [Gri78]); 3) based on properties of method components (e.g., [Fre83]); and 4) based on support facilities provided (e.g., [PT77]). However, in our effort, we developed the BF guided by its roles in CDM.

In CDM a BF plays two roles. The primary role is to guide identification of existing method components which are comparable. The secondary role is to be used at times as a basis for assessing SDMs (i.e., checking features of SDMs against a feature framework). To achieve these, we believe that a BF should consist of two parts: 1) a Function Framework, which aggregates design issues that existing method components have addressed (external properties) and 2) a Type Framework, which aggregates internal characteristics (e.g., structures) that existing method components have had. Thus, in order to satisfy its primary role, such a Function Framework should help in identifying which method components address similar issues while such a Type Framework should help in identifying which method components have similar natures. Thus, this should ensure meaningful comparisons (i.e., apples vs. apples) and guide identification of which method components could be compared further (as we have shown in Chapter 3 and 4). In order to satisfy its secondary role, such a Function Framework should be useful

in revealing what different issues method components address while such a Type Framework should be useful in revealing different natures of method components.

To facilitate incremental development of a BF, the BF must have an extensible structure. As a hierarchical structure can be easily extended or adjusted, we chose hierarchy to serve as the structure of the BF. An edge of the hierarchy denotes an *is-a* [KM85] relation. Note that the BF described briefly in Sec. 3.2 is a very early version of the BF that has been enhanced as will be described in this section.

### Type Framework: MCTH

In analyzing our selected SDMs (including the work described in Sec.4.2.2.), we identified a number of types that are useful in characterizing the structure of SDMs and their method components. We call this Type Framework the Method Component Type Hierarchy (MCTH). Figures 5.5 and 5.6 defines its top-level types, gives examples of their subtypes, and indicates from which components these types are deduced.

Thus, for example, note that *Concept* is further decomposed based on the nature of a *concept*. We expect that such decomposition should effectively guide comparisons. Using this decomposition, one may first compare *problems* SDMs address, and begin to identify the *principles* to be used to deal with the *problems*, and the *guidelines* for developing the *artifacts* that support the *principles*. Eventually, the analyst should be able to find the *problems*, *principles*, and *guidelines* that could be compared.

*Artifact* is decomposed based on the formality with which an *artifact* is defined because we think that MSDL, defined later, can ensure effective comparison only of those *artifacts* with similar roles. Decomposition can help to reveal the differences between the structures of *artifacts*. Based on a rationale similar to that used for decomposing *artifact*, we decompose *representation* and *action* based on the level of formality with which a *representation* is defined and the technical nature an *action* has.

The Method Component Type Relation Matrix (MCTRM)(Fig. 5.7) describes some of the most common relations among instances of those types. For example, MCTRM[*Action*, *Concept*] describes that an *action* should apply some *concept*. In Sec. 5.5.2, a more complete version of MCTH is presented with the classifications of the method components.

### **Function Framework: MSDL**

The Function Framework is a set intended to contain all the design issues that have been addressed by existing method components. Each issue defines a category used to organize the method components that address this issue. This set can be divided in a number of ways (e.g., from the perspectives listed by Freeman in [Fre83]) to organize classifications of the method components.

After identifying and analyzing the method components of the selected SDMs (including the work described in Sec. 4.2.2.), we decided to divide this issue set according to the modeling and documentation characteristics an issue is about. By analyzing the selected SDMs, we found that relations between the method components and these issues are often rather well understood and rather explicitly

---

**Concept:**

- **Definition:** an idea that influences the design of an SDM.
- **Subtypes:** decomposition criterion: conceptual role a concept plays in an SDM.
  1. *Problem* of software design and software application. (e.g., reduce software complexity (SD))
  2. *Principle* for coping with these problems. (e.g., design software with high degree of cohesiveness (SD))
  3. *Criteria* for deciding what constitutes an artifact. (e.g., decide a JSD entity)
  4. *Guideline* for designing software and coping with these problems. (e.g., find a point of “highest abstraction” in the data flow (SD))
  5. *Measures* for quantitative comparison or evaluation of the quality of artifacts.

**Artifact:**

- **Definition:** a description of some sort of entity involved in a design process.
- **Subtypes:** decomposition criterion: formality with which an artifact is defined.
  1. *Programs:* (e.g., a model process (JSD)).
  2. *Diagram:* (e.g., a data flow diagram (SD)).
  3. *Relation:* (e.g., use-hierarchy (RDM)).

Figure 5.5: Part of BF: Definitions of the top-level types in Method Component Type Hierarchy (MCTH)

---

---

**Representation:**

- **Definition:** a means for describing or specifying design *artifacts*.
- **Subtypes:** decomposition criterion: degree of formality of a representation.
  1. *Language:* (e.g., the structure language (JSD)).
  2. *Diagrammatic notation:* (e.g., Structure\_Diagram\_Notation (JSD)).
  3. *Mathematical representation:* (e.g., RDM uses relations to specify system functions)

**Action:**

- **Definition:** one or more physical and/or mental processing steps used in design. An *action* may create or modify a design *artifact*.
- **Subtypes:** decomposition criterion: the technical nature of an action.
  1. *Develop:* (e.g., develop system specification of JSD);
  2. *Model:* (e.g., model the environment outside the system in JSD)
  3. *Decompose:* (e.g., decompose a function in Structured Design)
  4. *Specify:* (e.g., specify implementation of a module in BOOD)
  5. *Define:* (e.g., define interface of a module in BOOD)
  6. *Derive:* (e.g., derive a program from the data structure of its output in DSSD)
  7. *Identify:* (e.g. identify objects in BOOD)
  8. *Select:* (e.g., select entities in JSD)

Figure 5.6: Part of BF: Definitions of the top-level types in Method Component Type Hierarchy (MCTH)(cont.)

---

	<i>Concept</i>	<i>Artifact</i>	<i>Representation</i>	<i>Action</i>
<i>Concept</i>	is-a is-part-of	affect decide*	affect	affect
<i>Artifact</i>	support	is-a is-part-of *	determine	affect
<i>Representation</i>	support	support *	is-a is-part-of	influence
<i>Action</i>	apply *	input * output *	apply	is-a is-part-of *

Figure 5.7: MCTRM: Method Component Type Relation Matrix

described in SDMs (e.g., JSD clearly describes what issues the entity structure addresses), and therefore should be more objectively decidable. More importantly, we found that components addressing these issues occupy a large and central part of the selected SDMs. Because of this, we defined a Model of the Software Design Life-cycle(MSDL) (Fig. 5.8) that emphasizes modeling and design documentation.

MSDL is defined as a transformation from a software application problem to the software design. The *Problem Domain* is decomposed based on the application domains and characteristics of software systems. The *Problem Model Domain* and *Solution Model Domain* are decomposed, respectively based on aspects according to which a problem or a software system needs to be modeled. *Design Document Domain* is decomposed based on aspects according to which a software design needs to be documented. With these decompositions, issues can be organized according to the life-cycle phases in which they need to be addressed. Further, method components can be classified under those organizational issues. In Sec. 4.2 and the



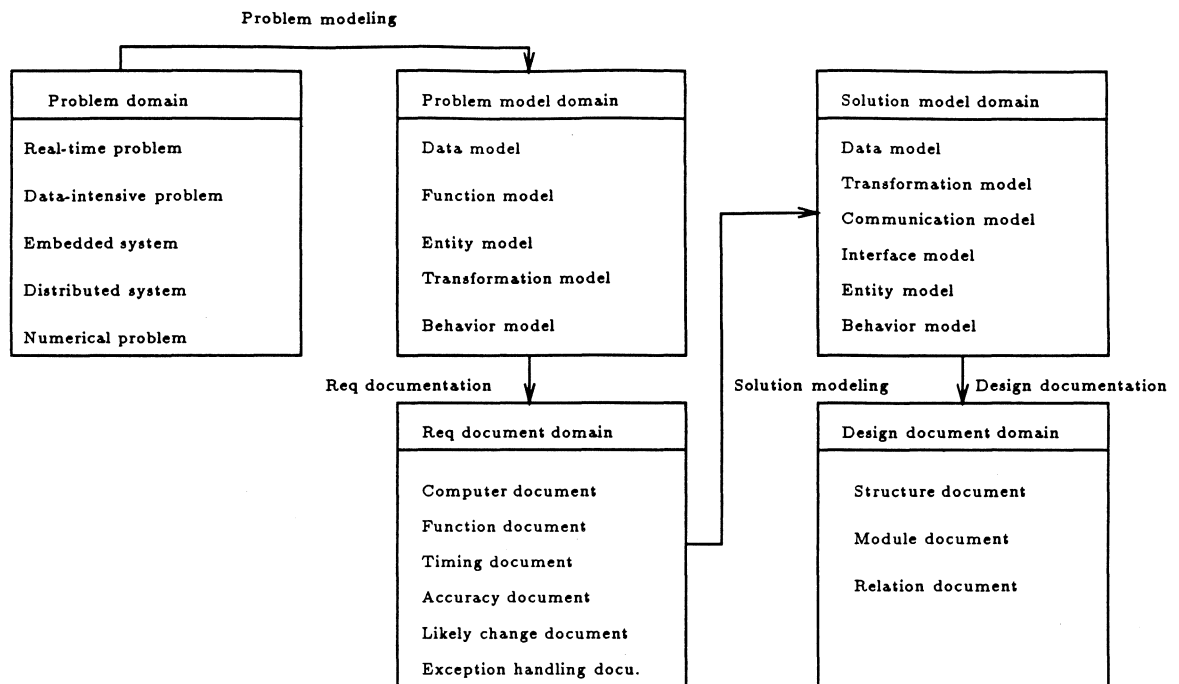


Figure 5.8: A Model of the Software Design Life-cycle (MSDL)

Appendix we show a more complete version of MSDL with the method component classifications under this framework. Again, our actual decompositions (Fig 5.8) are based on analyses of the selected SDMs (e.g., the decomposition of *Problem Model Domain* is deduced from analyzing JSD and SD).

### 5.4.3 *Modeling Formalisms (MF)*

In CDM, an MF is used to aid explicit characterization of method components and inter-component relations to support analysis and classification of these components. As we discussed earlier, an MF should be capable of modeling all major components of an SDM to avoid incomplete and misleading comparisons. Using our evolutionary development strategy, we adopted a number of modeling

---

**Data Model:** Descriptions of data structures from the customers' point of view. These data structures are usually logical in the sense that they could be implemented in a number of ways.

**Transformation Model:** Descriptions of data flows that are visible to customers and/or helpful for designers in understanding the problem. These kinds of data flows usually start as inputs to the system and end up to the outputs from the system. (e.g., data flow diagram of SD)

**Entity Model:** Descriptions of the environment outside the system. This environment is the one under which the system will be used and/or operated. These descriptions should be in terms of the environment components and their behaviors. (e.g., entity structure of JSD)

**Function Model:** Descriptions of the desired system outputs, usually expressed as functions of system inputs. (e.g., system function of JSD)

Figure 5.9: Definitions of the Problem Model Domain framework

---

---

**Data Model:** Descriptions of data structures used to realize the system.

**Transformation Model:** Descriptions of how desired outputs are to be computed by the system. This may entail identifications and elaborations of system programs involved in the computations. (e.g., data flow diagram of SD)

**Entity Model:** Descriptions of the system in terms of the system components and the operations they may perform and/or undergo. A system component must 1) exist in time and space, and 2) perform and/or undergo actions. (e.g., object of BOOD)

**Interface Model:** Descriptions of how real-world events communicate with system components. This may require identifications and elaborations of the system components responsible for such communications. (e.g., connection of JSD)

**Communication Model:** Descriptions of how system components communicate with each other. (e.g., data stream of JSD)

Figure 5.10: Definitions of the Solution Model Domain framework

---

---

**Structure Document:** Documents that decompose the design into modules and briefly describe every module. (e.g., module guide of RDM)

**Module Document:** Documents that describe the details of every module. The description should be about how a module can be used and how it is implemented. (e.g., module specification of RDM)

**Relation Document:** Documents that describe the relations between the modules and/or between parts of the modules. (e.g., use-hierarchy of RDM)

Figure 5.11: Definitions of the Documentation Domain framework

---

formalisms into the MF (see Fig 5.12) in order to model method components of those types and relations defined in MCTH.

Using the evolutionary development strategy, we believe it is important to try to avoid adopting formalisms that might be too complex and too powerful. Also, as humans will use the MF, the formalisms adopted should have good understandability and expressiveness, while supporting a high level of formality.

A version of HFSP, that has been enhanced based on our earlier experiment (see Chapter 4) in this evolutionary development process, has been adopted into the MF primarily because it can model 1) a design *action* rigorously as a function that maps some *artifacts* to other *artifacts*, thereby capturing the relations included in MCTRM[*Action*, *Artifact*], 2) a design *action* through the hierarchy of functional abstractions, thereby capturing one relation included in

MCTRM[*Action, Action*], and 3) the *concepts* a design *action* should apply, thereby capturing the MCTRM[*Action, Concept*] relation.

To complement the weakness of HFSP in modeling *artifacts*<sup>1</sup>, the Warnier Diagram is adopted into the MF. The Warnier Diagram is capable of clearly modeling *is-part-of* relations among *artifacts*, which characterizes a part of the MCTRM[*Artifact, Artifact*] relation.

Set definition notation is adopted into the MF because it is capable of defining *criteria* for deciding an *artifact* set membership, characterizing the MCTRM[*Concept, Artifact*] relation. The predicates that define the properties of an *artifact* could be specified either formally (e.g., in logical notations) or informally (e.g., in English). An *Artifact/Representation* table is incorporated into the MF to characterize the MCTRM[*Rep., Artifact*] relation.

We have marked the MCTRM entries with star in Fig 5.7 to indicate the relations which can be at least partially characterized by the MF. Based on the evaluation (Sec. 4.3.1) and our knowledge about modeling formalisms, we view those relations as being either essential for comparing SDMs or more easily modeled with the existing modeling techniques. For example, an MCTRM[*Concept, Representation*] relation that describes how a *concept* affects a *representation*, is not easy to model formally and, moreover, does not seem to be very useful for comparing two related SDMs.

To utilize this MF which consists of a number of modeling paradigms, an architecture of the SDM model can be a template consisting of a number of fields.

---

<sup>1</sup>This conclusion is based on the version of HFSP described in [Kat89]

Each field defines the method components of certain type, or defines certain aspects (relations) of these components. All the definitions in these fields should be consistent, (e.g., names of components should be consistently defined and used). We define such a template by Fig. 5.13 and 5.14. These two figures present and define the structure and fields of the template.

Note that every field in the template is optional. Analysts are responsible for deciding whether a field should be specified or not. The decision could be made based on 1) in what aspects SDMs will be analyzed and/or compared and 2) how effectively specification of the field can characterize the SDM (e.g., our experience has indicated that ACH can characterize RDM effectively, but may not for other SDMs). It is expected that some fields (e.g., ACH and AFH) should be specified more often than others (e.g., ATH and CDA).

It should also be noted that a given field of the template may not necessarily be specified for all components of an SDM. (For example, a ATH may not contain all the artifacts of an SDM, but perhaps only those artifacts related through the *is-a* relation). Only when we can specify the aspects of components effectively should those components be specified.

To evaluate the BF and MF, we must use them to classify method components and model SDMs. In the next chapter, we will describe this effort.

1. HFSP [Kat89]: a software process modeling formalism. Based on our previous evaluations of HFSP (Sec. 4.3), we enhanced HFSP by using the notation '@' before a *criterion* to help to indicate the *criterion* to which a design *action* should apply.
2. Warnier Diagram [War76], a diagrammatic representation typically used in hierarchical depictions of the is-part-of relation. (In our applications, they are linearized and shown as text. We also use boldface to indicate those that require further definition).
3. Mathematical set notation. This format is defined as (word in bold font is considered as reserved):

**Criterion A (for X)**

**X set** = { x if P(x) = *True* }

This states: 1) the name of the criterion is A; 2) it will be used to determine the artifacts that are members of the set X; 3) an artifact x is an X artifact if it satisfies P, which is a predicate defining the properties of X.

4. Artifact/Representation table. This table is used to indicate what *representation* an SDM recommends for use in representing an artifact.

---

Figure 5.12: The current version of MF

---

## **A Model of a Design Methodology (MDM)**

1. Artifact Type Definition (ATD)
  - 1.1. Artifact Type Hierarchy (ATH)
  - 1.2. Artifact Composition Hierarchy (ACH)
  - 1.3. Criteria for Designing Artifacts (CDA)
  - 1.4. Representations for Expressing Artifacts (REA)
2. Action Functional Hierarchy (AFH)
  - 2.1. First Level
  - 2.2. Second Level
  - 2.3. ...

Figure 5.13: A template for specification of a model of design methodology

---



Template Element	Definition
MDM	A more formalized description of a design methodology.
ATD	Description of the structure of design artifacts and inter-artifact relations.
ATH	Description of <i>is-a</i> relations among design artifacts.
ACH	Description of <i>is-component-of</i> relations among design artifacts.
CDA	Indication of what criteria should be applied for identifying, establishing and refining design artifacts and descriptions of those criteria.
REA	Indication of what representation (or rendering/medium) should be used to express a design artifact.
AFH	Hierarchical and functional descriptions of activities that are carried out to design software.

Figure 5.14: Definitions of the template components

# Chapter 6

## Experiment 2: Comparison of SDMs

### 6.1 Goals and Design of the Experiment

#### 6.1.1 Goals of the Experiment

In this chapter we describe another experiment we have carried out on CDM. This experiment was aimed at achieving the following goals:

1. Further evaluation and enhancement of CDM. We used CDM to compare more SDMs and compare them in some additional aspects. We expected that in doing this we would be able to identify weaknesses in CDM and to modify CDM accordingly to enhance it. By doing this, we also hoped to further validate CDM—demonstrate that CDM could be used to aid the comparisons of a large variety of SDMs.
2. Evaluation of the BF. We used the BF described in the last chapter to classify the components of SDMs. We checked whether the BF was complete enough to classify a large number of method components and whether the BF allowed

objective classification of the method components. By using CDM that is based on these classifications, we examine how effectively a version of these classifications work. This will be examined from two aspects. The first is how effectively the classifications guide the comparisons. The second is how effectively the classifications are in directly aiding the comparison of SDMs.

3. Evaluation of the MF. We used the MF described in the last chapter to model SDMs, thereby examining the completeness and appropriateness of the MF.

As we have evaluated a version of the MF in the first experiment (Chapter 4), here in this experiment, we emphasize the first two goals.

### 6.1.2 Design of the Experiment

Before starting this experiment, we had to decide which SDMs to compare. We used three criteria to choose SDMs for comparison in this experiment:

1. Some of the SDMs to be compared should be disparate. By doing so, we hope to evaluate CDM in supporting comparison of diverse SDMs.
2. Some of the SDMs to be compared should be similar. By doing so, we hope to evaluate CDM in supporting identification of detailed differences.
3. The SDMs to be compared should be relatively familiar to us. This allows us to finish this experiment in a limited time frame.

Based on these criteria, we chose the following SDM pairs for comparisons. 1) RDM vs. BOOD, 2) JSD vs. SD, 3) DSSD vs. SD (they are disparate—DSSD is data oriented, and SD is function oriented), and 4) LCP vs. DSSD (they are similar—they are both data oriented). In developing the BF and MF, we have

studied all of these SDMs. Thus, it was relatively easy for us to develop their models and compare them.

In this experiment we compared SDMs in some additional aspects to further evaluate CDM. We anticipated that modeling SDMs using the MF would help us to compare SDMs in these additional aspects:

- **Criteria for determining artifact:** the criteria to be applied by a design action to design and determine a certain artifact.
- **Artifact composition:** the structure and recommended contents of a design artifact.
- **Representation applied:** the representation means used to display a particular artifact.

Note that this experiment is aimed at examining CDM in a broader context, thus, the comparisons which use the CDM strategies that have been validated in the first experiment are presented relatively briefly.

The presentation of this experiment consists of three parts. The first part describes the models of the SDMs and the classifications of the method components of these SDMs, showing how the BF and MF have been used. The second part describes the comparisons, including the selection of comparison topics and the comparisons between method components. The third part, based on the first two parts, evaluates the CDM, BF and MF, examining how effectively they aided the comparisons of SDMs.

## 6.2 Step 1: Build Process Models

### 6.2.1 Rational Design Method (RDM)

#### Overview

The Rational Design Methodology (RDM) [PC86, PCW84] describes an approach for documenting and organizing software requirements and design specifications. It elaborates what a requirement or a design should specify and into what structure it should be organized. RDM suggests applying information hiding to help in structuring a design document in order to achieve separation of concerns and to ease making changes in documents.

The first step in using RDM to document a design is to develop the *module guide*, which specifies the structure of the design document. The module guide should be tree-structured where each node represents a design module and describes its responsibility. Children of a node are the components of this node.

The second step in RDM is to develop the interfaces to the design modules. An interface should provide sufficient information for designing the corresponding module implementation and for enabling designers of other modules to use this module. Each module may contain a number of *access programs* that are invocable by the programs of other modules.

The third step in RDM is to develop the *use-hierarchy*. A use-hierarchy could be a matrix where the entry in position(A, B) is true if and only if the correctness

of program A depends on the presence in the system of a correct implementation of program B.

The fourth step in RDM is to implement the design modules. However, before coding a major design, the design decisions should be documented in a document called the *module design document*. This document is designed to allow an efficient review of the design before the coding begins and to be used for maintenance of the implementation.

## Model of RDM

### Artifact Composition Hierarchy

$$\text{Design\_Docu} \left\{ \begin{array}{ll} \text{Module\_Guide} & (1, 1)^1 \\ \text{Module\_Spec} & (1, 1) \\ \text{Module\_Design\_Docu} & (1, 1) \\ \text{Use\_Hierarchy} & (1, 1) \\ \text{Process\_Hierarchy} & (0, 1) \end{array} \right.$$

$$\text{Module\_Guide} \left\{ \text{Module} \quad (1, k) \right.$$

$$\text{Module} \left\{ \begin{array}{ll} \text{name} & (1, 1) \\ \text{desc} & (1, 1) \\ \text{children} & (0, j) \end{array} \right.$$

$$\text{Module\_Spec} \left\{ \text{Interface\_Spec} \quad (1, k) \right.$$

---

<sup>1</sup>(1, k) indicates the lower and upper bounds on the number of occurrences of the part (i.e., *Design\_Docu* should have one and only one *Module\_Guide*).

$$\text{Interface\_Spec} \left\{ \begin{array}{l} \text{name} \quad (1, 1) \\ \text{Spec} \quad (1, k) \end{array} \right.$$

$$\text{Spec} \left\{ \begin{array}{l} \text{dataTypes} \quad (0, i) \\ \text{Programs} \quad (0, n) \\ \text{exceptions} \quad (0, m) \end{array} \right.$$

$$\text{Programs} \left\{ \begin{array}{l} \text{name} \quad (1, 1) \\ \text{desc} \quad (1, 1) \\ \text{inParameters} \quad (0, n) \\ \text{outParameters} \quad (1, m) \\ \text{timing} \quad (0, 1) \\ \text{accuracy} \quad (0, 1) \end{array} \right.$$

$$\text{Use\_Hierarchy} \left\{ \text{Entry} \quad (1, n) \right.$$

## Action Functional Hierarchy

### First Level:

- (a)  $\text{RDM}(\text{Req}, \text{Design} | \text{Req\_Docu}, \text{Design\_Docu}) \Rightarrow$
- (1)  $\text{Develop\_Req\_Document}(\text{Req} | \text{Req\_Docu})$
  - (2)  $\text{Develop\_Module\_Structure}(\text{Design} | \text{Module\_Guide})$
  - (3)  $\text{Specify\_Module\_Interface}(\text{Req\_Docu}, \text{Design}, \text{Module\_Guide} | \text{Module\_Spec})$
  - (4)  $\text{Derive\_Use\_Hierarchy}(\text{Design}, \text{Module\_Guide}, \text{Module\_Spec} | \text{Use\_Hierarchy})$
  - (5)  $\text{Specify\_Module\_Internal\_Structure}(\text{Module\_Spec} | \text{Module\_Design\_Docu})$
  - (6) **Where**  $\text{Design\_Docu} = \text{Is\_Composed\_Of}(\text{Module\_Guide}, \text{Module\_Spec}, \text{Use\_Hierarchy}, \text{Module\_Design\_Docu})$

**Second Level:**

(b) *Develop\_Module\_Structure*(*Design*, @DM|*Module\_Guide*) ⇒

(1) *Identify\_Design\_Secret*(*Design*|*Secret*)

(2) *Develop\_Guide*(*Secret*|*Module\_Guide*);

(c) *Develop\_Module\_Interface*(*Req\_Docu*, *Design*, *Module\_Guide*|*Module\_Spec*) ⇒

(1) *Specify\_Data\_Type*(*Req\_Docu*, *Design*, *Module\_Guide*|*dataTypes*)

(2) *Specify\_Program*(*Req\_Docu*, *Design*, *Module\_Guide*|*Programs*)

(3) *Specify\_Undesired\_Event*(*Req\_Docu*|*exceptions*)

(4) **Where** *Module\_Spec* = *Is\_Composed\_Of*(*dataTypes*, *Programs*, *exceptions*)

## 6.2.2 Booch's Object Oriented Design (BOOD)

### Overview

See section 4.2.1.

### Model of BOOD

#### Artifact Composition Hierarchy

Object	{	<i>Name</i>	(1,1)
		<i>State</i>	(1, <i>i</i> )
		<i>Class</i>	(0,1)
		<b>Operation</b>	(1, <i>k</i> )
		<i>Visibility</i>	(1,1)



$$\text{Operation} \left\{ \begin{array}{ll} \textit{Operation\_def} & (0, j) \\ \textit{Timing} & (0, 1) \\ \textit{Space} & (0, 1) \end{array} \right.$$

$$\text{Visibility} \left\{ \begin{array}{ll} \textit{Object\_See} & (1, n) \\ \textit{Object\_Seen} & (1, m) \end{array} \right.$$

$$\text{Design\_Spec} \left\{ \text{Subsystem} \quad (1, n) \right.$$

$$\text{Subsystem} \left\{ \text{Module} \quad (1, m) \right.$$

$$\text{Module} \left\{ \begin{array}{ll} \textit{Name} & (1, 1) \\ \text{Interface} & (1, 1) \\ \text{Implementation} & (1, k) \end{array} \right.$$

$$\text{Interface} \left\{ \begin{array}{ll} \textit{Attributes} & (1, k) \\ \text{Procedure} & (1, j) \\ \text{Visibility\_Spec} & (0, 1) \end{array} \right.$$

$$\text{Procedure} \left\{ \begin{array}{ll} \textit{Procedure\_Def} & (1, 1) \\ \textit{Time\_Constraint} & (0, 1) \\ \textit{Space\_Constraint} & (0, 1) \end{array} \right.$$

$$\text{Visibility\_Spec} \left\{ \begin{array}{ll} \textit{Module\_See} & (0, i) \\ \textit{Module\_Seen} & (0, j) \end{array} \right.$$

## Criteria for Designing Artifacts

**Criterion IO** (for *Object*):

*Object Set* = {  $x$  If  $P_1(x) \wedge P_2(x) \wedge P_3(x) \wedge P_4(x)$  }

where:

$P_1(x)$ :  $x$  must have state;

$P_2(x)$ :  $x$  must be characterized by actions that it suffers and that it requires of other objects;

$P_3(x)$ :  $x$  is denoted by a name;

$P_4(x)$ :  $x$  has visibilities that are restricted for other objects.

## Representation for Design Artifacts

An object should be represented using the notations defined in the paper [Boo86].

Artifact Name	Representation
<i>Object</i>	Booch's Notation for representing object

## Action Functional Hierarchy

(a)  $\text{BOOD}(\text{Req\_Spec}|\text{Design\_Spec}) \Rightarrow$

- (1) *Identify\_Object*(*Req\_Spec*, @IO|*Objects*, *States*)
- (2) *Identify\_Operations*(*Req\_Spec*, *Objects*, *States*|*Operation*)
- (3) *Establish\_Visibility*(*Req\_Spec*, *Objects*, *States*, *Operation*|*Visibility*)
- (4) *Establish\_Interface*(*Visibility*, *Objects*, *States*, *Operation*|*Subsystem*, *Interface*)
- (5) *Establish\_Implementation*(*Interface*|*Implementation*)
- (6) **Where** *Design\_Spec* = *is\_composed\_of*(*Interface*, *Implementation*);

**Second Level:**

- (b) *Identify\_Object*(*Req\_Spec*, @IO|*Objects*, *States*)  $\Rightarrow$

- (1) *Identify\_Nouns*(*Req\_Spec*|*Nouns*)
- (2) *Identify\_Concrete\_Object*(*Req\_Spec*, *Nouns*, @IO|*Concrete\_Object*)
- (3) *Identify\_Abstract\_Object*(*Req\_Spec*, *Nouns*, @IO|*Abstract\_Object*)
- (4) *Identify\_Server*(*Req\_Spec*, *Nouns*, @IO|*Server*)
- (5) *Identify\_Agent*(*Req\_Spec*, *Nouns*, @IO|*Agent*)
- (6) *Identify\_Actor*(*Req\_Spec*, *Nouns*, @IO|*Actor*)
- (7) *Identify\_Class*(*Req\_Spec*, *Agent*, *Server*, *Actor*, *Concrete\_Object*, *Abstract\_Object*|*Class*)
- (8) *Identify\_Attributes*(*Objects*|*States*)
- (9) **Where** *Objects* = union(*Concrete\_Object*, *Abstract\_Object*, *Class*, *Agent*, *Actor*, *Server*)

(c) *Identify\_Operation*(*Req\_Spec*, *Object*, *States*|*Operation*) ⇒

- (1) *Identify\_Suffered*(*Req\_Spec*, *Object*, *States*|*Operation\_Suffered*)
- (2) *Identify\_Required*(*Req\_Spec*, *Object*, *States*|*Operation\_Required*)
- (3) *Define\_Time\_Order*(*Req\_Spec*, *Operation*|*Time\_Order*)
- (4) *Define\_Space*(*Req\_Spec*, *Operation*|*Space*)
- (5) **Where** *Operation* = union(*Operation\_Suffered*, *Operation\_Required*)

(d) *Establish\_Visibility*(*Req\_Spec*, *Objects*, *States*, *Operation*|*Visibility*) ⇒

- (1) *Specify\_Object\_See*(*Objects*|*Objects\_See*)
- (2) *Specify\_Object\_Seen*(*Objects*|*Object\_Seen*)
- (3) **Where** *Visibility* = union(*Objects\_See*, *Object\_Seen*)

(e) *Establish\_Interface*(*Visibility*, *Object*, *States*, *Operations*|*Subsystem*, *Interface*) ⇒

- (1) *Derive\_Module*(*Object*|*Module*)
- (2) *Specify\_Attr*(*States*, *Module*|*Attributes*)
- (3) *Specify\_Proc*(*Operations*, *Module*|*Procedures*)
- (4) *Specify\_Visibility*(*Visibility*, *Module*|*Visibility\_Spec*)
- (5) **Where** *Subsystem* = *is\_in\_term\_of*(*Module*),

(6) *Interface = is\_composed\_of(Attributes, Procedure, Visibility\_Spec);*

### 6.2.3 Jackson Systems Development (JSD)

#### Overview

See section 4.2.1.

#### Model of JSD

#### Artifact Composition Hierarchy

$$\mathbf{Entity\_Action\_List} \left\{ \begin{array}{l} \mathit{Entity\_Name} \quad (1, k) \\ \mathit{Action\_Name} \quad (1, j) \end{array} \right.$$

$$\mathbf{Action\_Desc} \left\{ \mathit{Action\_Desc\_Entry} \quad (1, j) \right.$$

$$\mathbf{Action\_Desc\_Entry} \left\{ \begin{array}{l} \mathit{Action\_Name} \quad (1, 1) \\ \mathit{Desc\_Text} \quad (1, 1) \\ \mathbf{Entity\_Related} \quad (1, i) \\ \mathbf{Attributes} \quad (1, l) \end{array} \right.$$

...

## Criteria for Designing Artifact

Here, we give two examples of *criteria* definition: 1) for identifying an *Entity* and 2) for identifying an *Action*.

**Criterion IE** (for *Entity*):

*Entity Set* = {  $x$  **If**  $P_1(x) \wedge P_2(x) \wedge P_3(x)$  }

where:

$P_1(x)$ :  $x$  must exist in the real world outside the system;

$P_2(x)$ :  $x$  must perform and suffer actions in a significant time ordering;

$P_3(x)$ :  $x$  must be capable of being regarded as an individual.

**Criterion IA** (for *Action*):

*Action Set* = {  $x$  **If**  $P_1(x) \wedge P_2(x) \wedge P_3(x)$  }

where:

$P_1(x)$ :  $x$  must take place at a point in time;

$P_2(x)$ :  $x$  must take place outside of the system;

$P_3(x)$ :  $x$  must be atomic.

## Representations for Expressing Artifact

Artifact Name	Representation
<i>Real_World_Process</i> <i>Entity_Structure</i> <i>Real_World_Model</i>	<i>Structure_Diagram_Notation</i>
<i>Init_System_Spec_Diagram</i> <i>System_Spec_Diagram</i> <i>Connection</i>	<i>System_Spec_Diagram_Notation</i>
<i>Model_Process</i>	<i>Structure_Text</i>
<i>Function_Process</i>	Not specified
<i>System_Function</i> <i>Timing</i>	<i>Text</i>

## Action Function Hierarchy

### First Level:

- (a)  $JSD(Real\_World|Design\_Spec) \Rightarrow$
- (1)  $Develop\_Spec(Real\_World\_Desc|System\_Spec\_Diagram)$
  - (2)  $Develop\_Impl(System\_Spec\_Diagram|System\_Impl\_Diagram)$
  - (3) **Where**  $Real\_World\_Desc = Interview(Users, Developers, Real\_World),$
  - (4)  $Design\_Spec = union(System\_Spec\_Diagram, System\_Impl\_Diagram);$

### Second Level:

- (b)  $Develop\_Spec(Real\_World\_Desc|System\_Spec\_Diagram) \Rightarrow$
- (1)  $Develop\_System\_Model(Real\_World\_Desc|Init\_System\_Spec\_Diagram)$
  - (2)  $Develop\_System\_Func(Init\_System\_Spec\_Diagram$   
 $|System\_Spec\_Diagram);$

### Third Level:

- (c)  $Develop\_System\_Model(Real\_World\_Desc|Init\_System\_Spec\_Diagram) \Rightarrow$

- (1) *Model\_Reality(Real\_World\_Desc|Real\_World\_Model)*
- (2) *Model\_System(Real\_World\_Model|Init\_System\_Spec\_Diagram);*
- (d) *Develop\_System\_Func(Init\_System\_Spec\_Diagram|System\_Spec\_Diagram) ⇒*
- (1) *Define\_Func(Init\_System\_Spec\_Diagram|System\_Function)*
- (2) *Specify\_Process(Init\_System\_Spec, System\_Function|Function\_Process)*
- (3) *Define\_Timing(Init\_System\_Spec\_Diagram, System\_Function|Timing)*
- (4) **Where** *System\_Spec\_Diagram =*  
*is\_composed\_of(Init\_System\_Spec\_Diagram, System\_Function, Function\_Process, Timing);*

#### **Fourth\_level:**

- (e) *Model\_Reality(Real\_World\_Desc|Real\_World\_Model) ⇒*
- (1) *Identify\_Entity\_Action(Real\_World\_Desc|Entity\_Action\_List)*
- (2) *Specify\_Action(Entity\_Action\_List|Action\_Desc)*
- (3) *Model\_Entity\_Structure(Entity\_Action\_List|Entity\_Structure)*
- (4) **Where** *Real\_World\_Model = is\_in\_terms\_of(Real\_World\_Proc),*
- (5) *Real\_World\_Proc = is(Entity\_Structure);*

- (f) *Model\_System(Real\_World\_Model|Init\_System\_Spec\_Diagram) ⇒*
- (1) *Identify\_Model\_Process(Real\_World\_Proc|M\_Proc\_Name)*
- (2) *Connect(Real\_World\_Proc, M\_Proc\_Name,*  
*Data\_Stream, State\_Vector|Connection)*
- (3) *Specify\_Model\_Process(Connection, Real\_World\_Proc,*  
*M\_Proc\_Name|Model\_Process)*
- (4) **Where** *Init\_System\_Spec\_Diagram =*  
*is\_in\_term\_of(Model\_Process),*
- (5) *Real\_World\_Model = is\_in\_term\_of(Real\_World\_Proc);*

#### **Fifth\_level Decomposition:**

- (g) *Identify\_Entity\_Action(Real\_World\_Desc|Entity\_Action\_List) ⇒*
- (1) *Identify\_Action(Real\_World\_Desc|Action\_List)*

- (2) *Identify\_Entity(Real\_World\_Desc, Action\_List|Entity\_List)*
- (3) **Where** *Entity\_Action\_List = union(Action\_List, Entity\_List);*

**Sixth level Decomposition:**

- (i) *Identify\_Action(Real\_World\_Desc|Action\_List) ⇒*
  - (1) *Identify\_Verb(Real\_World\_Desc|Verbs)*
  - (2) *Select\_Action(Real\_World\_Desc, Verbs, Entity\_List, @IA|Action\_List)*
  - (3) *Specify\_Attributes(Real\_World\_Desc, Action\_List|Action\_List);*
  
- (h) *Identify\_Entity(Real\_World\_Desc, Action\_List|Entity\_List) ⇒*
  - (1) *Identify\_Noun(Real\_World\_Desc|Nouns)*
  - (2) *Select\_Entity(Real\_World\_Desc, Nouns, @IE|Entity\_List);*

## 6.2.4 Structured Design (SD)

### Overview

Structured Design (SD) [SMC74, PJ80] describes methods to be used to model problems and to design the structures of the programs that solve the problems. It is similar to RDM in that both are aimed at producing maintainable programs that are easy to change and understand. However, SD attempts to achieve this by taking a different approach—namely pursuing design of programs that have high-degrees of binding and low-degrees of coupling.

The first step of SD is to model the problem by specifying its data transformation aspect. The model is specified by the Data Flow Diagrams (DFD). A data flow diagram describes how input data are transformed into desired outputs.



The second step of SD is to find the major data stream of the data flow. In this step, one identifies the central processing part of the data flow, namely the part where the data items are most processed as abstractions, rather than as concrete entities directly resulting from input/outputs.

The third step of SD is to derive a first-cut of the program structure based upon the major data flow stream that has previously been identified. The program structure consists of a main module and a number of sub-modules, that are to be called by the main module. These modules are functionally bound together.

The fourth step of SD is to finalize the program by adding the sub-modules that should be non-functionally bound to the program. For example, these are the modules that perform initialization/termination or input/output.

## Model of SD

### Artifact Composition Hierarchy

$$\text{Program\_Structure} \begin{cases} \text{Name} \\ \text{Sub\_Module } (0, i) \end{cases}$$

### Criteria for Designing Artifact

**Criterion IHA** (for Identifying *Highest Abstraction*):

*Highest Abstraction Set* = {  $x$  |  $x \in E$  of DFD  $\wedge$  *Most Logical*(Data( $x$ )) }

where:

1. *DFD*(Data Flow Diagram):  $\langle G, \text{Data} \rangle$ ,
  - 1) where  $G$ : a directed graph  $\langle E, V \rangle$ ;
  - 2)  $\text{Data}(e)$ : a function such that given a edge  $e \in E$ , returns the data item attached to the edge  $e$ ;
2. *MostLogical*( $d$ ): a function that returns true when the data item  $d$  is most logical (it is furthest from the input and output of program.)

## Representations for Expressing Artifact

Artifact Name	Representation
<i>Problem</i>	Natural language
<i>Problem_Structure</i> <i>Data_Flow</i> <i>Major_Data_Stream</i>	Data Flow Diagram notation
<i>Init_Program_Structure</i> <i>Program_Structure</i>	Structure Chart notation

## Action Functional Hierarchy

- (a)  $\text{SD}(\text{Problem} | \text{Design\_Spec}) \Rightarrow$
- (1)  $\text{Model\_Problem}(\text{Problem} | \text{Problem\_Structure})$
  - (2)  $\text{Model\_Data\_Flow}(\text{Problem\_Structure} | \text{Data\_Flow})$
  - (3)  $\text{Identify\_Major\_Stream}(\text{Data\_Flow}, @\text{IHA} | \text{Major\_Data\_Stream})$
  - (4)  $\text{Derive\_Program\_Structure}(\text{Major\_Data\_Stream}, \text{Data\_Flow} | \text{Init\_Program\_Structure})$
  - (5)  $\text{Add\_Modules}(\text{Init\_Program\_Structure}, \text{Problem} | \text{Program\_Structure})$
  - (6) **Where**  $\text{Design\_Spec} = \text{is\_in\_terms\_of}(\text{Program\_Structure})$

## 6.2.5 Data Structured Systems Development (DSSD)

### Overview

Data Structured System Development (DSSD) [Orr77, Han86] provides methods for modeling problems, (especially the structure of the desired outputs from the program) and for designing program hierarchy. It also emphasizes the development of the right software to satisfy the requirements of customers by correctly solving the targeted problem.

The first step of DSSD is to model the structure of the desired outputs as a hierarchy (tree). In this hierarchy, a node is a part of the data structure defined by its parent node. This hierarchy shows what the outputs will look like to an end-user (customer), to ensure getting the right outputs. Thus, this hierarchy should be represented in a notation that is most familiar to an end-user. For example, in designing report generation software, the output hierarchy could be modeled as a template of the report.

The second step of DSSD is, based on the structure of the desired outputs, to model the logical structure of the outputs, the designer's view of the data structure. For example, in designing report generation software, the structure could be a conceptual hierarchical structure of the report.

The third step is to derive the program structure from the desired output structure and logical output structure. The program structure is a hierarchy of procedures, each of which is often directly responsible for generating a certain output in the logical output structure. An execution of an implementation of the program should generate the desired output.

## Model of DSSD

### Artifact Composition Hierarchy

$$\text{Data\_Structure} \left\{ \begin{array}{l} \text{Name} \\ \text{Data\_Items } (0, i) \end{array} \right.$$

$$\text{Data\_Items} \left\{ \begin{array}{l} \text{Name} \\ \text{Data\_Items } (0, i) \end{array} \right.$$

### Criteria for Designing Artifacts

**Criterion IA** (for identifying *Atom*):

*Atom Set* = { *x* If in a Warnier Diagram, *x* has no bracket on its right }

**Criterion IU** (for identifying *Universal*):

*Universal Set* = { *x* If in a Warnier Diagram, *x* has a bracket on its right }

### Representations for Expressing Artifact (REA)

Artifact Name	Representation
<i>Output_Structure</i>	Any notations appropriate for customers
<i>Logical_Output_Structure</i>	Warnier-Orr Diagram notation
<i>Process_Structure</i>	

### Action Functional Hierarchy

**First Level:**

(a) DSSD(*Problem*|*Design\_Spec*) ⇒

- (1) *Sketch\_Problem(Problem|Output\_Structure)*
- (2) *Identify\_Logical\_Output(Output\_Structure|Logical\_Output\_Structure)*
- (3) *Derive\_Process\_Structure(Output\_Structure, Logical\_Output\_Structure, @DPS  
|Process\_Structure)*
- (4) **Where** *Design\_Spec = is\_in\_term\_of(Process\_Structure)*

**Second Level:**

- (b) *Identify\_Logical\_Output(Output\_Structure|Logical\_Output\_Structure) ⇒*
- (1) *Identify\_Atoms(Output\_Structure, @IA|Atom\_List)*
- (2) *Specify\_Frequency(Atom\_List|Frequency)*
- (3) *Universal\_Analysis(Frequency|Occurrences)*
- (4) *Draw\_Diagram(Occurrence, Frequency|Logical\_Output\_Structure)*

## 6.2.6 Logical Construction of Programs (LCP)

### Overview

LCP (Logical Construction of Program) [War76] describes the methods to be used in developing a program based on the structure of the input data.

The first step of LCP is to model the inputs of the program as a hierarchy, in which one must define and note the number of times each element appears in the input hierarchy. This hierarchy should be represented by a Warnier Diagram.

The second step is do the same for the outputs of the program.

The third step is to derive the structure of the program based on the structure of the inputs. To do this one must first identify the types of instructions to be used, and then put them in a specific order: read instructions, preparation and

execution of branches, calculation and output instructions, and finally draw the result of this as a flowchart.

The fourth step is, based on the structure of the outputs, to validate the program structure, ensuring that the program will produce the desired outputs.

## Model of LCP

### Artifact Composition Hierarchy

$$\text{Data\_Structure} \left\{ \begin{array}{l} \text{Name} \\ \text{Data\_Items } (0, i) \end{array} \right.$$

$$\text{Data\_Items} \left\{ \begin{array}{l} \text{Name} \\ \text{Data\_Items } (0, i) \end{array} \right.$$

### Representations for Expressing Artifacts

Artifact Name	Representation
<i>Logical_Input_File</i>	Warnier Diagram notation
<i>Logical_Output_File</i>	
<i>Process_Structure</i>	
<i>Process_Structure</i>	Flowchart notation

### Action Functional Hierarchy

#### First Level:

(a)  $\text{LCP}(\text{Problem}|\text{Design\_Spec}) \Rightarrow$

- (1) *Model\_Input\_Structure(Problem|Logical\_Input\_Structure)*
- (2) *Model\_Output\_Structure(Problem|Logical\_Output\_File)*
- (3) *Derive\_Process\_Structure(Logical\_Input\_File, Logical\_Output\_File|Process\_Structure)*
- (4) **Where** *Design\_Spec = is\_in\_term\_of(Process\_Structure)*

**Second Level:**

- (b) *Derive\_Process\_Structure(Logical\_Input\_File, Logical\_Output\_File|Process\_Structure) ⇒*
- (1) *Derive\_Process(Logical\_Input\_File|Process\_Composition)*
- (2) *Transform\_to\_Flowchart(Process\_Composition, Logical\_Input\_File|Ordered\_Process)*
- (3) *Validate\_Process(Logical\_Output\_File, Ordered\_Process|Process\_Structure)*

### 6.3 Step 2: Classify Components

Having completed the above models, we then used the previously defined BF to classify the method components of the selected SDMs. Fig. 6.1, 6.2, 6.3 and 6.4 show the classifications of the method components under MCTH.

Fig. 6.5, 6.6 and 6.7 show the classifications of the *artifacts* under MSDL. These *artifacts* are the inputs and outputs of the actions (e.g., the SDM models specified in the last section) We believe that, based on the relations between *artifact* and other method component types (e.g., MCTRM[*Concept, Artifact*]), it is straightforward for us to classify the *concepts, representations* and *actions* within MSDL.

Concept Hierarchy		Method Component	
Level-1	Level-2		
Problem		Produce changable program (SD, BOOD, RDM)	
		Manage software project (RDM)	
		Design correct software (JSD, BOOD, LCP, DSSD)	
		Reduce software complexity (SD)	
Principle		Information-hiding (BOOD,RDM)	
		Abstract data type (BOOD)	
		Separation of concerns (RDM)	
		Use data/process connection (DSSD,LCP)	
		Model reality (JSD, BOOD)	
Criterion	Deciding artifact	Decide an object (IO) (BOOD)	
		Decide an operation (BOOD)	
		<i>IE</i> (Deciding an entity (JSD))	
		<i>IA</i> (Deciding an action (JSD))	
		Decide "highest abstraction" (IHA) (SD)	
		Identify atom (DSSD)	
	Deciding structure		Identify universal (DSSD)
			Deciding a decomposition (RDM)
			- simple enough to understand
			- independent implementation
Guideline	Identify artifact	- interface is not likely to change	
		- changes are localized	
		Find a verb to identify an action (JSD)	
		Find a noun to identify an entity (JSD)	
	Deriving artifact		Find a noun to identify an object
			Find a verb to identify an operation (BOOD)
			Find "highest abstraction" in a DFD (SD)
			Derive process structures from output
		Derive logical structures from output (DSSD)	
		Derive process structures from input	
		Derive logical structures from input (LCP)	



Concept Hierarchy (Cont.)		Method Component
Level-1	Level-2	
Guideline (Cont.)	Choosing structure	Describe Module Structure (RDM) - By roles - By secret - By facilities provided
		Define Program Rules (SD) - Match program to problem (SD) - Effect scope is in control scope (SD) - Upper limit of module size (SD) - Write initialization modules (SD) - Minimize duplicated codes (SD) - Isolate dependencies (SD) - Reduce parameters (SD)
Measure	Range	Coupling(SD) Cohesiveness (SD)
	Scale	Coincidental binding (SD) Logical binding (SD) Temporal binding (SD) Communication binding (SD) Sequential binding (SD) Functional binding (SD) Interface complexity (SD) Type of connection (SD) Type of communication (SD)

Table 6.1: *Concepts* classified within MCTH

Artifacts Hierarchy		Method Components
Level-1	Level-2	
Program		<i>Model_Process</i> (JSD)
		<i>Function_Process</i> (JSD)
		<i>Module</i> (SD)
Diagram		<i>Object</i> (BOOD)
		<i>Output_Structure</i> (DSSD)
		<i>Logical_Output_Structure</i> (DSSD)
		<i>Logical_Output_File</i> (LCP)
		<i>Logical_Input_File</i> (LCP)
		<i>Process_Structure</i> (DSSD,LCP)
		<i>Initial_System_Spec_Diagram</i> (JSD)
		<i>System_Spec_Diagram</i> (JSD)
		<i>Function_process</i> (JSD)
		<i>Connection</i> (JSD)
		<i>Entity_Structure</i> (JSD)
Text	Plain Text	<i>System_Function</i> (JSD)
		<i>Real_World_Desc</i> (JSD)
<i>Timing</i> (JSD)		
<i>M_Proc_Name</i> (JSD)		
Templated Text		<i>Visibility</i> (BOOD)
		<i>Module</i> (BOOD)
		<i>Entity_Action_List</i> (JSD)
		<i>Action_Desc_Entry</i> (JSD)
Relation		<i>Module</i> (RDM)
		<i>Abstract_Interface</i> (RDM)
		<i>Is_Composed_Of</i> (RDM)
List		<i>Use_structure</i> (RDM)
		<i>Connection</i> (SD)
		<i>Entity_List</i> (JSD)
List		<i>Action_List</i> (JSD)
		<i>Action_Desc</i> (JSD)

Table 6.2: Artifacts classified under MCTH

Rep. Hierarchy		Method Component
Level-1	Level-2	
Language	Computer language	<i>Structure_Text</i> (JSD)
Diagrammatic Notation		<i>OOD_Notation</i> (BOOD)
		<i>Structure_Diagram_Notation</i> (JSD)
		<i>Sys_Spec_Diagram_Notation</i> (JSD)
		<i>Warnier_Diagram</i> (LCP, DSSD)
		<i>Flow_Chart</i> (LCP)
		<i>Structure_Chart</i> (SD)
		<i>DFD_Notation</i> (SD)

Table 6.3: Representations classified within MCTH

Action Hierarchy		Method Component	
Level-1	Level-2		
Construction	Develop	<i>Develop_Spec</i> (JSD) <i>Develop_Impl</i> (JSD) <i>Develop_System_Model</i> (JSD) <i>Develop_System_Func</i> (JSD)	
		Model	<i>Model_Reality</i> (JSD) <i>Model_System</i> (JSD) <i>Model_Entity_Structure</i> (JSD)
	Specify	<i>Establish_Interface</i> (BOOD) <i>Establish_Implementation</i> (BOOD) <i>Specify_Object_See</i> (BOOD) <i>Specify_Object_Seen</i> (BOOD)	
		<i>Specify_Process</i> (JSD) <i>Connect</i> (JSD) <i>Specify_Attributes</i> (JSD)	
		Define	<i>Define_Time_Order</i> (BOOD) <i>Define_Space</i> (BOOD) <i>Define_Func</i> (JSD) <i>Define_Timing</i> (JSD)
	Identify	<i>Identify_Object</i> (BOOD) <i>Identify_Operation</i> (BOOD) <i>Identify_Concrete_Object</i> (BOOD) <i>Identify_Abstract_Object</i> (BOOD) <i>Identify_Server</i> (BOOD) <i>Identify_Agent</i> (BOOD) <i>Identify_Actor</i> (BOOD) <i>Identify_Class</i> (BOOD) <i>Identify_Attributes</i> (BOOD)	
		<i>Identify_Entity_Action</i> (JSD) <i>Identify_Model_Process</i> (JSD) <i>Identify_Entity</i> (JSD) <i>Identify_Action</i> (JSD) <i>Identify_Noun</i> (JSD, BOOD) <i>Identify_Verb</i> (JSD, BOOD)	
		Select	<i>Select_Entity</i> (JSD) <i>Select_Action</i> (JSD)

Table 6.4: The JSD and BOOD Actions classified within MCTH

MSDL		Method Components
Level-1	Level-2	
Problem	Data	<i>Output_Structure</i> (DSSD)
Model	Model	<i>Logical_Output_File</i> (LCP)
Domain		<i>Logical_Input_File</i> (LCP)
	Trans.	<i>System_Spec_Diagram</i> (JSD)
	Model	<i>Data_Flow</i> (SD)
	Entity	<i>Entity_Structure</i> (JSD)
Model		<i>Entity_List</i> (JSD)
		<i>Action_List</i> (JSD)
		<i>Entity_Action_List</i> (JSD)
		<i>Action_Desc</i> (JSD)
Func.		<i>System_Function</i> (JSD)
Model		<i>Timing</i> (JSD)
		<i>Data_Flow</i> (SD)
		<i>Output_Structure</i> (DSSD)
		<i>Logical_Output_File</i> (LCP)

Table 6.5: Classification of *artifacts* under the Problem Model Domain

MSDL		Method Components
Level-1	Level-2	
Solution Model Domain	Data Model	<i>State</i> (BOOD) <i>Logical_Output_Structure</i> (DSSD) <i>Logical_Output_File</i> (LCP) <i>Logical_Input_File</i> (LCP) <i>State_Vector</i> (JSD) <i>Major_Data_Stream</i> (SD)
	Transformation Model	<i>Operation</i> (BOOD) <i>Process_Structure</i> (DSSD,LCP) <i>Function_Process</i> (JSD) <i>Program_Structure</i> (SD) <i>Data_Flow</i> (SD)
	Communication Model	<i>State_Vector</i> (JSD) <i>Data_Stream</i> (JSD)
	Interface Model	<i>Connection</i> (JSD)
	Entity Model	<i>Object</i> (BOOD) <i>Operation</i> (BOOD) <i>State</i> (BOOD) <i>Model_Process</i> (JSD)

Table 6.6: Classification of the *artifacts* under the Solution Model Domain

Design Life-Cycle			Method Component
Level-1	Level-2	Level-3	
Documentation Domain	Structure		Module_Guide (RDM)
	Document		
	Module		<i>Module_Spec</i> (RDM)
	Document		<i>Interface</i> (BOOD)
	Relation	Use-relation	<i>Use_Hierarchy</i> (RDM)
Document	Visibility	<i>Visibility_Spec</i> (BOOD)	

Table 6.7: Classification of the artifacts under the Document Model Domain

## 6.4 Comparison of BOOD with RDM

### 6.4.1 Step 3: Select Comparison Topics

From Figure 6.1 (i.e., the parts of Problem and Principle), we can see that

- Both BOOD and RDM are aimed at designing easily changed program by applying the principle of information-hiding.
- RDM is also aimed at helping the management of a software project. It suggests the importance of separating the concerns of different designers.

- BOOD is aimed at designing a right software system by emphasizing the modeling of the environment under which the system will be operated. RDM does not explicitly emphasize this.

From the functional classifications (Fig. 6.5, 6.6 and 6.7), we can see that RDM primarily supports the process of documenting a design. It does not provide methods for modeling either the problems or the system (i.e., solution). To summarize their differences in documenting a design, we can see from Fig 6.7 that:

- BOOD does not address issues related to developing the Structure Documents. In contrast, *Module\_Guide* of RDM addresses these issues.
- BOOD and RDM both address the issues related to developing the Module Documents. BOOD does this by using the notion of *Interface* while RDM does it by using *Module\_Spec*. Thus, we will compare *Interface* with *Module\_Spec*.
- BOOD and RDM both provide notions for documenting certain relations among modules.



## 6.4.2 Step 4: Compare Method Components

### Comparisons in the Documentation Domain

#### (i) Compare *Interface* with *Module\_Spec*

As our strategy is to focus on comparing the functions of design actions, we compare the actions *Establish\_Interface* of BOOD with *Develop\_Module\_Interface* of RDM.

- **Differences in inter-artifact dependency:** *Establish\_Interface* has four inputs:

- *Object*;
- *States*;
- *Operations*;
- *Visibility*;

*Develop\_Module\_Interface* has three inputs:

- *Req\_Docu*;
- *Design*;
- *Module\_Guide*.

By analyzing those inputs and their compositions, we can see that *Interface* of BOOD is established based on well-defined artifacts. *Visibility* should

help in deciding which *Operations* or *States* should be selected and specified into the *Interface*. In contrast, *Module\_Spec* of RDM depends on *Module\_Guide* which provides overall descriptions of the modules. However the contents of such descriptions are not well defined in RDM. Therefore, generally, it is not clear how precisely and rigorously *Module\_Guide* can guide defining *Module\_Spec*. In addition, unlike BOOD, visibility information will not be available for specifying *Module\_Spec* in RDM.

- **Differences in human involvement:** Since *Object* of BOOD specifies *State* and *Operations*, establishing *Interface* of *Object* should require only deciding which of those should be in the interface and which formalism should be used to specify them. Thus, *Establishing\_Interface* of BOOD is a guided human process. Based on the analyses of the artifacts that a *Module* depends upon, we think that *Develop\_Module\_Interface* of RDM is also a guided human process (i.e., guided by the corresponding descriptions given in the *Module\_Guide*.). However, since the contents of *Module\_Guide* is not well defined, it is not clear how well *Module\_Guide* can guide *Develop Module Interface*.
- **Differences in scope:** *Interface* of an *Object* defines the interface of the abstract data type. Similarly, *Module\_Spec* of RDM is also used to achieve information hiding. However, *Module\_Spec* is defined as a template that contains many optional fields to accommodate various needs in documenting

a design. Some of them are beyond the basic concepts of abstract data type, (e.g., one field may contain the specification of undesired events (i.e., *exceptions*)). Thus, RDM allows to address a broader scope of issues in specifying the interface.

(ii) Compare *Object* with *Module\_Guide*

Though BOOD does not explicitly provide a strategy for structuring a design, it suggests that an object oriented design should be organized according to abstraction levels which are expressed through objects. In this sense, we make some comparisons between BOOD's *Object* and RDM's *Module\_Guide*. To further evaluate CDM, we compare the *criteria* for deciding an object and a module, and the *representation* that they use.

- **Difference in criteria for determining artifact:** By comparing the models of BOOD and RDM, we found that BOOD provides very concrete criteria for determining what an object is (e.g., must have a name, state, ...). In contrast, RDM suggests a set of criteria and guidelines which are more general and intuitive (See Fig. 6.1, where they are described). As our modeling formalism is still limited, we cannot formally model them yet.
- **Differences in representation applied:** By checking Fig. 6.3, we find that BOOD suggests using a diagrammatic notation to describe the document structure while RDM does not suggest any notation.

### 6.4.3 Step 5: Summarize Differences

Table 6.8 summarizes the differences between the RDM and BOOD components. Based on this summary, we have the following further observations:

- RDM is more a collection of software design principles than a well-defined SDM. The reasons are 1) many of its artifacts and actions are not explicit and well-defined, and 2) the guidelines or criteria in RDM are defined rather intuitively, (e.g., RDM suggests that a module interface should enable a software person to understand the module without reading its internal implementation details). This is easy to understand but it is not a very useful prescription for aiding a designer in achieving this.
- RDM supports documentation of design. It describes a number of criteria for deciding what constitutes an acceptably sound and complete design document. However, it fails to clearly describe the needed inputs to RDM. This argument is based upon the observation that RDM starts with documenting requirement/design. Thus, different inputs (e.g., different solution model artifacts), may cause many detailed portions of RDM to vary significantly.

Comparisons in the Documentation Domain				
Comparisons of the Structure Document				
Component	Dependency	Scope	Need for human	Proc.
<i>Module_Guide</i>	<i>Design</i>	Characterize is-	Unspecified	N/A
vs.		component-of structure.		
?				N/A
Comparisons in the Module Document				
<i>Module_Spec</i> (RDM)	<i>Module_Guide</i> <i>Req_Docu, Design</i> which are not well-defined	Loosely defined: It could contain only a set of data types or a set of functions; It may specify undesired events;	Guided human process, the degree of need for human activity may vary greatly	N/A
vs.				
<i>Module_Interface</i> (BOOD)	<i>Object, Operation, etc.</i> which are well defined	Must be for an ADT	Guided human process	N/A
Comparisons in the Relation Document				
<i>Use_Hierarchy</i>	<i>Design, Module_Spec</i>	use relation	Mechanical	N/A
vs.				
<i>Visibility_Spec</i>	<i>Visibility</i>	Potential use-relation	Guided human process	N/A

Table 6.8: Summary of the differences between the RDM and BOOD components

## 6.5 Comparison of JSD with SD

### 6.5.1 Step 3: Select Comparison Topics

From Fig. 6.1 (i.e., Problem, Principle), we can see that:

- SD focuses on reducing program complexity and producing changeable programs. Accordingly, it suggests how to develop a program that has a high degree of cohesiveness and a low degree of coupling.
- JSD focuses on the development of a correct and stable software system that satisfies its specified requirements. Thus, it suggests first modeling the environment that uses the system before designing the system.

From the functional classifications (Fig. 6.5, 6.6 and 6.7), we see that JSD and SD both address issues involved in modeling the problem and its solution.

In modeling the problem (see Fig. 6.5), we found:

- Both JSD and SD by themselves do not address issues involved in modeling data structures<sup>2</sup>.
- SD does not address issues involved in developing the Entity Model of the problem.

---

<sup>2</sup>JSD suggests using relational, network and other data models to model data structures.

- Both JSD and SD address issues involved in developing the Function Model and the Transformation Model. SD addresses those issues through specifying data flow while JSD does this through specifying *System\_Function* and *System\_Spec\_Diagram*. Thus, we should compare *Data\_Flow* with *System\_Function* and *System\_Spec\_Diagram*.

In modeling the solution (Fig. 6.6), we found:

- Both JSD and SD require identification of some important data in a design. SD requires identification of *Major\_Data\_Stream* and JSD requires identification of *State\_Vector*.
- SD does not explicitly address issues involved in modeling the Interface and Communication Model. In contrast, JSD explicitly addresses those issues.
- SD provides no method for helping to develop an Entity Model. In contrast, *Model\_Process* of JSD is aimed at addressing this issue.
- Both JSD and SD address issues related to specifying the Transformation Model. SD addresses these issues through developing the *Program\_Structure* while JSD does this through specification of *Function\_Process*. This indicates that we should compare *Program\_Structure* with *Function\_Process*.

## 6.5.2 Step 4: Compare Method Components

### Comparisons in the Problem Model Domain

#### (i) Compare *Data\_Flow* with *System\_Spec\_Diagram*

Our strategy is still to compare the corresponding design actions to understand the differences between the artifacts they produce. The design actions to be compared are *Model\_Data\_Flow* and *Develop\_Spec* (defined in Sec. 5.5.1).

- **Differences in inter-artifact dependency:** these two artifacts depend on some similar artifacts: the informal descriptions of the problems (*Real World Desc* and *Problem\_Structure*).
- **Differences in scope:** the issues they address, from the viewpoint of modeling data transformation, are also very similar—both are capable of modeling data flow. However, *System\_Spec\_Diagram* is also able to specify the mechanism (*State\_Vector* or *Data\_Stream*) via which data are transferred (see *Model\_System*). In addition, *System\_Spec\_Diagram* can explicitly indicate the boundaries of the real-world (i.e., *Entity\_Structure*), the system interface (i.e., *Model\_Processes*) and internal implementations (i.e., *Function\_Processes*). Thus, a *System\_Spec\_Diagram* can explicitly indicate interactions between the events in the real-world, the system interface and system internal implementation.



- **Differences in development procedure:** the procedures that JSD and SD suggest for modeling data flows are quite different. Generally speaking, JSD takes a breadth-first approach while SD takes a depth-first approach. The breadth-first nature of the process that JSD suggests is exhibited in *Model\_System*. *Model\_System* suggest modeling all data flows between the *Real\_World\_Processes* and *Model\_Processes* first. It is only after *Model\_System* is complete that one then uses *System\_Function*, to specify a complete data flow from *Model\_Process* to the process producing the *System\_Function*. In contrast, SD does not provide any explicit guidance for how to do the data flow modeling. However, SDM seems to suggest that data flow be modeled by first considering a desired output, then identifying all its required inputs and then going further to model the data flow until reaching the process that produces the desired output. This is inherently a depth-first process.
- **Differences in human involvement:** SD provides no guideline for modeling data flow. In contrast, JSD provides guidelines for the process of defining *Init\_System\_Spec\_Diagram* which is a part of the data flow model (*System\_Spec\_Diagram*). However, similarly to SD, JSD provides no method to model the data flows that start from the *Model\_Processes* and proceed to the *System\_Functions*. Based on these observations, we conclude that they are both essentially human processes.

- **Differences in the representations applied:** In SD, a *Data\_Flow* should be specified in Data Flow Diagram notation. In JSD, a *System\_Spec\_Diagram* should be specified in the System Spec Diagram Notation.

(ii) Compare *Data\_Flow* with *System\_Function*

We compare *Model\_Data\_Flow* with *Define\_Func* to see the differences between the artifacts *Data\_Flow* and *System\_Function* in the context of developing the Functional Model.

- **Differences in inter-artifact dependence:** They depend on similar artifacts, namely the customer's requirements (*Problem\_Structure*). However, *System\_Functions* additionally depends on *Init\_System\_Diagrams*, a system interface description. This provides more assistance for both customers and designers in deciding and understanding the requirements.
- **Differences in development procedures:** The JSD model (Sec. 5.5.1) illustrates very clearly that a *System\_Function* is not to be defined until *Real\_World\_Process* and *Model\_Process* are defined. Thus, the events upon which a *System\_Function* is to be performed can be specified in terms of the *Real\_World\_Process*. SD does not explicitly specify the order in which a data flow diagram is to be drawn.
- **Differences in scope:** No significant difference.

- **Differences in human involvement:** JSD and SD both give no guidelines for specifying the system outputs. However, since a *System\_Function* is to be specified after *Real\_World\_Process* and *Model\_Process* are modeled, both designers and customers should have a better sense about what functions they can expect.
- **Differences in the representation applied:** They both do not suggest any notations for specifying system functions.

### Comparisons in the Solution Model Domain

#### (iii) Compare *Program\_Structure* with *Function\_Process*

Both *Program\_Structure* and *Function\_Process* describe the data flows through the system. However, *Program\_Structure* is a hierarchy of the functions processing the *Data\_Flow*. *Function\_Process* is still a direct description of the data flow. We compare the actions (i.e., the SD model(1)-(5) with the JSD model(d)(2)) producing those two artifacts to understand their differences.

- **Differences in inter-artifact dependency:** A *Program\_Structure* is derived from a *Data\_Flow*. Defining a *Function\_Process* depends on the *Init\_System\_Spec\_Diagram* (which describes system interface and its connections with the real-world) and *System\_Function*. From analyzing the way in which it is suggested that *Program\_Structure* be derived, we see that

*Program\_Structure* is a functional structure that processes the *Data\_Flow*.

In contrast, *Function\_Process* is still a direct description of the data flow.

- **Differences in the representation applied:** Based on the models of JSD and SD, we can see that *Program\_Structure* uses Structure Chart notations while JSD does not suggest any notation for specifying a *Function\_Process*.

Since there are such major differences in the way in which they support modeling data flow, we do not think that comparing other aspects is very worthwhile.

However, we would like to comment on *Program\_Structure* (Comments on *Function\_Process* can be found in the comparison between *Function\_Process* and *Operation of BOOD*).

- **Human involvement:** deriving *Program\_Structure* is done by applying the guideline for identifying highest points of abstraction. Thus, this is a guided human process.
- **Scope:** addresses the issue on how to develop a function structure (i.e., functional-call hierarchy) to process the data flow.

### 6.5.3 Step 5: Summarize Differences

Table 6.9 summarizes the differences between JSD and SD. Based on these differences, we make the following observations:

- SD supports modeling a data flow that is to achieve one, or perhaps a few functions. It appears likely that it would not be very easy to use for modeling a data flow that has many outputs.
- SD's support for deriving a functional hierarchy that processes the data flow is quite unique, and it should be useful for designing a *program*. For completely modeling a large scale and complex system, SD seems to be weak compared with JSD.
- This comparison illustrates that JSD can be viewed as providing a method for developing data flow. JSD has been characterized as a data-oriented design methodology and as being similar to object-oriented design methodologies. However, its similarities with SD have never been identified and addressed clearly. Our findings seem to be very valuable in aiding the integration of the two SDMs.

## 6.6 Compare DSSD with SD

### 6.6.1 Step 3: Select Comparison Topics

From Fig. 6.1, we can see that:

- DSSD focuses on designing a concrete software system that satisfies the specified requirements.

Comparisons in the Problem Model Domain				
Component	Dependency	Scope	Need for human	Proc.
<b>Overall differences</b>				
<i>Data.Flow</i> vs. <i>Entity.Structure</i>	<i>Problem</i>	Define data transformation	Unspecified human process	N/A
	<i>Real.World.Desc</i>	Define components outside the system and their behaviors	Guided human process	N/A
<b>Comparisons in the Transformation Model</b>				
<i>Data.Flow</i> vs. <i>System.Spec.D.</i>	<i>Problem</i>	Describe data flow for producing a few functions	Unspecified human process	To be done in depth-first manner
	<i>Real.World.Desc</i>	Describe data flows of whole system	Guided human process	Part of it to be done in breadth-first manner
<b>Comparisons in the Functional Model</b>				
<i>Data.Flow</i> vs. <i>System.Function</i>	<i>Problem</i>	Describe the outputs produced from input	Unspecified human process	During defining problem
	<i>Real.World.Desc</i> <i>Init.System.Spec.D.</i>	Describe the outputs to be produced upon executing actions	Unspecified human process,	After specifying the real-world model and interface
<b>Comparisons in the Solution Model Domain</b>				
<b>Comparisons in the Transformation Model</b>				
<i>Program.Structure</i> vs. <i>Function.Process</i>	<i>Data.Flow</i>	Describe function hierarchy processing a data flow	Guided human process	N/A
	<i>Init.Sys.Spec.D.</i>	Describe data flows	Not well guided	N/A

Table 6.9: Summary of the differences between the JSD and SD components

- SD focuses on reducing program complexity and producing easily changeable programs. Thus, it suggests how to develop programs that have a high degree of cohesiveness and a low degree of coupling.

From the functional classification for specifying the Problem Model (Fig. 6.5), we can observe the following:

- DSSD addresses issues related to modeling the data structures of the outputs of the system. In contrast, SD does not address these issues.
- SD addresses issues concerned with specifying the Transformation Model. In contrast, DSSD does not.
- Both DSSD and SD address issues concerned with developing Function Models. DSSD does this through specifying the data structures of the outputs (i.e., through specifying *Output\_Structure*). SD does this through specifying the data flow reaching that output (i.e., through specifying *Data\_Flow*). Thus, we should compare *Output\_Structure* with *Data\_Flow*.

From the functional classification for specifying the Solution Model 6.6, we observe the following:

- DSSD addresses issues concerned with modeling the data structures for realizing the system outputs. In contrast, SD does not. However, SD requires the identification of the internal data needed to realize the system, (i.e., identification of *Major\_Data\_Stream*).

- Both DSSD and SD address issues concerned with specifying the Transformation Model. DSSD does this through specifying *Process\_Structure* while SD does this through modeling *Data\_Flow* and *Program\_Structure*.

## 6.6.2 Step 4: Comparing Method Components

### Comparisons of the Problem Model Domain

#### (i) Comparing *Data\_Flow* with *Output\_Structure*

From the view of addressing issues concerned with specification of the Function Model, we can see that there are:

- **Differences in scope:** *Output\_Structure* describes the structures of outputs. *Data\_Flow* describes what outputs will be expected, given specified inputs. *Data\_Flow* will not describe the structures of these outputs.
- **Differences in inter-artifact dependency:** There are not many differences. The specifications of *Data\_Flow* and *Output\_Structure* both depend on the requirements or on the customer's needs.
- **Differences in human involvement:** There are not many differences here either. Both require significant human involvement.



## Comparisons in the Solution Model Domain

### (i) Compare *Major\_Data\_Stream* with *Logical\_Output*

We note the following differences in developing the Data Model:

- **Differences in inter-artifact dependency:** *Major\_Data\_Stream* depends on *Data\_Flow*. *Logical\_Output* depends on *Output\_Structure*. The difference is that *Logical\_Output* depends solely on the output structure while *Major\_Data\_Stream* depends on the outputs as well as the inputs to the system.
- **Differences in scope:** The issues they address are similar in that both aim at identifying the structures of internal data (both inputs and outputs) which are needed to realize the system (or to perform the desired functions). *Logical\_Output* is an internal output. Analyzing this output in DSSD helps to identify the needed inputs. They both help in achieving functional and communicational binding. (Note that the data flow for processing the internal data is functionally bound while the processes responsible for reading and writing those internal data are communicationally bound)
- **Differences in human involvement:** They both are human processes. SD provides criteria that help in distinguishing *Major\_Data\_Stream* from *Data\_Flow*. DSSD describes, in reasonable detail, how to derive and distinguish *Logical\_Output* from *Output\_Structure*.

(ii) Compare *Data\_Flow* vs *Process\_Structure*

In analyzing processes for creating the Transformation Model, we observed the following differences:

- **Differences in inter-artifact dependency:** *Data\_Flow* depends on *Problem*. *Process\_Structure* depends on *Logical\_Output\_Structure* and *Output\_Structure*. Thus, *Process\_Structure* depends on better defined artifacts.
- **Differences in scope:** There were few differences.
- **Differences in human involvement:** They are both human processes. However, *Process\_Structure* can be derived from *Logical\_Output\_Structure*.

(iii) Compare *Program\_Structure* vs *Process\_Structure*

- **Differences in inter-artifact dependency:** *Program\_Structure* depends on *Data\_Flow* and *Major\_Data\_Stream*. *Process\_Structure* depends on *Logical\_Output\_Structure* and *Output\_Structure*.
- **Difference in scope:** Developing *Program\_Structure* entails deriving a *functional* decomposition from a *Data\_Flow*. Developing *Process\_Structure* is aimed at producing a hierarchical *procedural* program.
- **Differences in human involvement:** They both are guided human processes. However, in both cases the guidelines are very clear and concrete. Thus these development processes are close to mechanical processes.

### 6.6.3 Step 5: Summarize Differences

Table 6.10 summarizes the differences between SD and DSSD. Based on this summary, we can make the following observations:

- DSSD makes a clear distinction between modeling *what* the system is to produce and *how* the system produces it. In contrast, SD does not make this clear distinction. *Data\_Flow* describes both a process and its product (see our classification (Fig. 6.5) and summary (Table 6.10), where *Data\_Flow* is considered as supporting both the Function Model and Transformation Model).
- In DSSD, a design starts with application of the structure of the output. With this output structure, the required inputs are then identified. The process of modeling *Data\_Flow* is more arbitrary; it could start either from identifying inputs or from identifying outputs.
- It is interesting to note that both SD and DSSD incorporate mechanisms for determining program components that are really responsible for achieving various functions of the program. The approaches used by SD and DSSD are similar; they suggest that the designer distinguish logical data (SD looks for highest points of abstraction, DSSD looks for the *Logical\_Output\_Structure*) from physical data.

Comparisons under the Problem Model Domain				
Comparisons under the Function Model				
Component	Dependency	Scope	Need for human	Procedure
<i>Data_Flow</i> vs. <i>Output_Structure</i>	<i>Problem</i>	Describe inputs to system and outputs from system	Unspecified	N/A
	<i>Problem</i>	Describe data structure of the system output	Unspecified	N/A
Comparisons under the Solution Model Domain				
Comparisons under the Data Model				
Component	Dependency	Scope	Need for human	Procedure
<i>Major_Data_Stream</i> vs. <i>Logical_Output_Str.</i>	<i>Data_Flow</i>	To find the central processing part of <i>Data_Flow</i>	Partially guided	N/A
	<i>Output_Structure</i>	To find the functions that read the needed inputs	Guided	N/A
Comparisons under the Transformation Model				
<i>Data_Flow</i> vs. <i>Process_Structure</i>	<i>Problem</i>	How the output is produced from input	Unspecified	During modeling problem
	<i>Output_Structure</i> <i>Logical_Output_Str.</i>	How the output is procedurally produced	Guided or mechanical	After output structure is specified
<i>Program_Structure</i> vs. <i>Process_Structure</i>	<i>Data_Flow</i>	Functional decomposition	Guided	N/A
	see above	hierarchical procedural program	Guided or mechanical	N/A

Table 6.10: Summary of the differences between SD and DSSD

## 6.7 Compare LCP with DSSD

By analyzing Fig. 6.1 (i.e., Problems and Principles), we see that:

- Both DSSD and LCP aim to develop a correct software system that satisfies its given requirements.
- Both DSSD and LCP recognize the close relation between data structure and program structure, and use this relation as the basis for their other strategies.

From the classification under the Problem Model Domain (Fig. 6.5), we find the following:

- LCP and DSSD both address issues in modeling the structure of the output data. LCP uses *Logical\_Output\_File* whereas DSSD uses *Output\_Structure*.
- Neither LCP nor DSSD addresses either the Transformation Model or the Entity Model.

From the classification under the Solution Model Domain (Fig. 6.6), we can infer:

- Both LCP and DSSD address issues in modeling the data structures needed for realizing the system. LCP uses *Logical\_Input\_File* and *Logical\_Output\_File*. DSSD uses *Logical\_Output\_Structure*.
- Both LCP and DSSD address issues in creating the Transformation Model.

## 6.7.1 Step 4: Comparing Design Methodologies

### Comparisons under the Problem Model Domain

We compared *Logical\_Output\_File* of LCP with *Output\_Structure* of DSSD, and found that they are same. They are both specified as a hierarchal data structure. They both are used to model the outputs of the system as the needs of customers. Neither LCP nor DSSD provides detailed guidelines for how to develop those two artifacts.

### Comparisons under the Solution Model Domain

#### Compare *Process\_Structure*(LCP) with *Process\_Structure*(DSSD)

- **Differences in inter-artifact dependency:** From the DSSD and LCP models, we can see that deriving a *Process\_Structure* in DSSD requires only the modeling of the output data structure—*Logical\_Output*. However, deriving *Process\_Structure* in LCP requires the modeling of both input and output—*Logical\_Input\_File* and *Logical\_Output\_File*.
- **Differences in human involvement:** they both provide rules for deriving a *Process\_Structure*.

- **Differences in scope:** Not much. *Process\_Structure* (for both LCP and DSSD) is the architecture for a procedural program that produces the desired outputs.
- **Differences in representations applied:** LCP suggests that *Process\_Structure* be specified in *Flow\_Chart* notation. DSSD suggests the use of the Warnier-Orr Diagram notation, an extended version of the Warnier Diagram notation.

### 6.7.2 Step 5: Summarize Differences

Table 6.11 summarizes the differences between LCP and DSSD. Based on the summary, and the models of LCP and DSSD, we have the following observations:

- LCP assumes that designers know the inputs and outputs of the program to be designed. Thus, it suggests using the modeling of those two artifacts to help the development of the program. In contrast, DSSD assumes that designers know only the outputs of the program to be designed. Thus, it suggests using the modeling of the output to identify the inputs and to construct the program. This understanding should be helpful in deciding which of these two SDMs to use.

Comparisons under the Solution Model Domain				
Comparisons under the Transformation Model				
Component	Dependency	Scope	Need for human	Rep.
<i>Process_Structure</i> (LCP) vs.	<i>Logical_Input_File</i> <i>Logical_Output_File</i>	How the output is produced from input	Guided human process	<i>Flowchart</i>
<i>Process_Structure</i> (DSSD)	<i>Output_Structure</i> <i>Logical_Output_Structure</i>	How the output is produced	Guided human process	<i>Warnier – Orr</i>

Table 6.11: Summary of the differences between LCP and DSSD components

## 6.8 Evaluation

To demonstrate our BF and MF evolutionary development strategy (Fig. 5.4) and to examine the validity of our current BF and MF, we should evaluate both the BF and MF by using that strategy. In Sec. 4.3.1 we have already evaluated an earlier version of the MF by using the BF (i.e., we indicated which of the types and relations defined in MCTH could and could not be supported by that version of the MF (i.e., HFSP)). Thus, here we focus only on the evaluation of the BF. In pursuit of our first goal (Sec. 2.1), we evaluate the BF from the following perspectives: 1) *Sufficiency/effectiveness*: does the BF sufficiently and effectively support assessments of the SDMs? 2) *Objectivity*: how objective are the classifications within the BF? 3) *Completeness*: can all the major method components of the SDMs be classified within the BF?



### 6.8.1 How to Evaluate

As the method components of an SDM are often not explicitly indicated, it is difficult to decide if the BF is complete enough to classify all the components. To deal with this we use a model of the SDM to aid in making such decision. In this way we hope to determine whether any component in the model has not been classified within the BF (although the model cannot be used to verify whether or not all components have been classified). By developing more mature SDM models and classifying more method components, we expect to gain more confidence in the completeness of the BF.

Note that the evaluation of *sufficiency/effectiveness* derives from the analysis of the comparisons while the evaluation of the *objectivity* and *completeness* derives from the BF and SDMs themselves.

### 6.8.2 Evaluation of the Type Framework (MCTH)

#### Evaluation of Sufficiency/Effectiveness

The sufficiency/effectiveness of MCTH should be evaluated against its goals:

- 1) guiding comparisons and
- 2) aiding assessments of SDMs.

Our earlier experiment (Chapter 4) seems to have indicated that the top-level types of MCTH are quite effective in guiding comparisons. In the experiment, we

have used Fig. 6.1 to directly aid in making comparisons. For example, we analyzed the Problems at which BOOD and RDM are aimed, and Principles they used, to understand the differences between BOOD and RDM. In the other comparisons carried out as part of this experiment, we have done similar analyses (e.g., see Sec. 6.3.1 and 6.4.1). These all show that the classification within *concept* is useful in aiding comparisons.

As we expected, this experiment shows that some of the other decompositions, particularly of *artifact* and *action*, do not seem to help nearly as much in guiding the comparisons. They do, however, help in aiding assessment of SDMs. For example, in most cases, it is reasonable to compare two *artifacts* (e.g., *Model\_Process* and *Object*) even two that have different subtypes (e.g., Program and Diagram). This can help to show explicit differences arising from differences in formality used in defining these two artifacts.

The classifications under MCTH (e.g., Fig. 6.1) also directly aid assessment of the SDMs. For example, the classifications under *problem* and *principle* help to assess their *user-orientation* [Bra83]; JSD's and BOOD's emphases on *Modeling reality* might be used as the basis for the conclusion that they are better oriented to users.

## Evaluation of Objectivity

In classifying components, we had no difficulty in deciding the degree of formality with which an *artifact* is defined and the level of formality a *representation* supports. Thus, we believe that we can classify *artifacts* and *representations* within MCTH in a highly objective manner.

One difficulty we encountered in classifying *concepts* was deciding if a *concept* should be a *principle* or a *guideline*. We decided that a *guideline* usually covers a narrow range of issues and is often more concrete than a *principle*. However, this is sometimes hard to decide objectively. For example, we decided fairly easily that *Separation of concern* is a *principle* but *Find a verb to identify an action* is a *guideline*. However, it took us more thought to decide that *Describe module structure* (e.g., *By facilities provided*) is a *guideline* rather than a *principle*.

We found that a *criterion* often can be relatively easily distinguished from a *guideline* because a guideline describes how to reach a goal whereas a *criterion* is used to determine if the goal has been reached. For example, we decided fairly easily that the RDM module decomposition rules (e.g., *simple enough to understand*) should be *criteria* rather than *guidelines*.

It is easy to distinguish an *identify* action from a *select* action since a *select* action requires that candidates be available. However, it is fairly difficult to decide whether an action is a *specify* action or a *define* action. We decided that they

differ mainly in that the latter more completely describes the property of a thing in a declarative manner. Again, it is hard to decide objectively whether an *action* should be a *model* action or a *specify* action because their semantics are very close.

We believe that these difficulties result from the complex nature of the design activity and the current state of the art of SDM development as well. To cope with them, we could define these BF items more precisely. However, we believe that cooperation from SDM advocates would be even more effective. We believe it is important for SDM advocates to define SDMs more precisely and to indicate more explicitly the nature of a method component and the roles it plays. We believe that our work makes an important contribution, not simply in pointing out the need for more precision in SDM description, but rather in pointing out specifically where greater precision is most needed.

### **Evaluation of Completeness**

The SDM models we have built (e.g., the JSD model defined in Sec. 5.5.1) indicate that MCTH is a suitable vehicle for classifying all the method components we have made explicit (the italic identifiers) in these models. For example, *criteria IE* and *IA* modeled in Sec. 5.5.1 are classified within Fig. 6.1; all *representations*, *Structure\_Diagram\_Notation*, *System\_Spec\_Diagram\_Notation*, and *Structure\_Text* in the JSD model are classified within Fig. 6.3.

Similarly, we evaluated the completeness of classifications of the *artifacts* and *actions* (Fig. 6.2 and 6.4) using our models (e.g., the JSD model specified in HFSP (Sec. 5.5.1)). We checked to see if all the inputs and outputs of the design actions and those actions themselves were classified or not, and found that all the components explicitly indicated in the models were successfully classified.

### 6.8.3 Evaluation of the Function Framework (MSDL)

#### Evaluation of Sufficiency/Effectiveness

Our earlier experiment (see Chapter 4) seems to have indicated that a portion of MSDL (i.e., *Solution Model*) is quite useful in guiding comparisons. Our subsequent experiment indicates that, in fact the whole MSDL is useful for guiding comparisons.

For example, the comparison of RDM and BOOD indicates that classification under the Documentation Domain can be used to guide the comparison of the method components that address design documentation issues. The comparison of JSD and SD indicates that classification under the Problem Model Domain can be used to guide comparisons of the method components that address issues about modeling application problems.

In this experiment, we also found another advantage of using the MSDL—namely that it can help a analyst to focus the comparisons. For example, in the comparison between JSD and SD, we compared *Data\_Flow* of SD with *System\_Function* of JSD under the Function Model of the Problem Model. This model helped us to focus on comparing the most critical and telling aspect defined within the Function Model—namely input/output.

We now indicate how the MSDL classifications (e.g., Fig. 6.5) might also help in making assessments of SDMs.

Taking Kung's feature framework [Kun83] (which includes *understandability*, *expressiveness*, *processing independence*, *checkability concerns*, and *changeability*) as a testbed, we conclude that our classifications can be used to aid assessments of at least the *understandability* and *expressiveness* aspects of SDMs. For *understandability*, the classifications help by identifying what *artifacts* (e.g., *Entity\_Structure*) and *representations* (e.g., *Structure\_Diagram\_Notation*) an SDM (e.g., JSD) suggests for use in describing a problem. Based on previous assessment of the understandability of these *artifacts* and *representations*, one can then assess how effectively the SDM supports *understandability*. For *expressiveness*, the classification helps by facilitating the identification of the aspects (e.g., transformation and data) of a problem or system that an SDM can model, and what modeling methods (e.g., data flow, or relational) it provides. This can then directly aid the assessment of the *expressiveness* of the overall SDM itself.

The current MSDDL may need to be augmented to indicate how method components address other aspects, such as changeability, (e.g., how *Data\_Flow* supports modeling data transformation).

### Evaluation of Objectivity

Based on our experience in using MSDDL, we find that MSDDL is quite capable of supporting objectivity in classifying method components.

At the top level of MSDDL, we find that it is easy to decide objectively whether a component addresses an issue about a targeted problem or the system model. This can be decided, for example, by using explicit descriptions given by SDMs (e.g., *Entity\_Structure* in JSD) and by identifying the inputs and outputs of the system model (*Logical\_Output\_File* in LCP). It is also fairly easy to decide, in a similar way, if a component addresses an issue about a system model or a design document. For example, BOOD fairly clearly indicates the difference; it uses “identify object” to indicate the modeling activity and “produce a module specification” to indicate the documentation activity.

At the second level, we find that it is also fairly easy to decide what models or documents a method component supports because those are either described in SDMs or are well understood already. For example, in Fig. 6.6, JSD describes clearly that *State\_Vector* and *Data\_Stream* address the issue of communication

between two processes. Moreover, it is well understood that *Process\_Structure* in DSSD models the procedural structure of a program.

### Evaluation of Completeness

Based on the models (e.g., the JSD model) of the selected SDMs, we conclude that MSDL is quite complete since it is capable of classifying most method components which are explicitly defined in the models. For example, those *artifacts* indicated in Sec. 5.5.1, like *Entity\_Action\_List*, *Action\_Desc*, *Action\_List*, *Entity\_list*, *Entity\_Structure*, *Model\_Process*, etc., are classified either in Fig 6.5 or 6.6.

We note that other function frameworks (e.g., page 41 and 51 [WFP83a]) cover issues concerned with managing the design process and validating designs better than MSDL does. We think that this is because the SDMs we selected place much less emphasis on these issues. Thus, our models and frameworks need to be improved in order to address these issues. MSDL seems to lack sufficient details to express more precisely what issues a component addresses. For example, in Fig. 6.6, *Operation* and *State* are directly classified under Entity Model. The detailed issues they address inside an Entity Model are not explicitly shown by this classification. *Timing* is directly classified under Function Model. Thus, improvement is indicated here as well.



Some important models (e.g., a behavior model [HLN<sup>+</sup>90]) of a system are still missing in the framework. This suggests that more SDMs need to be added into *Selected\_SDMs* to enhance the BF.

#### 6.8.4 Evaluation of *Modeling\_Formalism*

In this experiment, we have identified some weaknesses of the MF. Here, we list those weaknesses:

- **Modeling guidelines:** the MF does not support modeling of guidelines. For example, DSSD has the guideline: “When the data is a *sequence*, use simple sequence of instructions”. Failing to model this hindered us from more explicitly showing the differences/similarities between LCP and DSSD.
- **Modeling criteria for decomposition:** the MF does not support modeling of the criteria for decomposing a system. For example, In RDM, a set of criteria is given to aid making the decision for how to decompose a system. Failing to model this characteristic would hinder us from comparing the structure of *Model\_Guide* with the structures suggested by other SDMs.
- **Modeling the other relations between design artifacts:** the MF does not support modeling some other relations between design artifacts. Design artifacts can be related in certain ways. For example, SD suggests that a model be related with its sub-models by function-call relations. Failing to

explicitly indicate the semantics of the relations greatly hinders comparing the underlying structures that organize systems.

### 6.8.5 Evaluation of CDM

In this experiment, we found that CDM is generally effective for comparisons of the SDMs. However, we do find that CDM can probably be improved in the following aspects:

- It is necessary to compare the major concepts earlier in the comparison process. By doing this, one can better understand the other differences between the SDMs. We suggest doing this in step *Select\_Comparison\_Topics*. We believe that understanding the differences in major concepts between SDMs should be very important for selecting the method components to compare. For example, in comparing JSD with SD, we at first compare the Problems they are aimed and the Principles they use. By doing this, we have straightforwardly gained a better understanding of their other differences.

## 6.9 Status

At present we have finished the first cycle of evaluation and adjustment of the BF (see Fig. 5.4, However, including the experiment described in Chapter 4,

development has gone through two iterations (Sec. 3.2.2, and 5.4.2)). The BF is close to a stable stage. Most of the method components specified in the models (e.g., Sec. 5.5.1) of the selected SDMs have been successfully classified within the BF. However, because our evaluation of the BF relies to some extent on SDM models that are still under development, we think that further modeling of SDMs may help to show where the BF needs to be enhanced. The effectiveness of the BF also needs more evaluation, which should be based on experiments in using it to aid comparisons and assessments of SDMs (this work will be presented in the next chapters).

Based on our evaluation, we think that one possible enhancement to the BF would be to further decompose MSDL. For example, the Entity Model could be decomposed to consist of Entity\_Identifier, Data, and Function. Doing this would enable us to classify the functions of the components (e.g., *Operation*, *State*) more precisely, which could then guide comparisons more effectively.

In this effort, we use the MF to model selected SDMs, We are now finishing the development of these models (i.e., the *Build\_Process\_Model* step in the second iteration of our evolution cycle). Although these models will need to be more thoroughly evaluated based on their applications, we can already begin to make some observations about MF improvement. One possible enhancement to the MF is to adopt more powerful formalisms to capture other kinds of

relations among *artifacts* (e.g., *inheritance* relations among the *Objects*, *call* relations between *Model\_Process* and *Function\_Process*, which cannot be clearly indicated by the Warnier Diagrams). A formalism that we think is promising is OPRR (Object-Property-Role-Relation [Wel89, Smo91]). OPRR has been shown to be capable of clearly modeling BOOD artifacts. However, we need to determine whether OPRR is capable of modeling other *artifacts* as well.

We will continue the evolutionary development of BF and MF. However, as shown earlier (Sec. 5.2 and 5.6), this work is very difficult. The BF and MF must satisfy the diverse demands of the community (e.g., academic researchers, SDM advocates, tool builders, project managers and practitioners). Thus, it should be developed based on community consensus rather than only on our view of SDMs. Although we have been evolving BF and MF as objectively as we can, we expect that community collaboration on this work should be much more effective (e.g., SDM advocates should contribute by validating the classifications and SDM models). Such community involvement would lead to a BF and MF that would be more credible and thus more widely used. Further, as [Boa90] suggests, this would help in more systematically codifying, unifying, distributing and extending SDM knowledge, thereby effecting progress in both software engineering practice and research.

In the next chapter, we present an experiment that analyzes the CDM-based comparison.



# Chapter 7

## Assessment of the CDM-based Comparisons

### 7.1 Goals of the Experiment

CDM relies on the following ideas:

1. SDM models should help in identifying method components to be classified.
2. The classifications should help in identifying the method components that should/could be compared.
3. The classifications of the method components should help in revealing overall differences between SDMs.
4. The models of these components should help in making conjectures and drawing conclusions about the similarities and differences between them.

Our previous experiments (Chapters 4 and 6) have shown that these ideas are basically valid and CDM seems to be useful in comparing certain aspects of SDMs. These experiments have shown that CDM has the following advantages:

1. CDM can help in more explicitly and rigorously showing the basis for drawing conclusions about SDM's. Further, this helps others to independently evaluate the resultant comparison.
2. Comparisons resulting from using CDM are more explicit, precise and objective.
3. CDM can be more systematically applied.

In earlier chapters we explored and validated these conjectures, rather than thoroughly assessing the effectiveness of CDM. Thus, in these earlier chapters, we did not attempt to compare our comparison results against any other similar comparisons. As a result, we were not able to conclude specifically how well CDM helps in comparing SDMs, and where CDM would more or less work effectively than some other comparison approaches.

To tackle these problems, it is necessary to carry out an experiment to systematically assess CDM, exploring its advantages and limitations. It is the purpose of this chapter to describe one such experiment. While we have not carried out an extensive set of similar experiments, we believe that the results of this experiment are typical of what should be expected. In Section 7.2, we describe the design

of this experiment—its methods, data and tools. Section 7.3 then presents the analyses of our comparisons.

## 7.2 Design of the Experiment

### 7.2.1 Basic Method

In a scientific discipline (e.g., numerical analysis), if one claims that one approach can produce better results under certain conditions, one must carry out experiments to demonstrate this. One commonly used method for doing this is to analyze and compare the results produced by using the new approach with those obtained by using an older approach under the same given conditions. This enables one to assess the improvement that the new approach achieves (if any). In addition, one can explore the limitations of the new approach by comparing results produced under some broader conditions.

Accordingly we experimented by comparing comparison results obtained using CDM with those obtained using other approaches. First we compare the SDM comparison results obtained under *CDM preferred conditions* (where CDM is *assumed* to work effectively). In doing this we hope to evaluate the maximal effectiveness of CDM. Second, we compare the results produced under CDM non-preferred conditions. In doing this we hope to explore CDM's limitations.



In Chapter 3.3.2, based on our experience in modeling SDMs, we hypothesized that CDM should be effective in comparing the following aspects of SDMs: 1) inter-component dependency, 2) need for human involvement, 3) development procedure, and 4) scope of issues. Thus, we select those to be the CDM preferred conditions.

## 7.2.2 Experiment Data

In order to carry out the experiment, we fixed: 1) a set of SDMs to be compared, and 2) a set of SDM comparison results previously obtained by using a previous approach.

For this experiment, we chose Jackson Systems Development (JSD) [Jac83] and Booch's Object Oriented Design (BOOD) [Boo86] as the SDMs to be compared. Our comparisons will be solely based on the two publications, [Jac83] and [Boo86]. We did not attempt to use any extended formulation of JSD and BOOD.

In this experiment we compared our comparison results with those obtained by Alan Birchenough and John Cameron [BC91], because that earlier work compares JSD with BOOD under both CDM preferred conditions and non-preferred conditions. This should allow us to explore both advantages and limitations of CDM. In addition, this comparison seems to us to be well organized and credible.

Thus, it seems fair to use this comparison as a representative of comparisons using informal approaches.

Note that in pursuit of our experimental goals, we need to compare and assess not only final comparison results, but also the processes and bases leading to these results. In Chapter 4, we have already presented the processes and bases for comparing BOOD with JSD, thus in this Chapter we compare only these comparison results with Birchenough and Cameron's results.

### 7.3 Comparing the Comparisons

In this section, we compare our comparison results with the comparison results shown in [BC91]. First, the comparisons are made under the CDM preferred conditions. We analyze why the paper [BC91] did not reveal the differences we have found by using CDM (if any). By this we hope to show the advantages of CDM. Second, we describe the comparison results of [BC91] under the non-preferred conditions. By this we hope to identify limitations of CDM.

### 7.3.1 Comparing the Comparisons Under Preferred Conditions

#### Differences in Inter-artifact Dependency

[BC91] does not devote much effort to analyzing the dependencies among design artifacts. Based on the explanation given by Booch, [BC91] notes that JSD covers a broader range of issues than BOOD does by providing a strategy for system analysis.

In using CDM, we arrived at the same result (Sec. 3.4.1). However, our result is obtained by doing data flow analysis on the JSD process model. We checked to see whether any artifact depends on artifacts that are not well-defined. By doing this we examined the SDM coverage of the development life-cycle. As our comparison of this aspect is based on explicit process models and systematic analysis of those models, our comparison seems to us to be more objective and therefore more convincing.

#### Differences in Need for Human Involvement

In Section 4.2.4, we have shown how we, in using CDM, analyze different needs for human involvement. Here we analyze how [BC91] makes similar comparisons. On Page 294 of [BC91], the comparison of *action* of JSD with *operation* of BOOD, states:

*JSD provides firm guidance on how to find candidate actions, and how to review and, if necessary, reject them. The analyst is told to exclude from the list any events which are system output events (e.g. ...). As a check on completeness, the analyst is also directed to consider the inputs available at the system boundary, inasmuch as these are known at that stage; known data storage requirements if any; and, as in OOD, to perform a rough grammatical analysis in order to find the nouns and verbs that should be recognized as entities (objects) and actions (operations).*

From this, we can see that this conclusion rests upon the analysis and comparison of the guidelines and procedures provided in BOOD and JSD. This is certainly a plausible way of making comparisons of this aspect. However, in using CDM, we specified these guidelines and actions (procedures) more rigorously and explicitly. Thus, CDM seems to us to provide a more convincing basis for drawing the same conclusion.

It seems to us, moreover, that process modeling allows us to analyze and compare SDMs more thoroughly. For example, BOOD provides additional guidance; it illustrates the subtypes of an artifact to be identified. Thus, a designer can identify the artifact through identifying its subtypes, which are often more concrete and thus more easily identified. *Identify\_Object* specified in the BOOD model (b) (see Sec. 6.1.1) shows this kind of guidance. Such guidance is more

implicit than the guidance towards providing guideline and action descriptions. Thus, our analysis technique seems to be effective in identifying guidance that is more or less implicit.

Using the BF and MF, we can explicitly indicate that both JSD and BOOD use the guideline “finding noun” to identify the program components (i.e., entity and object respectively).

### Differences in Development Procedures

In comparing development procedure, [BC91] identified the following differences:

**Difference 1:** Page 295 of [BC91] states that:

*JSD encourages the analysis of actions before entities, whereas the OOD steps are to identify objects first and operation second.*

**Difference 2:** Page 295 of [BC91] states that, when comparing *action* of JSD with *operation* of BOOD:

*..., during modeling, only operations [actions of JSD] that change the state of an object [entity] (“constructor” in OOD) are included. Selector<sup>1</sup> are ignored until the network stage, and then they are not documented*

---

<sup>1</sup>An operation of BOOD that evaluates the current object state.

*explicitly as operations on an object, but as inspections of the objects state vector(private data frame).*

In using CDM, we are able to show the first difference more explicitly. We can indicate precisely, by referring to the relevant models (the BOOD model (a)(1)(2) and the JSD model (g)(1)(2) in Chapter 4), the difference in the order of performing the design actions that produce those artifacts.

We expect that CDM will not help much in revealing the second difference. The reason is that JSD does not describe this explicitly in the criteria definition (e.g., IA for identifying a JSD action). Thus, the model of JSD is likely to fail to capture this information. Consequently, our comparison of JSD with BOOD probably will not reveal this difference because it requires analysis of the informal descriptions of the artifacts. If JSD were to specify this characteristic clearly by defining it into the criteria (i.e., IA), then in comparing these criteria and design actions, CDM would be much more effective in exploring this difference.

### **Differences in scope issues**

Page 296 of [BC91] states:

*JSD is explicitly object-oriented only during the modeling stage, whereas OOD [BOOD] attempts to identify objects that can both describe the real-world and satisfy the system functional requirements.*

Using a suitable framework (e.g., the BF we have developed) in CDM helps show this difference clearly. In using CDM, we first model BOOD and JSD, where the criteria for determining object of BOOD and entity of JSD are specified. These criteria show that those two artifacts are identified using object-oriented concepts. Then, the classification shows that object addresses issues for modeling a system whereas entity addresses issues for modeling the targeted problem. From this we can conclude that JSD is explicitly object-oriented only during modeling the targeted problem whereas BOOD is object-oriented in modeling the system. Further analysis shows that Booch implicitly suggests that object should also be used to model the targeted problem. Thus, we reach the same conclusion by using CDM. Again we believe our CDM-based approach rests on a more solid and convincing basis for this conclusion.

### 7.3.2 Comparing the Comparisons under Non-preferred Conditions

**Difference 1:** Page 293 of [BC91] states:

*Jackson System Development and Object-Oriented Design have one major—arguably central—principle in common; namely that the key to software quality lies in the structuring of the solution to a problem in such a way as to reflect the structure of the problem itself.*

Using CDM and our BF, we should be able to draw a similar conclusion. In Chapter 6, we classified the method components of JSD and BOOD. The classification of *concepts* (see Fig. 6.1) presented in Chapter 6 helps us to understand that both JSD and BOOD use “Model Reality” as a principle to “Design right software”. Based on the BF, we also classified other major concepts of JSD and BOOD in a systematic manner. Thus, it seems that CDM, at least to some extent, supports the conclusion that JSD and BOOD both rest upon the same fundamental ideas.

**Difference 2:** Page 295 of [BC91], when comparing object of BOOD with entity of JSD, states:

*Only objects that suffer time-ordered actions, or about which it is necessary to maintain data, qualify as JSD entities.*

This difference seems to reveal some limitations of CDM. In using CDM, we specify the criteria for determining an artifact. Then by comparing the criteria for determining different artifacts to be compared, we can identify differences between the semantics of these artifacts. However, it seems to us that criteria that are explicitly described in SDMs are often incomplete. Thus, those criteria often can only be used to eliminate candidate artifacts, but cannot be used to decide an artifact. The above difference shows this clearly. JSD does explicitly explain that maintaining data is a sufficient condition for determining an entity, but implicitly



suggests this <sup>2</sup>. Thus, the effectiveness of the CDM strategy that suggests modeling and comparing criteria is seriously hampered by this lack of explicitness in the SDM. Without checking wider context and subtle implications of an SDM in the course of using CDM, one might well overlook some important differences between SDMs.

### 7.3.3 Comparing Integration Strategies

Based on the JSD/BOOD comparisons obtained in using CDM, we identified a way to integrate JSD with BOOD. We now show how our integration strategy compares with that suggested in [BC91]. First, we list the integration strategies suggested by [BC91]:

Page 298 of [BC91] states that:

*time-ordered analysis can be applied to document object class.*

Page 298 of [BC91] states that:

*It follows that every operation must be either predefined in the language or defined in a package specification. OOD is a highly suitable technique for designing and implementing such package.*

---

<sup>2</sup>as noted earlier, it is not included in the criteria for determining entities (see page 40 of [Jac83]).

Page 298 of [BC91] states that:

*OOD is better used to support bottom-up component design, while JSD is stronger when dealing with the user-oriented specification issue.*

Comparing our integration strategy (Sec. 4.2.6) with these, we find that our integration is far more complete, and includes all the strategies suggested by [BC91]. Furthermore, Our strategies are argued based on solid understanding of the weaknesses and strengths of JSD and BOOD, presented systematically within a development life cycle framework. It is also worth noting that we derived our integration strategy naturally from our comparison; the function classification suggests the ways in which the SDMs might be integrated; further comparisons help us to select from those ways. This seems to suggest that CDM (including a suitable BF) is a superior tool for studying the integration of SDMs.

## 7.4 Summary

In this section, we summarize our assessments of CDM as an SDM comparison approach, comparing it to the advantages we expected as summarized in Section 7.1. Our experiment indicated that CDM has the following advantages:

1. CDM enables to use the following strategies or techniques as the basis for drawing conclusions about differences between SDMs.

- *Data flow analysis*: This technique can be used to analyze data flow through design actions, and can help in indicating the dependence among design artifacts. This can pinpoint the scope of issues that an SDM addresses and thus help in analyzing the software development life-cycle coverage of the SDM.
- *Functional analysis*: This technique can be used to analyze the inputs and outputs of a design action, and can help in gaining an understanding of the issues that are addressed by this design action.
- *Sub-action analysis*: This technique can be used to analyze the lower-level actions to be performed in a design action, and can help in indicating the difference in scope of issues.
- *Procedural analysis*: This technique can be used to analyze how detailed guidance (e.g., for how to identify an artifact) is described in an SDM. This can help in indicating how much a human must be involved in a suggested design action.
- *Artifact elaboration analysis*: This technique can be used to analyze how well an SDM elaborates its artifacts by indicating their subtypes. This can help in indicating how well the SDM provides guidance for designers.

- *Criteria analysis*: This technique can be used to analyze the criteria described for determining an artifact. This can help in indicating differences between two artifacts.
  - *Control flow analysis*: This technique can be used to analyze the order of performance of design actions. This can help in identifying differences in the procedures used in making artifacts.
2. In using CDM, we expect to obtain more precise, explicit and objective comparison results. In this experiment, we found that those are achieved in CDM to some extent, in the following ways:
- Explicit comparisons are achieved in CDM by indicating explicitly which portion of a model is different from which portion of another model, and by describing explicitly how they are different. For example, control flows in the models can be used to explicitly indicate differences in the order of performing design actions. Criteria definitions can be used to explicitly indicate which criterion differs from which other criterion, and in what ways.
  - Precise comparisons are achieved by developing and comparing SDM models that capture the details of the SDMs. Process modeling as a technique allows us to more systematically model SDMs and thus helps in modeling the details of SDMs precisely. For example, modeling

the subtype hierarchy of design artifacts shows precisely how an SDM provides guidelines for determining artifacts.

- Objectivity is achieved by using well-recognized analysis techniques. These analyses lay down foundations for drawing conclusions and thus enable independent evaluations of the conclusions, which improve the objectivity of the comparisons. In addition, CDM does not rely upon experiences and examples of use as the primary basis for making comparisons. They are, instead, used as secondary aids in obtaining more objective comparisons.

3. In using CDM we expect that comparisons can be more systematically made.

In this experiment, we found that [BC91] seems to share the basic ideas with CDM to some extent. The paper [BC91] proceeds in this way: 1) compare the SDM's major concepts (Sec.6.2.1.1 of [BC91]), 2) describe the SDM's major steps (Sec. 6.2.1.2 of [BC91]), 3) describe what is to be compared (Sec. 6.2.1.3 of [BC91]), and 4) make comparisons (after Sec. 6.2.2.1). Thus, it seems to us that CDM is a more rigorous, explicit and precise version of the way people have previously gone about comparing SDMs. Moreover, CDM suggests formally modeling SDMs and classifying their parts, which can ensure that successive comparisons are done more systematically.

This experiment also indicated some limitations of CDM.

1. With the current state of the art of SDM development, it is almost impossible to capture all the information provided by an SDM in a formal, rigorous model. Many method components are not well and completely defined. (As Cameron [CCW91] suggests, it is unrealistic to suppose that the rules of a method can all be perfectly well defined). Those components that are not well-defined are difficult to model formally. Thus, some comparisons made during use of CDM must still rest upon some informal analysis or judgement of the informal descriptions of SDMs. Falling back upon such informality helps to prevent misleading results or overlooked differences/similarities.
2. CDM and the modeling approach sometimes fail to explore the implicit similarities between the semantics of artifacts. For example, it fails to show that the Model Processes of JSD are usually constructor operations of BOOD. Understanding this kind of similarities requires additional knowledge about the software design.
3. At the current stage, CDM is a systematic approach only at a certain high level. For analyzing detailed differences, it rests upon the judgement of those using CDM, obliging them to decide what analysis techniques to use in order to explore differences between SDMs.

To overcome the second limitation, in the next chapter, we define a process program for CDM to improve the repeatability of CDM.



# Chapter 8

## An Encoding of the CDM Process

Our research is aimed at providing a systematic approach to comparing SDMs. To improve the repeatability of the approach, we need to define CDM precisely and rigorously. This will entail other analysts to follow this process closely, and to obtain the same comparison results in comparing the same SDMs.

In this research we view an SDM as a piece of software. We also view CDM as software process used to compare and analyze SDMs. Process programming has been demonstrated to be capable of precisely and rigorously defining a software process. Therefore, we can use the process programming technique to define CDM precisely. Specifically, the development of detailed code to express a software process has been shown to be very effective in leading to very sharp understanding of processes. Thus, we believe that specifying CDM in a suitable coding language



can improve its understandability and hence its repeatability. We select the Ada-like notations to specify CDM, because Ada supports good readability and has precise syntax and semantics.

Although Ada supports expressing concurrent processes by using task and entry calls, we believe that the notation of CoBegin and CoEnd<sup>1</sup> of Concurrent Pascal [Han75] is more understandable. It expresses that the statements inside a CoBegin and CoEnd are to be concurrently executable. Thus, we used this notation in our encoding.

Because our encoding of the CDM process is not aimed at automating CDM, but rather presenting CDM precisely, we did not attempt to compile and execute our CDM code.

In this chapter we present process code for CDM. The reader can also use this process code as a detailed and precise summary of CDM. First, we describe the static structure of the CDM process code. Second, we define the program components and explain their functions.

---

<sup>1</sup>It has the semantics same with Parbegin and Parend that were proposed by Dijkstra [Dij65]

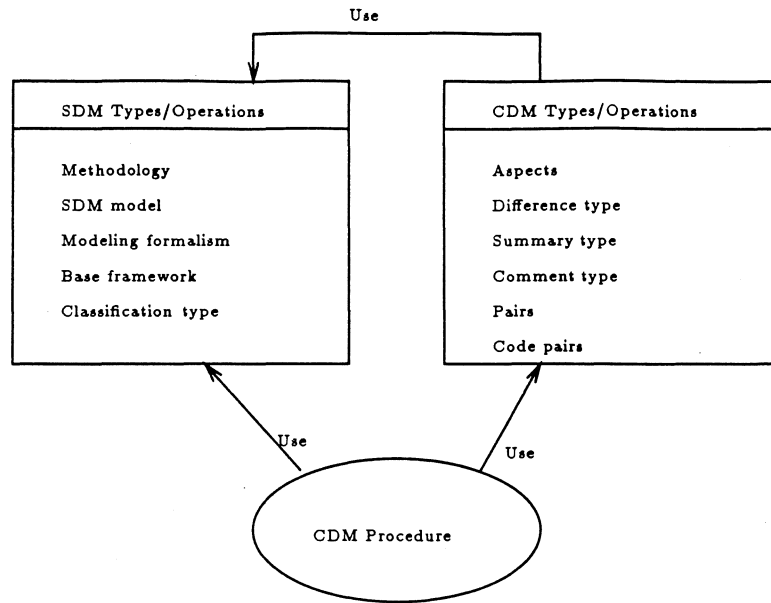


Figure 8.1: Static Structure of the CDM Process Program

---

## 8.1 Program Structure

Our CDM process code consists of two Ada packages (CDM and SDM\_Pkg) and one Ada procedure (Compare\_SDM). As Fig. 8.1 illustrates, CDM and SDM\_Pkg provide the data types and operations that are used in the SDM comparison procedure Compare\_SDM.

## 8.2 Compare SDM Procedure

In this section, we define the CDM process code.

This procedure consists of eight major steps which are labeled in the code. We now describe them briefly. A later section will explain the rationales behind these steps, guidelines applied by these steps, and criteria used by these steps.

1. Decide the aspects with respect to which the SDMs will be compared. These aspects guide subsequent CDM activities.
2. Select the formalisms used for modeling SDMs. The SDM models specified using the formalisms should help analyze the SDMs with respect to the aspects defined.
3. Develop and validate the SDM models.
4. Classify SDM components and validate the classifications.
5. Identify the overview differences between the SDMs based upon the classifications.
6. Identify the comparable SDM components. Decide the order in which they will be compared.
7. Compare comparable SDM components with respect to the aspects defined. If necessary, the components might be coded to aid the further analysis of these components.

8. Summarize the comparison results.

9. Make comments on the comparison results.

```
--| this package provides data types for
--| Methodology, Modeling_Formalism,
--| Base_Framework, SDM_Model, Classification_Type;

WITH SDM_Pkg;   USE SDM_Pkg;

WITH CDM;       USE CDM;

Procedure Compare_SDM(SDM_1   : IN Methodology;
                     SDM_2   : IN Methodology;
                     MF      : IN Modeling_Formalism
                               := Song_MF;
                     BF      : IN Base_Framework
                               := Song_BF;
                     Summary  : OUT Summary_Type;
                     Comments : OUT Comment_Type ) IS

-----

--
-- Variables used:
--
-----

Aspects : Aspect_List;

SDM_1_Model,
```

```
SDM_2_Model : SDM_Model;

Classification,
Classification_1,
Classification_2 : Classification_Type;
Overview_Differences : Difference_List;
Comparable_Pairs,
Ordered_Pairs : ARRAY (integer <>) of Pairs;
SDM_1_Model,
SDM_2_Model : SDM_Model;

-----

Begin

  --| decide what aspects to compare SDMs against.
  1. Aspects:= Define_Aspect_To_Compare;

  --| select modeling formalisms
  2. MF := Select_Modeling_Formalism(MF, Aspects);

  --| model SDMs;
  3. LOOP

      CoBegin

          IF Valid_Result_1 \= OK THEN

              SDM_1_Model := Model_SDM(SDM_1, MF);

          END IF;

          IF Valid_Result_2 \= OK THEN
```

```
        SDM_2_Model := Model_SDM(SDM_2, MF);

    END IF;

CoEnd;

CoBegin

    Valid_Result_1 := Validate(SDM_1, SDM_1_Model);

    Valid_Result_2 := Validate(SDM_2, SDM_2_Model);

CoEnd;

IF Valid_Result_1 = Valid_Result_2 = OK THEN

    Exit;

END IF;

END LOOP;

--| classify the SDM models by using the BF;

4. Valid_Result_1 := NOT_OK;

    Valid_Result_2 := NOT_OK;

LOOP

    IF Valid_Result_1 \= OK THEN

        Classification_1 :=

            Classify(SDM_1_Model, SDM_1, BF);

    END IF;

    IF Valid_Result_2 \= OK THEN

        Classification_2 := Classify

            (SDM_2_Model, SDM_2, BF);

    END IF;

CoBegin
```

```
Valid_Result_1 :=
    Validate(SDM_1, SDM_1_Model, Classification_1);
Valid_Result_2 :=
    Validate(SDM_2, SDM_2_Model, Classification_2);
CoEnd;
IF Valid_Result_1 = Valid_Result_2 = OK THEN
    Exit;
END IF;
END LOOP;

Classification := Merge_Classification
    (Classification_1, Classification_2);

5. Overview_Differences :=
    Analyze_Class(Classification);

--| identify the comparable components;

6. Comparable_Pairs :=
    Identify_Comparables(Classification);

--| order the comparable components;
Ordered_Pairs := Order_Pairs
    (Comparable_Pairs, SDM_1, SDM_2);

--| compare these comparable component pairs;

7. Detailed_Differences := NULL;
```

```
FOR I IN Ordered_Pairs'FIRST..Ordered_Pairs'Last LOOP

  Temp_Detailed_Differences :=

    Compare_Pairs (Ordered_Pairs[I], Aspects, Result);

  --| using coding to analyze the differences;

  IF Result = NEED_ANALYSIS THEN

    Codes := Coding_SDM(Ordered_Pairs[I], SDM_1_Model,

                        SDM_2_Model, SDM_1, SDM_2, Aspects);

    Temp_Detailed_Differences :=

      Analyze_Codes(Codes, Aspects);

  END IF;

  Detailed_Differences := Append(Temp_Detailed_Differences,

                                Detailed_Differences);

END LOOP;

--| making summary based on the comparisons;

8. Summary := Make_Summary(Detailed_Differences);

--| make comments based on the summary.

9. Comments := Make_Comments(Summary, SDM_1, SDM_2);

End Compare_SDM;
```



### 8.3 Definition of the CDM Package

In this section, we define the interfaces of the major procedures and data types used in the procedure Compare\_SDM.

```

PACKAGE CDM IS

TYPE Summary_Type      : ARRAY(integer <>)(integer <>) of String;

TYPE Difference_Type   : ARRAY(integer <>) of String;

TYPE Comment_Type     : ARRAY(integer <>) of String;

--|

--| Aspects with respect to which SDMs will be compared;

--|

TYPE Aspect IS

    RECORD

        Name: String;

        Desc: String;

    END;

TYPE Aspect_List is ARRAY of Aspect;

--| Pairs of method components;

TYPE Pairs IS

    RECORD

        Component_1: String;

        Component_2: String;

    END;

```

```
--| Codes of a pair of method components;

TYPE Code_Type IS

    RECORD

        Code_1: String;

        Code_2: String;

    END;

TYPE Result_Type IS (NEED_ANALYSIS, OK);

BEGIN

    --| decide what aspects to compare SDMs against.

    FUNCTION Define_Aspect_To_Compare RETURN Aspect_List;

    --| select modeling formalisms from existing formalisms.

    FUNCTION Select_Modeling_Formalism

        (MF : IN Modeling_Formalism;

         Aspects : IN Aspect_List)

        RETURN Modeling_Formalism;

    --| modeling an SDM

    FUNCTION Model_SDM(SDM_1 : IN Methodology;

                       MF : IN Modeling_Formalism)

        RETURN SDM_Model;

    --| code a pair of method components;
```

```
FUNCTION Coding_SDM(Coded_Pairs : IN Pairs;
                  SDM_Model_1 : IN SDM_Model;
                  SDM_Model_2 : IN SDM_Model;
                  SDM_1 : IN Methodology;
                  SDM_2 : IN Methodology;
                  Aspects : IN Aspect_List)
    RETURN Code_Type;

--| classify the SDM models by using the BF;
FUNCTION Classify(SDM_1_Model : IN SDM_Model;
                SDM_1 : IN Methodology;
                BF : IN Base_Framework)
    RETURN Classification_Type;

--| merge classfiications;
FUNCTION Merge_Classification
    (Class_1 : IN Classification_Type;
     Class_2 : IN Classification_Type)
    RETURN Classification_Type;

--| analyze the classification to get some observation.
FUNCTION Analyze_Class(Class : IN Classification_Type)
    RETURN Differences_Type;

--| identify the comparable components;
```

```
FUNCTION Identify_Comparables(Class: IN Classification_Type)
```

```
    RETURN Pairs;
```

```
--| order the comparable components according the dependency
```

```
--| relations among them.
```

```
FUNCTION Order_Pairs(Comparable_Pairs : IN Pairs;
```

```
    SDM_1 : IN Methodology;
```

```
    SDM_2 : IN Methodology)
```

```
    RETURN Pairs;
```

```
--| compare these comparable components;
```

```
FUNCTION Compare_Pairs(Compared_Pairs : IN Pairs;
```

```
    Aspects : IN Aspect_List;
```

```
    Result : OUT Result_Type)
```

```
    RETURN Difference_Type;
```

```
--| analyze the codes of method components to understand
```

```
--| the differences;
```

```
FUNCTION Analyze_Codes(Codes : IN Code_Type;
```

```
    Aspects : IN Aspect_List)
```

```
    RETURN Difference_Type;
```

```
--| make summary of the differences among the SDM components;
```

```
FUNCTION Make_Summary(Diff : IN Difference_Type)
```

```
    RETURN Summary_Type;
```



## 8.4 Implementation of the CDM Package

### 8.4.1 Define Aspects to Compare

**Principles:** Before comparing the SDMs, it is necessary to decide the aspects with respect to which comparisons will be made. These aspects then will guide the comparisons, helping selection of the modeling formalism, the classification framework, and other artifacts involved in the comparison. These aspects may also guide analysts in evaluating, selecting and integrating the SDMs.

**Criteria for deciding aspects:** The aspects should be decided according to the needs of the analysts. However, we suggest that the aspects should be at least partially described in terms of the method component types and inter-component relations. Analysts must name each aspect to facilitate being able to refer it. The name should clearly express the semantics of the aspect.

**Guidelines for looking for aspects:** An analyst is free to choose the aspects to compare. Examples of the aspects include human involvement in the development process, inter-artifact dependency, procedure differences, methodology applicability, expressiveness of the representation, etc.

**Action:**

```
FUNCTION Define_Aspect_To_Compare RETURN Aspect_List IS
    Aspects : Aspect_List;
```

```
BEGIN
  Elaborate_Aspects;
  LOOP
    Aspects[I].Name := Naming_Aspect;
    Aspects[I].Desc := Define_Aspect;
    IF NO_ASPECT THEN Exit;
  END LOOP;
END Define_Aspect_To_Compare;
```

### 8.4.2 Select Modeling Formalism

**Principles:** Using a formalism to model SDMs can help in highlighting the method components and inter-components relations in an SDM. The formalism helps in precisely expressing the SDM and thus understanding the SDM. However, a modeling formalism must be carefully chosen to prevent it from concealing important features of the SDM modeled.

**Criteria for choosing the formalism:** The modeling formalism selected should help in comparing SDMs with respect to the aspects to be studied. Generally, a modeling formalism selected must support abstraction of SDM features. In addition, its notations must be precisely and rigorously defined. As a human will analyze the SDM model specified in the formalism, it is desirable that the formalism provides high degree of understandability.

**Guidelines for choosing the formalism:** For analyzing the functions (e.g., what modeling capabilities it supports) of an SDM, a functional modeling formalism might be selected. For analyzing the differences in the procedures used to carry out design steps, a functional and/or procedural modeling formalism might be selected. For analyzing the human involvement required in using an SDM, a procedural modeling formalism might be selected. For analyzing the semantics of the artifacts suggested by an SDM, the rule-based modeling formalism might be selected.

**Action:**

```

FUNCTION Select_Modeling_Formalism
    (MF: IN Modeling_Formalism;
     Aspects: IN Aspect_list) RETURN
    Modeling_Formalism IS
    Final_Formalism, Formalism : Modeling_Formalism;
    Results    : String;
BEGIN
    FOR I IN Aspect_List'range LOOP
        Results := Analyze_Aspects(Aspects[I]);
        Formalism := Decide_Formalism(Results, MF);
        Final_Formalism :=
            Incorporate(Formalism, Final_Formalism);
    END LOOP;
    RETURN Final_Formalism;
END Select_Modeling_Formalism;

```



### 8.4.3 Model Design Methodologies

**Principles:** The SDM model at higher abstraction levels should be compact and clear, yet complete enough for further refinement. Since in this step it has not been decided which method components to compare, it is desirable to avoid specifying details that might be irrelevant to future comparisons.

**Criteria for deciding what method component to model:** The model of an SDM should be developed according to the aspects with respect to which the comparisons will be made. The model must aid directly the comparisons with respect to these aspects. For example, if the structures of artifacts will not be compared, these structures may not need to be modeled.

**Guidelines for identifying and modeling method components:** The framework (i.e., MCTH) we suggested to classify method components can be used to guide the identification and modeling of the method components. The principles behind an SDM are often described in an early part of the SDM. These principles are normally described as design theories, fundamental design principles, concepts, etc. The artifacts are often described as design models, components of the models, documents and specifications. The criteria are often described as the definitions of design artifacts. The guidelines are often described in an SDM as the techniques, heuristics, and strategies for choosing and specifying artifacts. The actions are often described in an SDM as design steps, procedures, or processes to be followed for producing

or evaluating artifacts. The representations are often described in an SDM as the modeling formalisms, notations, templates, or design languages.

**Actions for developing SDM models:** Analysts can choose their own way to model SDMs as long as the models satisfy the requirements we described. CDM itself does not suggest the process used in developing SDM models.

#### 8.4.4 Classify Method Components

**Principle:** The classification of method components can help in identifying comparable method components and reveal differences between the method components.

**Criteria for placing two components in a same class:** For a functional classification, the two components should have the same external functions. For a type classification, the two components should have same structures or internal characteristics.

**Action for developing the classifications:** With a given classification framework, the analysts can use various processes to classify method components. We suggest using the following process:

```

FUNCTION Classify(SDM_1_Model : IN SDM_Model;
                  SDM_1      : IN Methodology;
                  BF         : Base_Framework)
RETURN Classification_Type IS
    Classification      : Classification_Type;

```

```
BEGIN

    Classification := Classify_Structures
                        (SDM_1_Model, SDM_1,
                        BF.type_framework);

    Classification := Classify_Functions
                        (SDM_1_Model, SDM_1,
                        BF.function_framework);

    RETURN Classification;

END Classify;
```

### 8.4.5 Analyze Classifications

**Principle:** Analyzing the classifications of method components helps determine the overall differences between SDMs. It also helps reveal the differences in the functions provided by SDMs.

**Criteria for identifying differences:** Differences are found when finding that a method component of one SDM exists in a class where no method component from another SDM exists. This reveals that one SDM addresses an issue while another does not.

**Actions for identifying differences:** The analyst can use the above criteria to identify differences. We advocate no explicit process.

### 8.4.6 Identify Comparable Method Components

**Principle:** It may not be meaningful to compare two arbitrary method components. Only comparison of the method components that address the same or similar issues, and that have similar roles in SDMs, are meaningful. For example, an artifact structure can be compared with another artifact structure, but not with a representation. Thus, before comparing SDMs, the analyst must decide which components can be compared.

**Criteria for deciding a pair of comparable components:** The method components which are categorized in the same class could be comparable because they address similar issues and have similar roles in an SDM.

### 8.4.7 Order Comparable Method Components

**Principles:** Method components have dependencies. Comparisons of some method components may help the understanding of differences between other method components. Therefore, comparable method components need to be ordered according to their dependencies. Then analysts can compare method components in this order.

**Guidelines for ordering method components:** The most common dependency among method components is the composition relation. A method component might be a part of another method component. For example,

when comparing artifact “object” of object oriented designs, an analyst would like to first compare the definitions of “object” before comparing the definitions of “operations”, which are contained in objects.

#### **8.4.8 Compare Method Components**

**Principles:** The comparison of method components may require examination of the code for the components in order to analyze their detailed differences. The decision about whether the method components need to be coded cannot be made until the analyst starts to compare the components.

#### **8.4.9 Make Summary**

**Principle:** A summary should provide a complete view of the differences between the SDMs compared. The summary should be organized according to the aspects with respect to which the SDMs are compared.

#### **8.4.10 Make Comments**

**Principles:** Comments provide further observations on the differences between the SDMs compared. The observations may evaluate the SDMs based on their differences.

## Chapter 9

# Conclusion and Future Work

In the past two decades, quite a large number of SDMs have been developed to improve software productivity and software quality. These SDMs have been compared and evaluated in efforts to understand them, integrate them, and improve them. However, we have found that most of the previous comparisons were not scientific. We suggested the use of the classical scientific method to compare SDMs. Accordingly, we described a software-process-modeling based comparison approach, CDM, to improve the comparisons among SDMs.

In this thesis, we have done the following:

1. Described motivations for comparing SDMs,
2. Surveyed the previous comparisons among SDMs and analyzed their limitations,
3. Described the motivations for systematic and objective comparison among SDMs,

4. Laid down a foundation for using process modeling techniques to aid SDM comparisons,
5. Described CDM, a process-modeling based SDM comparison approach, including its two essential components—a classification framework and an SDM modeling formalism.
6. Carried out three experiments to validate CDM, the classification framework, and the modeling formalism.

We found that comparing SDMs through process modeling has a number of benefits:

- The comparisons are more objective, explicit and precise.
- The comparisons can be independently evaluated and can thus be made more convincing.
- The comparisons are directly helpful to the study of the integration of SDMs. Strategies for integrating SDMs might be directly derived from the comparisons.
- Software modeling/analysis techniques can be adapted to analyze SDMs. Moreover, software modeling/analysis tools might be used.

We found that comparing SDMs through process modeling has a number of limitations:

- The comparisons sometimes need to rely on the informal analysis on the informal descriptions of SDMs.
- The comparisons may fail to reveal the differences or the similarities in the implicit semantics of design artifacts.
- The CDM process is systematic only at a certain high level.
- The design problems and principles are hard to be formalized.

This research is still at an early stage. The following work is indicated as a continuation of this research:

- CDM needs to be used to compare more SDMs (e.g., real-time design methodologies) to further evaluate and enhance CDM.
- We plan to more thoroughly explore the aspects with respect to which CDM could be ineffective in comparing SDMs. For example, we may start by analyzing how CDM can help determine whether an SDM can support the design of a large scale software system.
- A database might be developed to store SDMs. a powerful database can facilitate SDM retrieval, and thus aid the comparison, selection, and integration of SDMs.
- The BF and MF described in Chapter 5 are still not very complete. More SDMs need be examined against the BF to improve the completeness of the



BF. Other useful modeling formalisms need be identified and incorporated into the MF to improve its completeness.

- Design/modeling tools might be used to aid the CDM-based comparisons.
- The software design problem space needs to be clearly defined. Thus, CDM can be used as the frontend of a technique for evaluating SDMs and identifying the application domains of the SDMs.
- The design issues addressed by a software design process might be weighted, enabling the quantitative evaluation of various SDMs. The function framework can be used to identify those issues and their relations. The type framework of the BF might be used to measure the comprehensiveness of an SDM.
- The objective and detailed comparison of SDMs might help in developing generic SDM models (e.g., one model for all object-oriented SDMs), which can facilitate and accelerate the study of SDMs, the integration of SDMs, and the customization of SDMs.

# Bibliography

- [ABC<sup>+</sup>91] P. Arnold, S. Bodoff, D. Coleman, H. Gilchrist, and F. Hayes. An evaluation of five object-oriented development methods. Technical report, HP Laboratories Technical Report. HPL-91-52, June 1991.
- [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.
- [Ber78] G.D. Bergland. Structured design methodologies. In *15th Annual Design Automation Conference Proceedings*, June 1978.
- [Ber81] G.D. Bergland. A guided tour of program design methodologies. *Computer*, 14(10):13–37, Oct. 1981.
- [BFL<sup>+</sup>83] F. Bodart, A. Flory, M. Leonard, A. Rochefeld, C. Rolland, and H. Tardieu. Evaluation of CRIS 1 I.S. development methods using three cycles framework. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 191–206. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.

- [Boa90] National Research Council's Computer Science/Technology Board. Scaling up: A research agenda for software engineering. *Comm. of ACM*, 33(3):281-293, March 1990.
- [Boo86] G. Booch. Object-oriented development. *IEEE Transactions On Software Engineering*, 12(2):211-221, February 1986.
- [Boo91] G. Booch. *Object-Oriented Design with Applications*. The Benjamin/Commings Publishing Company. Inc., 1991.
- [Bra83] I. Brandt. A comparative study of information systems design methodologies. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 9-36. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.
- [BRS83] M. L. Brodie, D. Ridjanovic, and E.O. Silva. On a framework for information systems design methodologies. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 231-242. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.
- [CCW91] J. R. Cameron, A. Campbell, and P. T. Ward. Comparing software development methods: An example. draft paper, 1991.

- [Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965. [reprinted in 1968].
- [EHZAG89] M. Elizabeth, C. Hull, Adib Zarea-Aliabadi, and D. A. Guthrie. Object-oriented design, Jackson system development (JSD) specification and concurrency. *Software Engineering Journal*, March 1989.
- [Fre83] P. Freeman. Fundamentals of design. In *Tutorial: Software Design Techniques*. IEEE Computer Society Press, Washington, DC, 1983.
- [Gri78] S. N. Griffiths. *Design Methodologies—A Comparison*, volume II. Infotech International, Maidenhead, England, 1978.
- [Han75] B. Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, pages 199–207, June 1975.
- [Han86] K. Hansen. *Data Structured Program Design*. Prentice-Hall Cliffs, NJ, 1986.
- [HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnuell, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Transaction on SE*, 16(4):403–414, April 1990.
- [Jac75] M. Jackson. *Principles of Program Design*. Academic Press, 1975.
- [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.
- [Kat89] T. Katayama. A hierarchical and functional software process description and its enactment. In *Proc. of 11th International Conference on Software Engineering*, May 1989.
- [KF87] G.E. Kaiser and P. H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the 9th International Conference on SE*, pages 180–188, 1987.
- [KH88] M. I. Kellner and G. A. Hansen. Software process modeling. Technical report, Technical Report CMU/SEI-88-TR-9, May 1988.
- [KM85] Roger King and Dennis McLeod. Semantic data models. In *Principles of Database Design, Vol.1, Logical Organization*, pages 115–151. Prentice-Hall, Inc. Englewood Cliff. NJ., 1985.
- [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.
- [Oli83] A. Olive. Analysis of conceptual and logical models in information systems design methodologies. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature*

- Analysis*, pages 63–86. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.
- [Orr77] K. T. Orr. *Using Structured System Design*. Yourdon Press, NY, 1977.
- [Ost87] Leon J. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, March 1987.
- [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.
- [PCW84] D.L. Parnas, P.C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proceedings of the 7th International Conference on Software Engineering*, pages 408–417, March 1984.
- [PJ80] M. Page-Jones. Transform analysis. In *The practical guide structured system design*, pages 181–203. Yourdon Press, 1980.
- [PT77] L. J. Peters and L. L. Tripp. Comparing software design methodologies. *Datamation*, 23(11):89–94, Nov. 1977.
- [RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

- [RS78] C. V. Ramamoorthy and H. H. So. Software requirements and specifications: Status and perspective. In *Tutorial: Software Methodologies*, pages 43–164. IEEE Computer Society, 1978.
- [SHO90] S. M. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 206–216, Irvine, Dec. 1990.
- [SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM System Journals*, 13(2):115–139, 1974.
- [Smo91] K. Smolander. Opr— a model for modeling systems development methods. In *The Proceedings of 2nd Workshop on the Next Generation of CASE. Trondheim, Norway*. 1991.
- [SO89] X. Song and L. J. Osterweil. Debus: a software design process program. Technical report, Arcadia-document, UCI-89-02, April 1989.
- [SO91] X. Song and L. J. Osterweil. A survey of process programming design formalisms. available from the authors upon request, June 1991.
- [Sol83] H. G. Sol. A feature analysis of information systems design methodologies: Methodological considerations. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 1–8. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.

- [TBC<sup>+</sup>88] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. Selby, J. C. Wilden, A. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development environments*, pages 1–13, Nov. 1988.
- [Uni83] United States Department of Defense. *Reference Manual for Ada Programming Language*, 1983.
- [War76] J.D. Warnier. *Logical Construction of Programs*. Van Nostrand Reinhold, New York, 1976.
- [Was80] A. Wasserman. Information system design methodology. *J. Amer. Soc. Inform. Sci.*, Jan. 1980.
- [Wel89] R.J. Welke. Meta systems on meta models. Technical report, CASE Outlook, Dec. 1989.
- [WFP83a] A. Wasserman, P. Freeman, and M. Procella. Characteristics of software development methodologies. In T. W. Olle, H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 37–58. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.
- [WFP83b] A. Wasserman, P. Freeman, and M. Procella. Characteristics of software development methodologies. appendix: Software development methodology—questionnaire. In T. W. Olle, H. G. Sol, and



- C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 59–62. Elsevier Science Publishers, B. V. (North-Holland). IFIP., 1983.
- [WG84] W. M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, 1984.
- [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.
- [Wil88] L. Williams. Software process modeling: A behavior approach. In *Proceedings of the 10th International Conference on SE*, pages 174–186, 1988.
- [YC79] E. Yourdon and L. L. Constantine. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [YT86] S. S. Yau and J. J. Tsai. A survey of software design technique. *IEEE Transaction on Software Engineering*, 12(6):713–721, June 1986.