
[All ETDs from UAB](#)

[UAB Theses & Dissertations](#)

2012

An Information Theory Based Representation of Software Systems and Design

Zekai Demirezen
University of Alabama at Birmingham

Follow this and additional works at: <https://digitalcommons.library.uab.edu/etd-collection>

Recommended Citation

Demirezen, Zekai, "An Information Theory Based Representation of Software Systems and Design" (2012).
All ETDs from UAB. 1508.
<https://digitalcommons.library.uab.edu/etd-collection/1508>

This content has been accepted for inclusion by an authorized administrator of the UAB Digital Commons, and is provided as a free open access item. All inquiries regarding this item or the UAB Digital Commons should be directed to the [UAB Libraries Office of Scholarly Communication](#).

AN INFORMATION THEORY BASED REPRESENTATION
OF SOFTWARE SYSTEMS AND DESIGN

by

ZEKAI DEMIREZEN

ANTHONY SKJELLUM, COMMITTEE CHAIR

MEHMET AKSIT

BARRETT R. BRYANT

THAMAR SOLORIO

MURAT M. TANIK

CHENGCUI ZHANG

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2012

AN INFORMATION THEORY BASED REPRESENTATION
OF SOFTWARE SYSTEMS AND DESIGN

ZEKAI DEMIREZEN

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

Software designers can benefit from the experience of engineering designers and information theory formalism. Every design starts with uncertainty and the art of software design involves uncertainty reduction. Information theory enables software designers to adopt a systems view that facilitates intellectual control over a given software design. Since design imposes organization through successive transformations in reaching the final product, it is possible to formalize design with information theory.

We investigated software design as a hierarchical decomposition of design spaces. We realized that before initiating the software design process there is minimal organization, representing higher entropy. The design decisions carrying out design activities through hierarchical decomposition reduce uncertainty and therefore introduce comparatively higher organization represented by lower entropy.

In this dissertation, the communication channel representation of software is developed through a process of 1) set-theoretical representation, 2) mapping to a communication channel formalism, and 3) hierarchical decomposition leading to entropy reduction. This information theoretical representation allows investigating the properties of software systems as communication channels.

DEDICATION

This work is dedicated to my mother, Hatice Demirezen.

ACKNOWLEDGMENTS

I would like to acknowledge many people for helping me during my doctoral work. I would like to express my gratitude to my committee chair and co-advisor Dr. Anthony Skjellum who provided his guidance and supported me throughout the process. I owe thanks and gratitude to my advisor Dr. Murat M. Tanik for being such a wonderful mentor. I want to acknowledge Dr. Mehmet Aksit for his advice, guidance, and support. I have benefited tremendously from our initial discussions on design space and his writings on design space analysis. I am also thankful to Dr. Murat N. Tanju for the countless hours of discussions, reviews, and his wisdom.

Also appreciated are the advice and encouragement of my committee members, Dr. Barrett R. Bryant, Dr. Chengcui Zhang, and Dr. Thamar Solorio. I would like to thank Dr. Purushotham Bangalore for his guidance as the graduate program coordinator. Finally, I want to acknowledge Dr. Jeff Gray who initially provided the opportunity for me to study abroad.

I wish to thank my friends and colleagues Dr. Arda Goknil, Dr. Geylani Kardas, Dr. Abidin Yildirim, Dr. Ozgur Aktunc, Dr. Bunyamin Ozaydin, Mr. Tomaz Lukman, and Dr. Robert Tairas for their enthusiastic discussions and help throughout this study. I am also very thankful to my sister, Zekiye, who has always been selflessly caring, supportive and encouraging. I express my gratitude to Mrs. Oya Tanik for her encouraging comments. Finally, I would like to thank to all my family, teachers, students, and friends, who supported me and may not be listed here.

TABLE OF CONTENTS

	<i>Page</i>
ABSTRACT	iii
DEDICATION	iv
ACKNOWLEDGMENTS	v
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF TERMS	xiii
 CHAPTER	
1 INTRODUCTION	1
1.1 Background, Opportunity, Challenges	1
1.2 Dissertation Statement	8
1.3 Proposed Approach and Solution	9
1.4 Overall Contributions	14
1.5 Impacts	14
1.6 Organization of Dissertation	15
2 THE QUANTITATIVE STUDY OF INFORMATION	17
2.1 History	17
2.2 Information Concepts	20
2.3 Communication Concepts	30
2.3.1 Communication System	30
2.3.2 Classification of Channels	32
2.4 Summary	35
3 THE SYSTEMS PERSPECTIVE	36
3.1 History	36
3.2 Nature of Hierarchical Systems	38

3.3	Formal Treatment of Hierarchical Systems	40
3.4	Decomposition of Hierarchical Systems	43
3.5	Summary	45
4	SOFTWARE DESIGN	46
4.1	General Design Principles	46
4.2	Software Design Principles	48
4.2.1	Design Specification Languages	50
4.2.2	Methods for Representing Semantics/Behavior	52
4.2.3	Control Abstractions	53
4.2.4	Coupling and Cohesion	55
4.3	Contemporary Design Decomposition Approaches	56
4.3.1	Objects	56
4.3.2	Design Patterns	57
4.3.3	Aspects	58
4.3.4	Domain-Specific Modeling Languages	58
4.4	Design Space Analysis	59
4.5	Summary	60
5	INFORMATION THEORETICAL ANALYSIS OF SOFTWARE SYSTEMS	62
5.1	Software Design is an Entropy-reduction Activity	64
5.1.1	Structural Spaces	66
5.1.1.1	Structural Entity Space	66
5.1.1.2	Structural Relation Space	66
5.1.2	Behavioral Spaces	67
5.1.2.1	Behavioral Flow Space	67
5.1.2.2	Behavioral Expression Space	69
5.2	Information Theoretical Approach	71
5.2.1	Set-theoretical Representation of Software Systems	72
5.2.2	Channel Formalism of Software Systems	74
5.2.3	Hierarchical Decomposition of Software Systems	77
5.3	Summary	78
6	EXAMPLES, RESULTS, AND ANALYSIS	80
6.1	Design Space Example	81
6.2	Information Theoretical Analysis Example	86
6.3	Summary	96
7	SUMMARY, CONCLUSIONS, AND FUTURE WORK	97
7.1	Summary and Conclusions	97
7.2	Future Work	100

7.2.1	Information Theory and Representation of Computer Programs as A Partition Transformation	101
7.2.2	Information Theory and Representation of Computer Programs as A Generalized Communication System	104
LIST OF REFERENCES		109
APPENDIX		120
A	JAVA IMPLEMENTATION	120
B	COLLECTED DATA	144

LIST OF TABLES

<i>Table</i>	<i>Page</i>
1.1 Combinations of Priority Level with Respect to Design Quality Attributes . . .	3
1.2 Design Steps in Traditional Engineering Disciplines vs. Unified Software Development Process	7
2.1 Product of Partition X and Y , $X \cdot Y$	26
4.1 The Three Constructs of Programming and Their Equivalent CCFGs	54
5.1 Mapping Between Problem Space Concepts and Solution Space Concepts . . .	65
5.2 Values of $V1 - V8$	74
6.1 A Design of Sort Behavior for an Array Class	85
6.2 Example Software System	87
6.3 Software Concepts to Set Concepts	87
6.4 Part of the Observed Values	89
6.5 Transmission Between Variables	90
6.6 Decomposition of Interaction Among Variables and Subsystems	92
7.1 Contributions and Associated Impacts	99
7.2 An Instance of ADD Program for $\{X, Y \mid 0 < x < 4 \ \& \ 0 < y < 4\}$	102

LIST OF FIGURES

<i>Figure</i>	<i>Page</i>
1.1 Engineering Process: Transformation of Needs, Requirements, Constraints into a Product	2
1.2 Design as an Uncertainty Reduction Process	5
1.3 Block Diagram of Design process	6
1.4 Decomposition of a System	10
1.5 Analysis of Software Systems	13
2.1 Sample Partition	21
2.2 Product of Two Partitions	22
2.3 The Function $h(p) = -p \log p - (1 - p) \log(1 - p)$	24
2.4 Information Quantities	30
2.5 Communication System	31
2.6 The Flow of Information Through a Channel	32
2.7 Classification of Channels	33
2.8 A Coin Tossing Example	34
4.1 Classification of Specification Languages	51
5.1 Structural Entity Space	67
5.2 Structural Relation Space	68
5.3 Behavioral Flow Space	69
5.4 Behavioral Expression Space	70
5.5 Software Design Activities Transforms Uncertainty to Certainty	71

5.6	Conceptual Representation of Set-theoretical Representation of Software System	73
5.7	A Communication Channel	75
5.8	Communication Between V3 and V4	75
5.9	Conceptual Representation of Channel Representation of Software Systems	77
5.10	Conceptual Representation of Hierarchical Decomposition of Software Systems	78
6.1	An Overall Diagram of Successive Design Decisions Leading to the Final Product	82
6.2	Structural Entity Design Space for the Library Example	83
6.3	Structural Relation Design Space for the Library Example	83
6.4	Flow Design Space for the Library Example	84
6.5	Expression Design Space for the Library Example	84
6.6	Hierarchical Representation of the Example	88
6.7	Actual Communication Between V1 and V2	89
6.8	Transmission Between Variables	90
6.9	Decomposition of Interaction Among Variables	91
6.10	Communication Between Two Subsystems	91
6.11	Conceptual Approach	93
7.1	Interpretation of Computer Programs as a Partition Transformation	102
7.2	Partition Transformations for the ADD Program	103
7.3	Communication Channel Representation of Program Constructs	104
7.4	Representation of Computer Programs as A Generalized Communication System	105
7.5	Partition Transformation for Coin Example	106

7.6	Fourier Transformation of the Inputs	106
7.7	Inverse Transformation of the Output	107
7.8	Communication-channel Representation of Coin Example	107
7.9	Polynomial Multiplication	108
7.10	Communication-channel Representation of Polynomial Multiplication	108

LIST OF TERMS

Bounded Rationality is the idea that one needs to have sufficient cognition, time, and information to reach a rational decision.

Cohesion is a measure of the strength of associations of elements within a system.

Communication Channel is a mathematical object that connects input variables to output variables in a probabilistic manner.

Complex System is a system composing a large number of components interacting in an irreducible way.

Complexity Analysis is the study of the relationships between elements within systems and the interactions among the systems.

Coupling is a measure of the strength of interconnection among subsystems.

Design is the transformation of existing conditions into preferred conditions.

Design Space Decomposition is a process of decomposing a design space toward a reduced entropy state.

Engineering is the study and practice of developing solutions to technical problems that are timely, cost-effective, and reliable.

Entropy is a measure of the degree of disorderliness.

Entropy Reduction is the process of reducing entropy through a series of actions.

Hierarchical Decomposition is the process of decomposing a system into subsystems.

Hierarchical System is a system organized into a hierarchical structure where its leaves are non-decomposable elements.

Information Theory is a field of mathematics that studies the quantification of information.

Internal Informational Exchange represents an interaction of elements within a system.

Module-based Design is the process of decomposing software systems into modules.

Nearly Decomposable Systems are systems with weak subsystem interactions.

Object-Oriented Design is the process of decomposing a software system into a set of objects with well-defined interfaces.

Organized System is a low entropy state of a system.

Parameter of Interest corresponds to a designer's intention and includes the criteria that will drive the design.

Partition is a collection of mutually exclusive events.

Semantic Gap characterizes the distance between problem space and solution space.

Software Engineering is the study and practice of a systematic, scientific, disciplined approach to developing software systems that are timely, cost-effective, and reliable.

Software System is a collection of algorithms that transform a set of inputs into desired outputs.

System is a set of interacting elements acting toward a goal.

Systems Theory is the interdisciplinary study of systems.

Uncertainty indicates the lack of knowledge about the occurrence or nonoccurrence of any event or the difference between modeled and actual states of a system.

Uncertainty Reduction is the resolution toward a lower entropy state.

Chapter 1

INTRODUCTION

In this dissertation, we study how to make software design more scientific and precise than it currently is.

1.1 Background, Opportunity, Challenges

Engineering is the study and practice of developing solutions to technical problems that are timely, cost-effective, and reliable [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Engineers solve technical problems by applying mathematical and scientific knowledge to develop artifacts (products). To achieve their goals, they follow appropriate engineering processes to create reliable technical products with economic value under constraints of time and resources [5, 6, 8, 11]. A process starts with the problem definition and recognition of other constraints (economic, technical, *etc.*), and ends with the production of artifact(s) with economic value. Within the engineering process, a clear distinction is made between design and manufacturing [5, 12]. Design Engineers/Designers are responsible for the abstract creation of a product that involves proof-of-concept prototyping and/or various forms of simulation and modeling [12, 13]. On the other hand, the physical realization of a product is the task of manufacturing engineers¹ [3]. Figure 1.1 represents the engineering process, which implicitly includes a multitude of internal processes.

¹A discussion of the manufacturing engineering process is not applicable to this introduction since our focus is on processes involved in design.

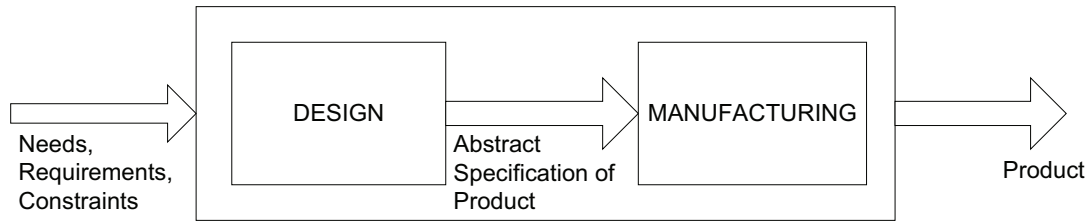


Figure 1.1: Engineering Process: Transformation of Needs, Requirements, Constraints into a Product (Adapted from [12])

The design process in engineering is fundamentally driven by engineering design quality attributes [9]. Design quality attributes set a realistic limit on expenditures, an expectation of completion time, and an expected reliability bound/lifetime for the product. Table 1.1 shows the spectrum of emphasis with respect to the key engineering design quality attributes cost, timeliness, and reliability [9]. Within this spectrum, efforts with low priority for cost, timeliness, and reliability are generally considered “research-oriented” activities. For pure research, for example, the main focus is on experimenting with minimal consideration for cost, issues, delivery times, and operational reliability to design an alternative artifact. On the other hand, high priority for the three quality factors is related to “pure” engineering activities. In pure engineering, the emphasis is on delivering a theoretically ideal product in the sense of the lowest possible cost, and the highest possible reliability within a minimum delivery time. All other possible combinations in between are shown in Table 1.1. Cost, Timeliness, or Reliability emphasis provide foundations of making decisions when compromises are required; this is where engineering judgment plays an important role.

Table 1.1: Combinations of Priority Level with Respect to Design Quality Attributes
(Adapted from [9])

	Cost	Timeliness	Reliability
Pure Research	low priority	low priority	low priority
Reliability Emphasis	low priority	low priority	high priority
Timeliness Emphasis	low priority	high priority	low priority
Timeliness/Reliability Emphasis	low priority	high priority	high priority
Cost Emphasis	high priority	low priority	low priority
Cost/Reliability Emphasis	high priority	low priority	high priority
Cost/Timeliness Emphasis	high priority	high priority	low priority
Pure Engineering	high priority	high priority	high priority

In each of the categories of Table 1.1 there are associated processes [14]. Manipulating these underlying processes impacts the resulting product quality attributes. Processes generally contribute to the quality of the product while consuming resources to varying degrees [14].

Process-centered engineering focuses on improving software processes to achieve higher quality products while maintaining the same quantity of resources [14]. It is accepted that even a simple process improvement may have a significant impact on the quality of a product [5].

Requirements, needs, and constraints are inputs to the engineering process [3, 5, 8]. These inputs reveal that some action must take place to satisfy the given requirements and produce quality artifacts [12]. First, a designer takes the requirements and constraints as inputs and makes a representation of an artifact to be constructed by a manufacturing engineer [8, 12]. The representation ranges from specifications in formal languages, such as design languages, to informal and visual descriptions, such as engineering drawings [12, 15]. In Figure 1.1, design output is shown as an abstract specification of a product. An

abstract specification of a product is the solution of a design problem and a product can be created from it [12].

In traditional engineering disciplines, such as mechanical, civil, and electrical, design is considered to be a fundamental activity [6]. The act of design starts with recognition of a *design problem* [5, 6, 8, 12, 15, 16]. A designer determines the problem according to his or her *parameter(s) of interest* [17]. A *parameter of interest* corresponds to a designer's judgment and includes the criteria that will drive the design. After analyzing a problem, the designer conceives of a solution or family of solutions that will correct or improve the current situation [12, 18]. A solution given by the designer can be interpreted as a specification for transformation [12, 18]. The solution process is a series of transformations following the prescribed engineering process, yielding a quality artifact.

The *nature of design*, and the expansion of the design concept towards a relatively more abstract domain were studied by Tanik and Ertas [9]. Following these authors, three axioms for the nature of design activities can be described as follows:

- The Axiom of Hierarchy or Specification Axiom: There are limitations on the cognitive and information processing capabilities of designers. No humans or collection of any number of them can specify or design an artifact at once. As the system becomes more complex, the human cognitive element becomes a bottleneck. Therefore, in order to exercise intellectual control on the design, hierarchies are a necessity.
- The Axiom of Feedback: Design solutions are achieved through an iterative process. This process is evolutionary in nature and includes feedback to proceed to the next iteration (reducing uncertainty). Rapid and successive feedback promotes the refinement of a design [13] and will produce better designs in a shorter period of time.

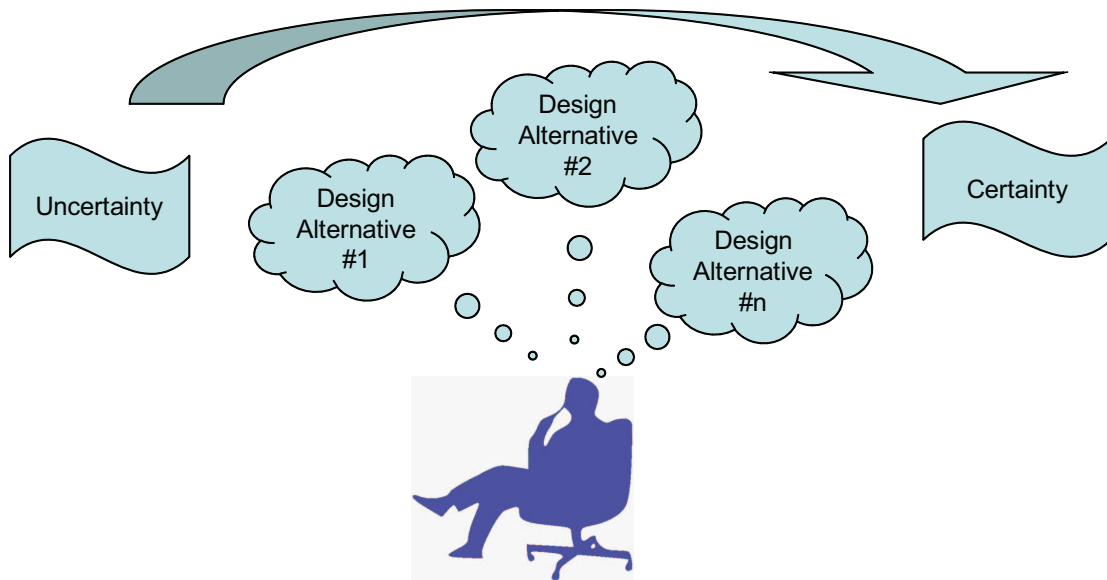


Figure 1.2: Design as an *Uncertainty Reduction Process*

- The Axiom of Automation: Automated tools are required to provide a fast and reliable design process, remaining sources of mistakes that lead to residual faults in artifacts.

The *design activity* can also be viewed as a problem solving activity [5]. Designers generate various alternative approaches to solve the given problem. During evaluation, designers consider that which is feasible and narrow the space of alternative designs [5, 6, 7, 8, 19]. There are usually several feasible alternatives. The designer is required to make decisions based on many parameters and to make choices among possible alternatives, while evaluating the feasibility of each choice [5, 8]. The optimum choice among them often is not obvious. This situation reflects the *uncertainty* that designers encounter in finding a solution. Every design decision resolves some part of an unclear situation and reduces the number of possible alternatives using the framework outlined in Table 1.1. Thus, each design activity can be considered as an *uncertainty reduction* process as shown in Figure 1.2.

The problem-solving process can also be viewed as consisting of successive steps with increasing precision in the context of relationships [2]. The four phases of this

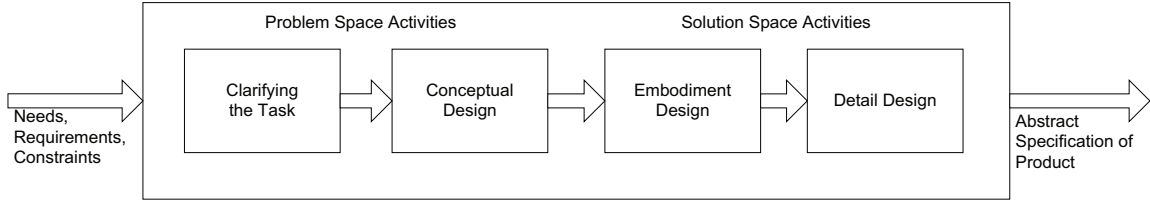


Figure 1.3: Block Diagram of Design process (Adapted from [5])

process are as follows: *Clarifying the Task*, *Conceptual Design*, *Embodiment Design*, and *Detail Design* [5]. Designers elicit customer requirements in the first phase. Engineers collect information about constraints and their importance. Requirements and Constraint specification is the output of this phase. *Conceptual design* occurs when a designer abstracts essential concepts, uses analysis techniques to decompose the given problem, establishes function structures, and generates all the concepts required for the working structure. In this phase, the designer considers a number of solution alternatives and narrows them down to a single solution. In *Embodiment Design*, designers unite all the concepts and produce a technical layout [5, 6, 8]. Finally, in *Detail Design*, designers complete the details of all individual components and realize them [5].

During the design process, designers work on two different spaces: *Problem Space* and *Solution Space* [19]. Problem Space includes only the details from business/customer domain [19]. On the other hand, Solution Space includes technical terms and incorporates solution details [19]. Each space has its own representation [3, 19]. The designer uses problem-space representation during *Clarifying the Task* and *Conceptual Design* steps [5, 19]. On the other hand, *Embodiment Design* and *Detail Design* are solution-space activities [5, 19]. The transition from problem-space to solution-space occurs between *Conceptual Design* and *Embodiment Design* [5, 19]. This transition is not an easy task and is characterized as a *semantic gap* in computer science [20]. Figure 1.3 depicts the steps.

The term *Software Engineering* was first coined in 1968 to emphasize the need for modern engineering techniques and methods for software development [21]. Tools and effective methods for the activities in software development have been devised and led to

Table 1.2: Design Steps in Traditional Engineering Disciplines vs. Unified Software Development Process (Adapted from [24])

Traditional Engineering Disciplines	Unified Software Development Process
Clarifying the Task	Requirements
Conceptual Design	Analysis
Embodiment Design	Design
Detail Design	Implementation & Testing

a better understanding of software process. However, today there is still a need to improve our understanding about software development, especially in the area of software design.

The software development life cycle, given in Booch [22], incorporates the following:

- Requirements: establish agreement on what the system should do, and properties that are nonfunctional as well;
- Analysis: study requirements, and analyze the problem space to create conceptual models;
- Design: transform the analysis specification into a design specification. The design specification includes implementation details and instructions;
- Implementation: code and integrate the design resulting in an executable system;
- Test: verify and validate the software product; and
- Deployment: make the software product available to its end user.

There is a particular difference in design between traditional engineering disciplines and software engineering. In software engineering, design refers to a step between analysis and implementation. In traditional engineering, design covers all the steps starting from requirements to the final output of the design process. Table 1.2 shows a correspondence between traditional design steps [5] and the Unified Software Development Process (USDP) [23].

1.2 Dissertation Statement

Software engineering in particular is an engineering discipline whose focus is the production of high quality software systems [17, 21, 25]. A software system is a collection of algorithms/programs designed to transform a set of inputs into desired outputs [15]. Software development is in a “pre-engineering” phase that is analogous to many pre-engineering phases found in engineering disciplines of the past [26].

Software Engineers have mostly applied ad-hoc, experience-based design techniques to date. Most of these practices have on the whole resulted in unreliable and costly products [27]. Furthermore, because of hardware technology advances, the size and quantity of software systems have greatly increased. Yet, our ability to reduce the unit software error rate has not improved correspondingly [28]. Recent observations conclude that the fundamental causes of software development failures today are the same failures caused in traditional engineering fields of some 100 years ago [26]. As Royce stated as early as 1968, design principles similar to “hard” engineering principles are needed to overcome these problems [25]. These principles must be supported by mathematical underpinnings. Although there are approaches for science-based design [7, 15], a complete mathematical theory of software design currently does not exist. Until such time as one is defined, tested, and broadly accepted, we need techniques to minimize the potential damage of poor designs. The proposed techniques should be defined in a formal way in order to provide a basis for establishing more complete design theories in the future.

It is clear that we need to make design a systematic engineering activity in software engineering. It is possible to make software design significantly more systematic by combining existing concepts from the theory of decomposition of complex systems, by introducing the novel idea of software design as an entropy-reduction process, and by employing information theory to connect software design with a communication channel abstraction.

The next sections will outline our approach to achieve the groundwork for such an important goal. The result will be a better understanding of the design process in software engineering based on the first principle foundations of science and the practices of “hard” engineering disciplines.

1.3 Proposed Approach and Solution

Through the use of formal mechanisms, this dissertation proposes to systematize software design. As indicated in the foregoing, each design activity can be considered to be an *uncertainty reduction* process as shown in Figure 1.2. We introduce formal mechanisms for uncertainty reduction in this dissertation. Historically, this process of transforming uncertainty to certainty is considered a concern of complexity analysis [29, 30]. Therefore, from a complex-system perspective, the software design problem is a form of complexity analysis and system decomposition. Complexity analysis and system decomposition together provide a combined mechanism to deal with transforming uncertainty to certainty [31].

In general, a complex system is composed of correlated elements [32, 33, 34]. The mathematical concept of a complex system is related to the degree of correlation between elements [30, 32, 33, 34, 35], which indicates the breadth and depth of interactions of elements constituting a complex system. We can use *structure* to show the correlation between elements. For example, gas molecules show apparent weak structural organization [36]. Each gas molecule is nearly independent of the other molecules. On the other hand, crystal molecules are correlated with each other and show highly organized macro structure. An organized structure includes redundancy and the amount of redundancy reduces the required information to reveal that structure. For example, determination of the position of each gas molecule is more difficult than determination of a crystal molecule [36]; with prior knowledge of a few molecules’ positions, it is easy to locate other molecules in a crystal [36]. By way of comparison, in communication theory, the structure

of a complex system corresponds to the structure of a message and redundancy within a message [37]. The gas and crystal analogs demonstrate that revealing the structure of an apparently disorganized system requires more information than revealing the structure of a conceptually more organized system would require. This fact corresponds to the amount of uncertainty in the system [36]. The mathematical concept of organization is closely related to the measure of uncertainty [30, 32, 33, 34, 35].

According to Simon, Ashby, Wiener and others [29, 33, 34, 38, 39], a common attribute of complex systems is that “Complexity takes the form of hierarchy” from the observer’s point of view [39]. Two important points that need to be specified for the definition of a hierarchical system follow: a) A first requirement is that all subsystems be correlated with each other, and the correlation strength imply the structure of the system [39, 40]. Simon denotes these as *nearly decomposable* to emphasize the correlation between subsystems; b) A second requirement is that there always be a terminating level of decomposition from which no further decomposition is feasible. Figure 1.4 illustrates the correlation between components and the decomposition of a system. To reduce complexity, strongly connected elements are grouped within subsystems [30, 39, 41]. This highlights that correlations among subsystems are weaker than correlation within subsystems. In Figure 1.4, correlations between strongly connected elements are represented with wider arrows.

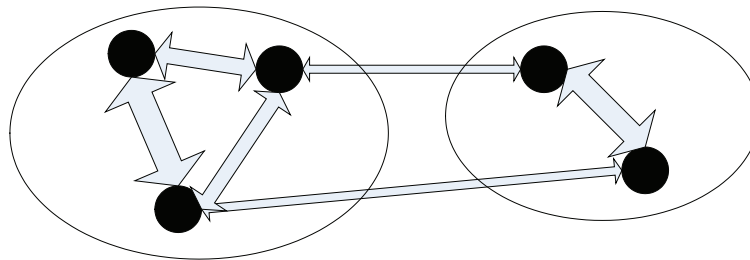


Figure 1.4: Decomposition of a System (Adapted from [30])

Traditionally, a two-pronged approach is taken to study software design - Static and Dynamic [22, 42]. Static aspects of design consist of analysis of data and their relationships [22, 42]. More specifically, in the case of object-oriented design, static analysis includes

data-specification issues, relationships between data, and encapsulation of them [22]. Dynamic aspects on the other hand focus on run-time issues that are composed of behavioral specifications [22, 42].

Our modeling of software design is fundamentally based on hierarchical decomposition and driven only by communication among the system attributes. Therefore, all analysis is dynamic and multivariate by definition. In our modeling approach, as described in this dissertation, the multivariate correlations among variables are modeled via information theory [33, 38]. Total correlation over the complex system is the sum of the total correlation within the subsystems plus the correlations among the subsystems [29, 35, 36, 43]. Furthermore each subsystem can be broken down into further subsystems and the fundamental rule holds in turn for the subsubsystems and their correlations [30, 41]. One of the basic criteria for evaluation of the decomposition is that the correlation among the subsystems be insignificant compared to the total correlation [30, 43].

Figure 1.5 presents four transition steps for the analysis of software systems via information theory. We start with mapping of software systems to set-theoretical representations. Software systems are represented with an arbitrary number of variables. Each variable is observed once per standard time increment, for example, at the end of a user event for a GUI application. These values are shown as a table in Figure 1.5. In the second transition, which is the mapping of set-theoretical representation to a channel formalism, we show the information transfer between variables and demonstrate the correlations among variables as communication channels. As a result of this step, we have a communication-channel representation of a software system. The third step, hierarchical decomposition, takes the channels as input and applies decomposition techniques to find subsystems. In Figure 1.5, this step is represented as a transformation between channels and set-subsets (hierarchical) combinations. Finally, Figure 1.5 demonstrates mapping between set-subsets and software structures. These steps provide a formal means for the representation and analysis of software design decomposition.

In the preceding paragraphs we introduced a plan to apply formal methods to software design. In our hypothesis, software design principles are consistent with general design principles. Therefore, following Rothstein, Simon, and Ashby [29, 34, 39], we consider the generation of design alternatives as a process of uncertainty reduction. We hypothesize that successive applications of information-theoretical multivariate analysis [38] would be the appropriate mathematical approach to achieve this goal. In a sense, these techniques help to transform uncertainty to certainty. Therefore, these techniques are applicable beyond the object-oriented systems that we used as a motivating case.

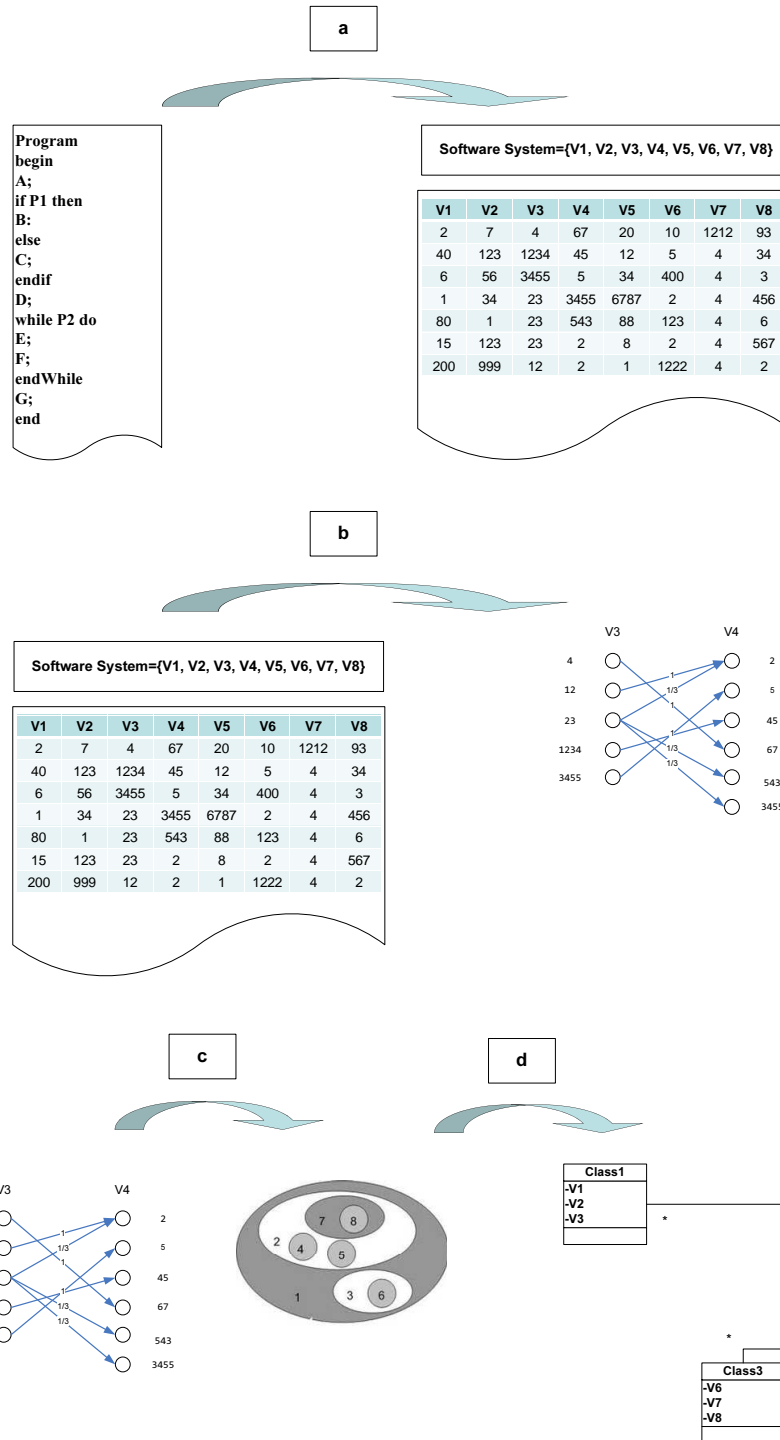


Figure 1.5: Analysis of Software Systems: a) mapping of software systems to set-theoretical representations, b) mapping of set-theoretical representation to channel formalism, c) hierarchical decomposition, d) mapping between set-subsets and software structures

1.4 Overall Contributions

The major contributions of this dissertation include 1) *design decomposition of software*, and 2) *communication-channel modeling of software*. Theoretical aspects of these contributions will be explored in Chapter 5 followed by representative examples demonstrating their utility in Chapter 6. Following Simon [39], in the context of hierarchical systems and nearly decomposable systems, and following Ashby [34], in the context of multivariate analysis, we develop a system representation for software using set-theoretical representations. This representation enables us to study software systems as communication channels. Following Simon [39], and Aksit [44], we present design of a software system as a decomposition process. We show that software development is fundamentally the design of a system and system design is in turn a decomposition of design space [44]. Following Prather [45], and Simon [39], we analyze decompositions. Finally, following Conant [41], we use information theory to quantify decomposition of software systems. As a result, we demonstrate feasibility and value of the communication channel modeling of software. Thereby we bring a mathematical formalism to software design, this work also reveals the value of mathematical formalism.

1.5 Impacts

This dissertation outlines a research approach chosen to investigate and develop a formal yet widely usable means to represent software systems. The results have an opportunity for transformative impact that would influence the area of software engineering by providing a more relatively stable context for discussing the representation of software systems than currently available. In this research, static and dynamic aspects of software systems are studied and a mathematical formalism is provided for their representations. The proposed mathematical formalism will be suitable for further design space analysis. The proposed formalization will have significant impact on current practice of software engineering

in terms of modeling and controlling the design alternatives through the application of information-theoretical approaches and techniques. These techniques will lead to better understanding of the design process in software engineering based on first-principles foundations of science as well as originating from those practices of “hard” engineering disciplines. Moreover, they will provide a capability for reasoning about software designs, so they support software designers’ modeling activities.

Another contribution of the work reveals the need for a mathematical theory to analyze the semantic decomposition of software systems. This dissertation discusses the requirements of using information and coding theory during software development. These outcomes will have an impact on future research that focuses on the analysis of the relationship between decomposition at run-time and programming statements.

In summary, the representation and analysis of software systems as integrated communication systems opens up possibilities for applying engineering mathematical analysis to software development. This means that the elusive concept of software formally represented in the past with automata and formal languages can evolve into the type of inquiry involving classical engineering mathematics. These notions are expanded in Chapter 5 and 7.

1.6 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 discusses the quantitative study of information and establishes the basic notations used in the following chapters.

Chapter 3 introduces hierarchical systems. Set-theoretical representation of complex systems, Information-Theoretical correlation analysis, and decomposition of correlations are detailed in that chapter.

Chapter 4 discusses software design, the area of Computer Science that we are advancing in this dissertation. Since the focus of this dissertation is on the analysis of software design, all the details are focused towards that direction.

We incorporate theoretical aspects of the design space decomposition using information flow in Chapter 5, while presenting representative examples in Chapter 6.

This dissertation concludes in Chapter 7 with the summary of the work, the contribution to the body of knowledge, and future research directions based on this work. The appendices consist of source code and collected data.

Chapter 2

THE QUANTITATIVE STUDY OF INFORMATION

The purpose of this chapter is to provide background research on information theory essential for the development of the dissertation, as well as to develop the necessary information-theoretic concepts and the notation to be used. Section 2.1 provides a historical perspective. Section 2.2 provides information concepts such as entropy, interaction, and correlation formulas. Section 2.3 defines communication concepts that are useful in the study of software systems in the succeeding chapters.

2.1 History

The beginnings of the development of information theory can be traced to the initial considerations for the development of the concept of entropy. Theoretical contributions involving entropy functions started in the original investigation of heat phenomena [46, 47]. After the invention of the steam engine, physicists focused on the transformation of heat into mechanical work, and vice versa in order to build better steam engines. Scientists of the time tried to accomplish this by moving from physical reality to the development of theoretical underpinnings. They thought that heat was a weightless substance and transferred from one body to another body [48]. In the mid 1800s, Carnot explained the limitations in the heat-work transformation using a flowing substance model. He observed that some energy is lost even in the most efficient engine possible [49, 50]. Eventually,

Clausius formulated the dissipation of useful energy in terms of a new quantity which he denoted *Entropy* [46, 47, 49].

Following Carnot's observation, Maxwell [51], Boltzmann [52], and Gibbs [53] defined heat as disordered motion of atoms and molecules with consideration of the atomic nature of matter [46]. Since the "gold standard" at the time was to reduce all known phenomena to mechanical motion, their approach to modeling was successful because it explained the heat phenomena through mechanical principles and the aid of the Entropy concept [47, 49, 50]. Their foundational investigations eventually initiated a new branch of mechanics, called *Statistical Mechanics* [46]. In his investigations of the second law of thermodynamics, Maxwell introduced a "gedanken experiment" involving an imaginary being, which was later called *Maxwell's demon* [49, 54]. Maxwell demon is used as a model to this day for the hypothetical investigation of the potential violation of the second law. Another leading physicist in the nineteenth-century, Ludwig Boltzmann, studied *Entropy* as being a measure of degree of orderliness or disorderliness of gas molecules [47, 52].

Szilard further developed the entropy concept emphasizing the information perspective [55]. He demonstrated that Maxwell's demon gets its information at the expense of entropy increase elsewhere [36, 55]. He showed the conceptual equivalence between information acquisition and thermodynamic entropy [49, 55]. Following up with Szilard work, Landauer established the connection between the notions of energy consumption, information, and computation [56]. He showed that the erasure of information requires energy dissipation [54, 56].

In the field of communication, Nyquist [57] and Hartley [58] introduced a quantification technique to measure the information in a message. Their approach used the logarithm of the number of all possible messages [37, 59, 60]. It took about two decades after Hartley's 1928 paper for the introduction of a general theory called *Communication Theory*, by Shannon [37]. He constructed a mathematical model for communication systems and discussed the amount of uncertainty in the output of a source and defined it as the

information content of a message [61, 62, 63, 64, 65]. He explained the calculation of channel capacity using time durations of channel symbols and constraints among them [37, 63, 66, 67]. He demonstrated fundamental theorems for noiseless and noisy channels and established the transmission rate limit for a given channel and a source. He noted the possibility of maximum message transmission rate with proper encoding of source messages and established the existence of a coding system to transmit a source message over the noisy channel with an arbitrarily small probability of error [37].

Almost immediately, applications of Shannon's approach to diverse fields started to appear in the literature. For example, in the field of psychology, the first information-theory related paper was written by George A. Miller and Frederick C. Frick in 1949 [68]. They proposed a method for quantifying organization in sequences of events [68, 69]. They used information theory to analyze serial dependencies in chain of responses. Their paper opened up the possibility of various application areas of the entropy formula in the field of psychology as well as other related disciplines [69, 70]. Garner and Hake published an article studying the amount of information in stimulus and responses relations [71]. Before their paper, variance analysis was already a widely used technique in psychology [71, 72]. Their work initiated uncertainty analysis of experimental data based on information theory [38, 69, 70]. Eventually, Garner and McGill established the relation between information and variance analysis [72]. All these works focused on Shannon's information measures that includes two variables, the sender's and the receiver's state. William McGill presented an extension of Shannon's measures and put the information relations between three and four variables [38]. He also developed the associated quantitative formulations of transmission, interaction, and correlation concepts for multivariate analysis [38, 69, 70]. Furthermore, Fred Henry Quastler [73] and Attneave [69] contributed to various applications of information in psychology. The early adoption of Shannon's approach in psychology opened up numerous other applications in related fields that proved fruitful for further developments.

2.2 Information Concepts

The approach of this dissertation is based on applications of information-theoretic principles to the field of software design. In this section, we introduce information concepts, definitions and essential notation which are useful in the study of software systems in the succeeding chapters.

Definition 2.2.1. A *set* is a collection of objects called *elements*. A *subset* B of a set A is another set if every element of B is also an element of A . All sets under consideration are included in a single set S that is called the *universal set*. The *empty set* is the set that contains no elements. The *union* of two sets A and B is a set consisting of all the elements that belong to A or to B or to both. The *intersection* of two sets A and B is a set whose elements are in both A and B . The *complement* A' of a set A is the set of all elements of S that are not in A . Two sets A and B are called *mutually exclusive or disjoint* if they have no common elements. A *partition* U of a set S is a disjoint collection of non-empty subsets A_i of S whose union is S . *Cartesian product* of the sets S_1 and S_2 is a set of ordered pairs a_1a_2 where a_1 is any element of S_1 and a_2 is any element of S_2 .

Example. Let A_i be defined as the set of all integers which leave the remainder i on division by 4. A_0, A_1, A_2 , and A_3 are mutually exclusive sets and their union is the set of integers.

Definition 2.2.2. Following [74], let S be a collection of elements denoted *elementary events*, and let δ be a set of subsets of S ; the elements of the set δ will be denoted *random events*. Assign to each set A in δ a non-negative real number $P(A)$ that will be denoted as *the probability of the event*. This number has been chosen to satisfy the following three conditions [74]:

- $P(A) \geq 0$
- $P(S) = 1$
- If event A and event B have no element in common, then $P(A \cup B) = P(A) + P(B)$

Example. In tossing a fair coin¹, there are two elementary events, “head” and “tail.” Hence $S = \{Head, Tail\}$, $\delta = \{\{\}, \{Head\}, \{Tail\}, S\}$, $P(\{Head\}) = \frac{1}{2}$ and $P(\{Tail\}) = \frac{1}{2}$.

Definition 2.2.3. The *Conditional Probability* [75, 76, 77] of an event A assuming M , $P(A | M)$, is by definition the ratio

$$P(A | M) = \frac{P(A \cap M)}{P(M)}. \quad (2.2.1)$$

Example. In rolling a fair die, there are six elementary events, each equally likely. Let f_i be the faces of the die, then $S = \{f_1, f_2, f_3, f_4, f_5, f_6\}$. The conditional probability of the event $T = \{f_3\}$ assuming that the event $O = \{f_1, f_3, f_5\}$ occurred is $P(T | O) = \frac{P(T \cap O) = P(T)}{P(O)} = \frac{1}{3}$.

Definition 2.2.4. Two events A and B are called *independent* if $P(A \cap B) = P(A)P(B)$. If A and B are independent events then $P(A | B) = P(A)$ and $P(B | A) = P(B)$.

Definition 2.2.5. Here we adopt the definition of [65]. A *partition* is a collection of mutually exclusive events whose union is S . Figure 2.1 shows a partition.

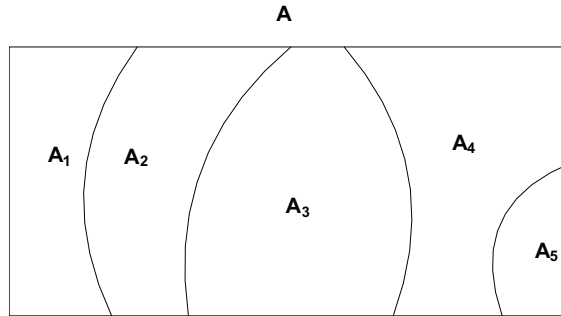


Figure 2.1: Sample Partition (Adapted from [65])

A partition U consisting of the events A_i will be denoted by the notation $U = [A_1, \dots, A_k]$ or simply $U = [A_i]$. Events A_i will be denoted *elements* of U .

- A *binary partition* is a partition with only two elements. $U = [A, A']$ is a *binary partition* consisting of the event A and its complement A' .

¹A fair coin is a mathematical object with two mutually equiprobable outcomes.

- An *element partition* is a partition whose elements are the elementary events $\{\zeta_t\}$ of the space S . It will be denoted by V .
- A partition B is a *refinement* of a partition U if each element B_j of B is a subset of some element A_i of U . The notation $B \prec U$ will be used to indicate that B is a refinement of U . Thus

$$B \prec U \text{ iff } B_j \subset A_i.$$

- The *product* of two partitions $U = [A_i]$ and $B = [B_j]$ is a new partition consisting of the elements that are all intersections $A_i B_j$ of the elements of U and B . This partition will be denoted by $U \cdot B$. The product of two partitions is shown in Figure 2.2.

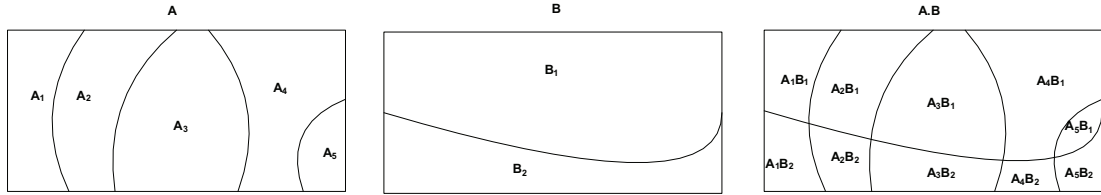


Figure 2.2: Product of Two Partitions (Adapted from [65])

Definition 2.2.6. Two partitions $U = [A_i]$ and $B = [B_j]$ are called *independent* if the elements A_i and B_j are independent for every i and j .

Definition 2.2.7. *Experiment E* consists of determining which of the events A_1, A_2, \dots, A_n occurs. A_1, A_2, \dots, A_n are called the possible outcomes of experiment E . These outcomes form a partition of the set S . A single performance of an experiment is called a *trial*.

Definition 2.2.8. Given n experiments E^1, E^2, \dots, E^n , that is, n partitions

$$S = A_1^{(i)} + A_2^{(i)} + \dots + A_{r_i}^{(i)} \quad i = 1, 2, \dots, n \quad (2.2.2)$$

of the set S [74]. Then n experiments E^1, E^2, \dots, E^n are called *mutually independent experiments* [65, 74, 76, 77, 78] when

$$P(A_{q_1}^{(1)}, A_{q_2}^{(2)}, \dots, A_{q_n}^{(n)}) = P(A_{q_1}^{(1)})P(A_{q_2}^{(2)}) \dots P(A_{q_n}^{(n)}) \text{ for any } q_1, q_2, \dots, q_n. \quad (2.2.3)$$

Definition 2.2.9. *Combined Experiment F* is the *cartesian product* of two experiments E_1 and E_2 whose events are all cartesian products of the form $A \times B$ where A is an event of E_1 and B is an event of E_2 , and their unions and intersections [65].

Definition 2.2.10. A *random variable* is a number $x(\zeta)$ assigned to every outcome ζ of an experiment [65, 74, 77].

Definition 2.2.11. *Entropy of a partition U* is a measure of uncertainty about the occurrence or nonoccurrence of any event A_i of a partition U . It will be denoted by $H(U)$ [65].

The postulates of the Entropy function, H [37]

1. $H(U)$ should be a continuous function of $p_i = P(A_i)$.
2. If $p_1 = \dots = p_n = 1/N$, then $H(U)$ should be a monotonic increasing function of N ; that is, for $U = [A_1, \dots, A_m]$ $p(A_1) = \dots = p(A_m) = 1/M$ and $Y = [B_1, \dots, B_n]$ $p(B_1) = \dots = p(B_n) = 1/N$ $M < N$ implies $H(U) < H(Y)$.
3. If one of the elements of U be broken down into two successive events, then a new partition B is formed and $H(B) \geq H(U)$.

Remark. Properties of Entropy [65]:

1. $H = 0$ if and only if all the p_i but one are zero,
2. Given a partition $U = [A_1, A_2, \dots, A_n]$, if A_1 be broken down into the elements B_a and B_b and be formed a new partition $B = [B_a, B_b, A_2, \dots, A_n]$ then $H(U) \leq H(B)$,
3. if $B \prec U$ then $H(B) \geq H(U)$,
4. For any U $H(U) \leq H(V)$ where V is the element partition,

5. Given a partition $U = [A_1, A_2, \dots, A_n]$, the entropy of U is maximum when $p(A_1) = \dots = p(A_n) = 1/n$.

Definition 2.2.12. *Shannon's Entropy Formula* [37] is a measure of the entropy of a partition U that is by definition the sum²

$$H(U) = -p_1 \log p_1 - \dots - p_N \log p_N = \sum_{i=1}^N \varphi(p_i) \quad (2.2.4)$$

where $p_i = P(A_i)$ and $\varphi(p) = -p \log p$. Since $\varphi(p) \geq 0$ for $0 \leq p \leq 1$, it follows from 2.2.4 that $H(U) \geq 0$ [65, 79, 80].

Example. For a binary partition $U = [A, A']$ and $P(A) = p$,

$$H(U) = -p \log p - (1-p) \log(1-p) \equiv h(p). \quad (2.2.5)$$

The function $h(p)$ is shown in Figure 2.3 for $0 \leq p \leq 1$. This function is symmetric, convex, and it reaches its maximum at the point $p = 0.5$ [79].

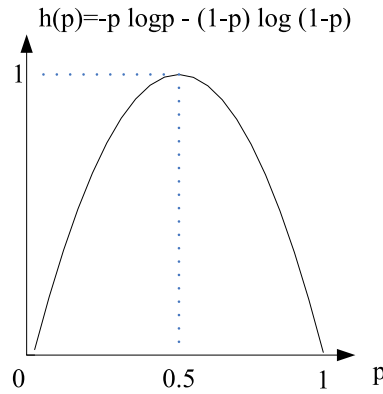


Figure 2.3: The Function $h(p) = -p \log p - (1-p) \log(1-p)$

The probability $P(A)$ of an event A measures the uncertainty concerning the occurrence or nonoccurrence of event A in a single performance of the underlying experiment S [65].

²Unless otherwise specified, we shall take logarithms to the base 2. The units of H are called *bits*.

For example, when $P(A) = 0.999$, we are almost certain that A will occur. On the other hand, occurrence of event A and our uncertainty change when $P(A) = 0.1$; now we are reasonably certain that A will not occur. Our uncertainty is maximal if $P(A) = 0.5$ [59].

The quantity $H(U)$ measures the uncertainty concerning the result of an experiment. There is an uncertainty surrounding the events A_i of the partition U prior to the performance of the underlying experiment. When the experiment has been carried out and the results concerning A_i become known, such uncertainty is resolved. Thus one can say that entropy measures the amount of information obtained [59, 64, 65]. The information provided by the experiment about the events A_i equal to the entropy of their partition and is measured by the sum given in equation 2.2.4.

Example. In a fair die, $P\{1\} = P\{2\} = P\{3\} = P\{4\} = P\{5\} = P\{6\} = 1/6$. Let f_i be the faces of die, the entropy of the partition $U = [f_1, f_2, f_3, f_4, f_5, f_6]$ in the fair-die experiment is as follows :

$$H(U) = -\frac{1}{6} \log_2 \frac{1}{6} - \dots - \frac{1}{6} \log_2 \frac{1}{6} = \log_2 6.$$

When the fair-die experiment is performed and the result is observed, then we gain information about the partition U equal to its entropy $\log_2 6$.

Definition 2.2.13. The *conditional entropy* [59, 65, 79, 80] of a partition U assuming M is by definition the sum

$$H(U | M) = \sum_{i=1}^{N_U} P(A_i | M) \log P(A_i | M) \quad (2.2.6)$$

where $P(M) \neq 0$, N_U is the number of elements in partition U , and $P(A_i | M) = \frac{P(A_i M)}{P(M)}$. $H(U | M)$ measures the uncertainty concerning the partition U on the average assuming partition M .

Remark. Properties of Conditional Entropy [65]

1. $H(U \mid M) \leq H(U)$
2. If U and M are independent partitions then $H(U \mid M) = H(U)$
3. If $B \prec U$ and B is observed, then $H(U \mid B) = 0$

Example. In a rolling die experiment, the prior partition is $U = [f_1, f_2, f_3, f_4, f_5, f_6]$ and $H(U) = \log_2 6$. When the experiment is performed and when we are informed that “even event,” denoted by E , or “odd event,” denoted by O , is revealed, then the posterior partition is $V = [E, O]$ and $H(V) = \log_2 2$. The conditional entropy is $H(U \mid V) = -(\frac{1}{3} \log_2 \frac{1}{3} + \frac{1}{3} \log_2 \frac{1}{3} + \frac{1}{3} \log_2 \frac{1}{3}) = \log_2 3$. The difference between the uncertainty of U and V is $\log_2 6 - \log_2 2 = \log_2 3$, and it is equal to the uncertainty about V assuming U .

Definition 2.2.14. The *joint entropy* [70, 79, 80, 81] of partitions U and M is by definition

$$H(U \cdot M) = H(U) + H(M \mid U) = H(M) + H(U \mid M). \quad (2.2.7)$$

Remark. Properties of Joint Entropy [65]

1. For any U and B , $H(U \cdot B) \geq H(U)$ and $H(U \cdot B) \geq H(B)$,
2. $H(U \cdot B) \leq H(U) + H(B)$,
3. $H(U \cdot M) = H(U) + H(M)$ when U and B are independent partitions.

Example. Two partitions $X = [a_1, a_2, a_3, a_4]$ with $p(a_1) = \frac{1}{2}, p(a_2) = \frac{1}{6}, p(a_3) = \frac{1}{6}, p(a_4) = \frac{1}{6}$, and $Y = [b_1, b_2]$ with $p(b_1) = \frac{1}{3}, p(b_2) = \frac{2}{3}$ are given. The product of two partitions and probabilities are shown in Table 2.1. Hence $H(X) \approx 1.79$, $H(Y) \approx 0.91$.

$H(X \mid Y) = \sum_{i=1}^2 p(Y = i) H(X \mid Y = i) = \frac{1}{3}(\frac{3}{4} \log_2 \frac{3}{4} + \frac{1}{4} \log_2 \frac{1}{4}) + \frac{2}{3}(\frac{3}{8} \log_2 \frac{3}{8} + \frac{1}{8} \log_2 \frac{1}{8} + \frac{2}{8} \log_2 \frac{2}{8} + \frac{2}{8} \log_2 \frac{2}{8}) \approx 1.54$, and hence $H(X \cdot Y) = H(Y) + H(X \mid Y) \approx 2.45$.

Table 2.1: Product of Partition X and Y , $X \cdot Y$

	a_1	a_2	a_3	a_4
b_1	$\frac{1}{4}$	$\frac{1}{12}$	0	0
b_2	$\frac{1}{4}$	$\frac{1}{12}$	$\frac{1}{6}$	$\frac{1}{6}$

Definition 2.2.15. The *relative entropy* [79, 82] is a measure of the distance between two partitions. Given a partition $U = [A_1, A_2, \dots, A_n]$ $p_i = P(A_i)$ and a partition $B = [B_1, B_2, \dots, B_N]$ $q_i = P(B_i)$, where $i = 1 \dots n$, relative entropy is defined as

$$D(U \parallel B) = \sum_{i=1}^n p_i \log \frac{p_i}{q_i}. \quad (2.2.8)$$

Remark. Properties of Relative Entropy [79]

1. $D(U \parallel B) \geq 0$,
2. $D(U \parallel B) \neq D(B \parallel U)$, and
3. Does not satisfy triangle inequality [83].

Relative Entropy can be interpreted as the notion of *gain of information*. This formula explains the heuristic idea of the distance between two partitions and gain of information. We are restating this derivation from Renyi's book [59]. Given a set E containing N elements and a partition of this set E_1, \dots, E_n . The number of elements of E_k is denoted by N_k thus $N = \sum_{k=1}^n N_k$. The probability of each element in the partition is given as $p_k = \frac{N_k}{N}$. An element of E chosen at random can be specified using its index number in E . And also, it can be specified by giving the set E_k to which it belongs and its index number within E_k . The amount of uncertainty of the index number in E for the chosen element is equivalent to sum of the uncertainty of the set E_k to which chosen element belongs and the uncertainty of its index number within E_k . Then we have

$$H(\xi) = H(\eta) + H(\zeta \mid \eta) \quad (2.2.9)$$

where the index number in E is denoted by ξ , the index k of the relevant set E_k is denoted by η , and the index number within E_k is denoted by ζ . Clearly $H(\xi) = \log_2 N$, $H(\eta) = \sum_{k=1}^n p_k \log \frac{1}{p_k}$, $H(\zeta \mid \eta) = \sum_{k=1}^n p_k \log_2 N_k$.

A nonempty subset of E , denoted by E' , is given. We are informed that an element chosen at random belongs to E' . What amount of information is provided about the uncertainty of the set E_k to which chosen element belongs? Let the intersection of E_k and E' be denoted by E'_k . Let N'_k be the number of elements of E'_k . Then we have $\sum_{k=1}^n N'_k = N'$, $q_k = \frac{N'_k}{N'}$, and $\sum_{k=1}^n q_k = 1$. The uncertainty of the set E_k to which chosen element belongs is denoted by the random variable η . The partition denoted by η is changed because of the new information. We are looking for measurement of this change in the partition resulting from the knowledge that an element chosen at random belongs to E' . The knowledge that an element chosen at random belongs to E' contains the information $\log_2 \frac{N}{N'}$. This information is the summation of two quantities. The first one is the amount of change in the partition η and the second amount is the information about ζ when η is already known. The amount of information gain about η is the difference between the prior and posterior distributions of η . Hence

$$\log_2 \frac{N}{N'} = D(Q \| P) + \sum_{k=1}^n q_k \log_2 \frac{N_k}{N'_k}. \quad (2.2.10)$$

Since $\sum_{k=1}^n q_k = 1$ and $\frac{NN'_k}{N'N_k} = \frac{q_k}{p_k}$ then $D(Q \| P) = \sum_{k=1}^n q_k \log_2 \frac{q_k}{p_k}$, which is equal to the 2.2.8.

Definition 2.2.16.

$$T(U:M) = H(U) - H(U | M) \quad (2.2.11)$$

is called the *mutual information* [33, 37, 38, 60, 61, 64, 69, 70] of the partitions U and M . The observation of M reduces the uncertainty about U from $H(U)$ to $H(U | M)$. This reduction is the mutual information between two partitions. Mutual information can be interpreted as the information about U contained in M .

Remark. Properties of Transmission [34, 38, 61]

1. $T(U : M) \geq 0$,
2. $T(U : M) = T(M : U)$, and

$$3. T(U : U) = H(U).$$

Definition 2.2.17. *Interaction* is the information between two of the partitions, due to additional knowledge of the third partition [34, 38, 69]. It is defined as

$$Q(U : A : B) = -H(U \cdot A \cdot B) + H(U \cdot A) + H(U \cdot B) + H(A \cdot B) - H(U) - H(A) - H(B). \quad (2.2.12)$$

Interaction formula can be expressed in terms of the entropy formula as follows [84]

$$Q(A^1 : A^2 : \dots : A^n) = \sum_{k=1}^n \triangle_{nk} \left\{ \sum_{\text{all } k\text{th order in } r} H(k \text{ partitions}) \right\} \quad (2.2.13)$$

$$\text{where } \triangle_{nk} = \begin{cases} -1 & \text{for even } r - k \\ 1 & \text{for odd } r - k \end{cases}$$

Figure 2.4 demonstrates certain quantities of entropy with a Venn diagram . The left circle is the information we get from M , it is the uncertainty about M . The right circle is the information we get from U . The overlap of the two circles represents the common uncertainty. Hence transmission between these two partitions is shown as the intersection of the two circles. The left half of the circle is the uncertainty from M alone. That part is denoted by $H(U | M)$ and it is the average amount of uncertainty that remains to be gotten from M after U is already known. The total area enclosed in both circles represents all the uncertainty that both U and M includes.

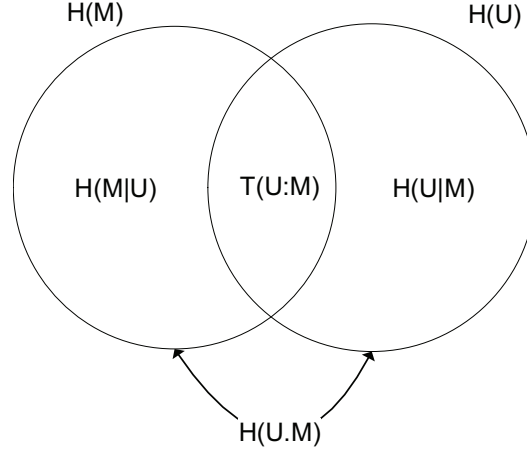


Figure 2.4: Information Quantities (Adapted from [81])

Definition 2.2.18. Transmission/Mutual Information function can be generalized to an arbitrary number of partitions. *Correlation* [29, 38, 69, 70] among partitions U_1, U_2, \dots, U_n is the total information transmission and by definition as follows:

$$C(U_1, U_2, \dots, U_n) = \sum_{i=1}^n H(U_i) - H(U_1 \cdot U_2 \cdot \dots \cdot U_n). \quad (2.2.14)$$

2.3 Communication Concepts

2.3.1 Communication System

Information theory is concerned with the quantitative analysis of an entity called *communication system* [37, 63] which is shown schematically in Figure 2.5. It consists of five parts. The *information source* [37] is responsible to generate messages to be communicated to the receiver. The *source encoder* [37] is responsible to encode the given message for transmission over the channel. The *channel* [37] is defined as the medium in which the information is transmitted from the source to the receiver. The *receiver decoder* [37, 63] is responsible for decoding the transmitted message. The *receiver* [37] is the actual message destination. The *source symbols* [37] are the symbols that were generated and sent by the source, and the *receiver symbols* [37] correspond to the symbols the sent symbols

map. The *noise* [37] that is always present in the noisy channels causes the symbol at the output of the source encoder to change during the transmission process.

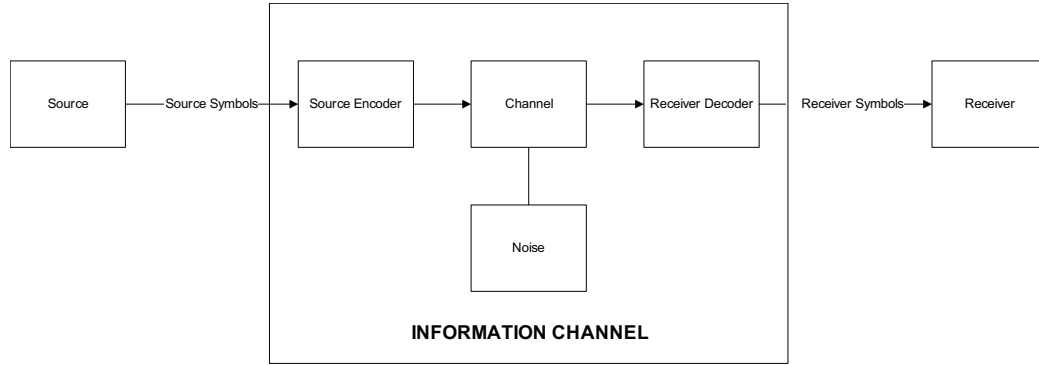


Figure 2.5: Communication System (Adapted from [37])

Formal representation of information flow through a communication channel is depicted in Figure 2.6 [84]. The uncertainty concerning the source partition X , denoted by $H(X)$, is called *source entropy* [60, 61, 63, 81]. It is the uncertainty concerning which symbol will be transmitted. The receiver partition Y consists of all the possible symbols that will be received. The amount of uncertainty in the receiver part, denoted by $H(Y)$, is called *receiver entropy* [60, 61, 81]. Therefore, $H(Y)$ may include uncertainty which the sender does not account. The conditional entropy $H(Y | X)$ is the measure of this uncertainty and it is equivalent to *noise*. In other words, part of the source entropy may not be received by the receiver because of noise. The quantity $H(X | Y)$ is the average amount lost, and it is called *equivocation* [61, 63, 84]. The *amount of information transmitted* [61, 63], $T(X : Y)$ Eq.(2.2.11), is the uncertainty shared by both source and receiver partitions. The *capacity* of a channel is the maximum possible rate of transmission [37].

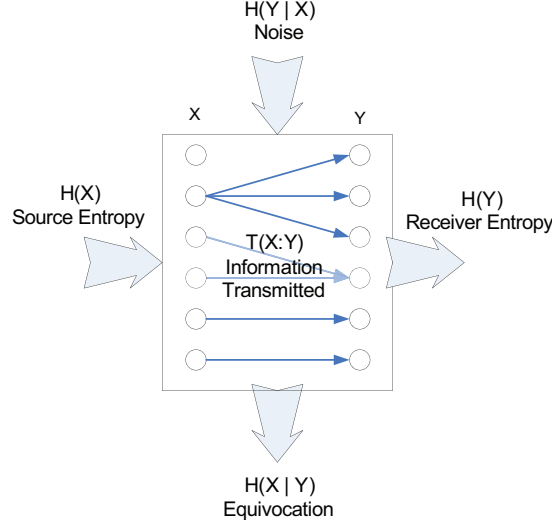


Figure 2.6: The Flow of Information Through a Channel (Adapted from [84])

Definition 2.3.1. *Redundancy* is the difference between the capacity of a communication channel and the rate of transmission [37, 63, 81].

$$R(X : Y) = T_{max}(X : Y) - T(X : Y) \quad (2.3.1)$$

2.3.2 Classification of Channels

Following [63], a communication channel can be classified as follows:

- A *lossless channel* is a channel in which output determines the input [63]. There are no transmission errors in lossless channel and $H(X | Y) = 0$ for all input distributions. This channel is also called a *noise-only channel* [84]. A lossless channel and corresponding Venn diagram for information quantities are shown in Figure 2.7a.
- A *deterministic channel* is a channel where $p(y_j | x_i) = 1$ or 0 for all i, j ; that is $H(Y | X) = 0$ for all input distributions [63]. This channel is classified as *equivocation only channel* [84]. Figure 2.7b demonstrates a deterministic channel.
- A channel is *noiseless* if it is lossless and deterministic. An example of a lossless channel is given in Figure 2.7c.

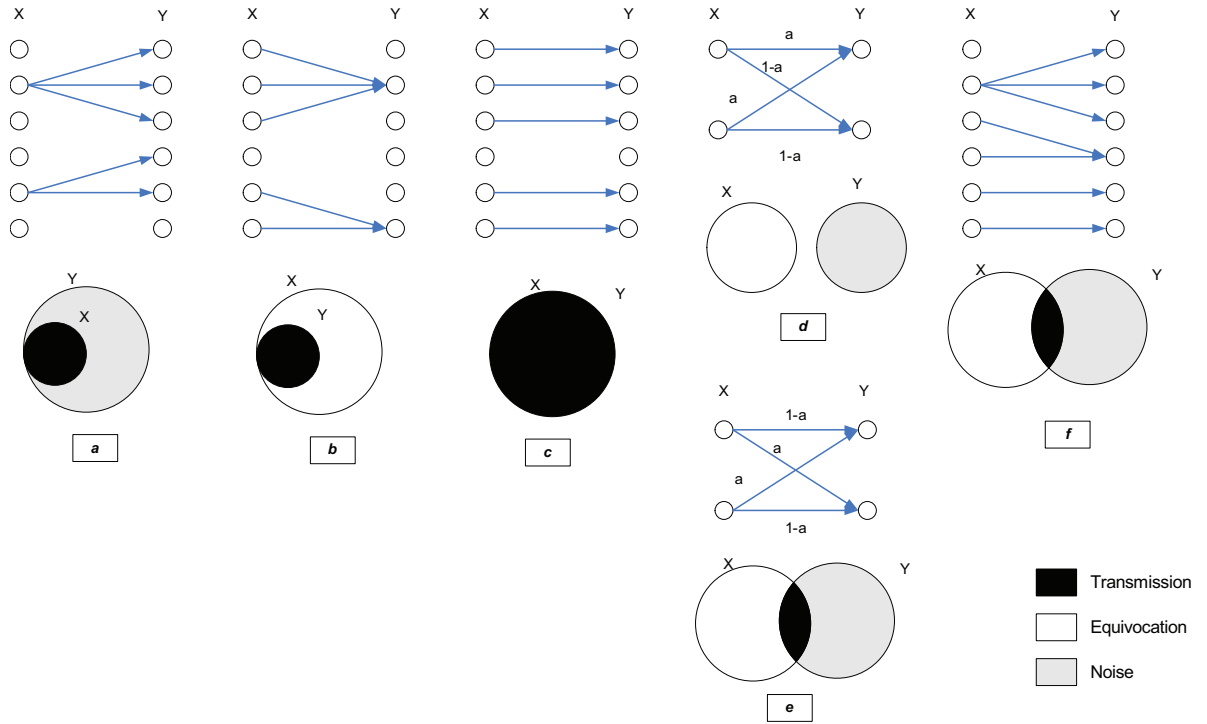


Figure 2.7: Classification of Channels: a) Lossless Channel or Noise Only Channel b) Deterministic Channel or Equivocation Only Channel c) Noiseless Channel d) Useless Channel e) Symmetric Channel f) Mixed Channel

- A *useless (or zero-capacity) channel* is a channel where $H(X | Y) = H(X)$ and $T(X : Y) = 0$ for all input distributions [63]. In a useless channel the output provides zero information about the input. An example of a useless channel is shown in Figure 2.7d.
- A channel is *symmetric* if $H(Y | X)$ is independent of the input distribution $p(x)$ [63]. In a symmetric channel, conditional entropy $H(Y | X)$ depends only on the channel probabilities $p(y_j | x_i)$. Binary Symmetric Channel [61] is shown in Figure 2.7e.
- A mixed channel has a mixture of noise and equivocation [63]. An example of a mixed channel is shown in Figure 2.7f.

Example. We're restating the example found in [63]. This example is selected to demonstrate the information flow between input and output. In this experiment, there are two coins available, one unbiased and a two-headed coin. In a single trial of the experiment,

a coin is selected at random and tossed twice, and the number of heads is recorded. The number of heads convey information about the identity of the coin. The number of heads obtains in two tosses of the coin is random variable which will be denoted by Y . The identity of the coin is a random variable which will be denoted by X . For the unbiased coin $X = 0$ and for the two-headed coin $X = 1$. A diagram representing the experiment is shown in Figure 2.8. What amount of information will be furnished hereby about the identity of the coin by the number of heads recorded?

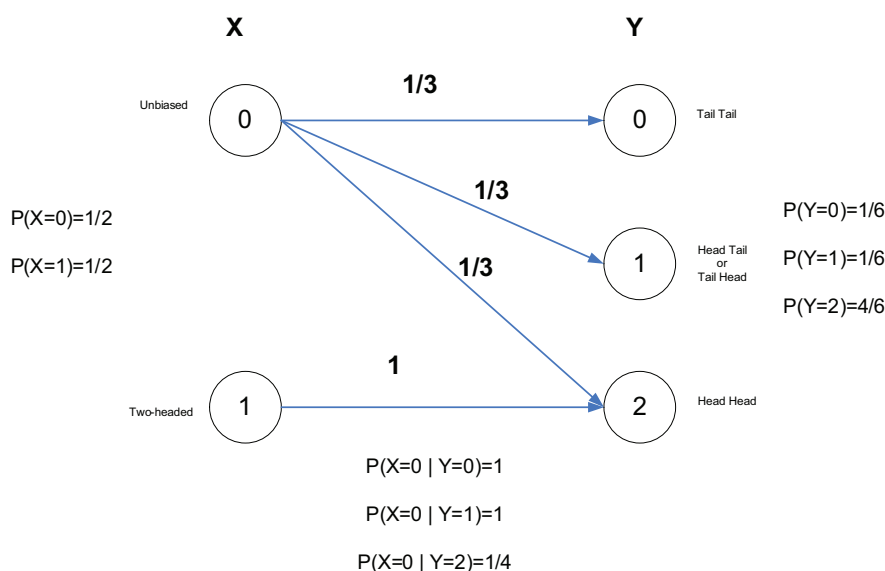


Figure 2.8: A Coin Tossing Example (Adapted from [63])

Intuitively, one can conclude that the unbiased coin was used when less than two heads are observed, however if both throws resulted in heads, the output of the experiment favors the two-headed coin. The quantitative analysis of this experiment is given using entropy and transmission calculations. The prior uncertainty concerning the identity of the coin is $H(X) = \log_2 2 = 1$, after the experiment is performed, the uncertainty about the identity of coin is $H(X | Y) = \frac{1}{6}(0) + \frac{1}{6}(0) - \frac{4}{6}(\frac{1}{4} \log_2 \frac{1}{4} + \frac{3}{4} \log_2 \frac{3}{4}) \approx 0.54$, and the information transmitted about X by Y is $T(X : Y) = H(X) - H(X | Y) \approx 0.46$.

2.4 Summary

The key concepts to be used in the following chapters were reviewed in this chapter. For the suitability to our application to software design we followed Papoulis approach [65] by defining entropy as the measurement of uncertainty about the partition. Since decomposition is the activity of partitioning a system into its successive subsystems, this line of analysis would be suitable when the notion of decomposition is used as a bridge between software design and entropic analysis for quantification purposes. Before the discussion of decomposition as systematic partitions, the details of information functions was given in this section. Utilization of entropy functions will be introduced in the following chapters.

Chapter 3

THE SYSTEMS PERSPECTIVE

This chapter will present a mathematical concept of organization and of systems used to model software design in the following chapters. Our modeling of software design is based on hierarchical decomposition and driven only by communication among the system attributes. From a complex-system perspective, the software design problem is a form of complexity analysis and system decomposition. This chapter introduces representation and analysis of hierarchical systems.

Historically there has been confusion of terminology in this area. One reason for this confusion originates from the disciplinary roots of the investigators, which includes physics [29, 33], mathematics [34], and economics [39]. The notions of *Complex Systems*, *Organized Systems*, and *Organization* are treated as equivalent concepts in this chapter. Consequently, they are used interchangeably throughout this dissertation.

3.1 History

Following Boltzmann's principles [52], the basic idea that underlies statistical mechanics is that an organized system has a lower entropy than a disorganized one [29, 36]. The difference between these two systems can be defined as a reduction in the entropy, and the difference can be calculated by the methods of statistical mechanics [47]. In statistical mechanics, an organized system is composed of ordered molecules [46, 47]. The states of particles and correlation among these particles was demonstrated with *Entropy* term [32,

33]. Watanabe demonstrated the information calculation as the measure of organization [33, 36, 85]. Rothstein used the redundancy calculation to demonstrate the organization [29, 32, 40, 86]. In communication theory, correlation is defined with the *Redundancy* term [37, 61, 63, 66, 79]. At the early stages of information theory, redundancy was studied and defined as the relationship between information and correlation [37].

Ashby made major contributions to the information-theoretical analysis of complex systems [34, 87, 88, 89]. He defined a complex system as a set of variables with constraints [90]. He mentioned that the presence of organization arises from communication between variables. In his 1965 paper [34], he examined the constraints as multivariate relationships within a system and denoted them as *Internal Informational Exchange*. He showed that when the variables are related, constraints exist and they can be quantified with information theory [34, 87, 88]. He also discussed how to measure information exchange within systems that actively change in time [34]. Although *Shannon's Communication Theory* [37] deals with source and receiver states without considering real-time issues, Ashby investigated temporal correlations in organizations [34]. After these results, he further demonstrated the decomposition of systems as an information-flow partition according to temporal and spatial information exchange [87].

Ashby's ideas led to the development of new areas of study, including within *General System Theory* [91]. He influenced system researchers in the area of *Reconstructability Analysis* [92]. The reconstructability problem is a methodological problem for reconstruction of an overall system from a given set of variables [92]. Several notable system researchers undertook the informational-theoretical investigation of system structure. Klir studied the processing of activity arrays through the use of information theory [93]. Broekstra used the term *Constraint Analysis* to indicate the application of information theory in the reconstructability problem [94]. While he discussed the disjoint partition of constraints to represent a structure system, he also developed the non-disjoint partition formalism. Krippendorff extended Ashby's work by defining new information

functions, such as *Informational Distance* and *Informational Bias* [84]. He noted the concept of *information calculus* and *lattice based representation of structure* within his spectral analysis papers. He pointed out the limitations in the interaction function [84]. Other contributors, such as Conant [41, 43, 95, 96], Gaines [97], Cavallo [98] , and Uyttenhove [99] applied information theory in the reconstructability problem. Conant discussed measurement of interaction between subsystems, information transfer through finite state systems, and channel capacity of automata in his dissertation. He applied information-theoretical measurement techniques to regulatory processes [43]. Later, he studied pairwise interaction of variables in a dynamic system. Following Ashby and Simon, Conant provided a technique to decompose a system into weakly connected subsystems [41]. His technique detects subsystems of a complex system while quantifying the interactions among the variables.

3.2 Nature of Hierarchical Systems

Simon, in his development of a science of design, investigated the nature of systems in general [18]. He defined *complex system* informally as the composition of large number of components interacting in a complex way [39]. Typically, in systems, the whole exhibits an emergent behavior and becomes more than a linear sum of the parts [35, 39, 43]. Therefore inferring the properties of the whole from the component interactions becomes difficult, even if the properties of the parts and the laws of their interaction should be known [35, 39, 43].

From Simon's perspective, complex systems are frequently hierarchical, exhibit a systems structure through evolutionary processes, are decomposable into components or subsystems, and because of their hierarchical nature, a relatively simple set of rules defines their behavior [39]. Therefore, complexity is organized as a form of hierarchy since complex systems are successively decomposed into lower and lower levels of subsystems [39]. By a *hierarchical system*, Simon expresses that a system is composed

of highly interrelated subsystems, organized into a hierarchical structure until reaching its non-decomposable components [35, 39, 43]. Because of the level of hierarchy, one can distinguish interactions among subsystems as well as interactions within subsystems and components [35, 39, 43]. Furthermore, it is possible that interactions among some subsystems can prove negligible compared with the interactions within their subsystems [39]. In such cases, one can make the assumption that they are independent systems. These systems can also be classified as *decomposable* into subsystems [39]. This leads to the classification of *nearly decomposable systems* as systems in which the interactions among the subsystems are weak, although the interactions cannot be neglected [39]. As such, Simon concluded that many complex systems are nearly decomposable and possess a hierarchical structure [39]. This observation indicated that the analysis of complex systems is feasible since one can differentiate their parts [35, 39, 43]. When Simon applied this classification to natural systems and concluded “If there are important systems in the world that are complex without being hierarchical, they may to a considerable extent escape our observation and our understanding. Analysis of their behavior would involve such detailed knowledge and calculation of the interactions of their elementary parts that it would be beyond our capacities of memory or computation” [39]. This naturally ties with his notion of *bounded rationality* in which he expounded that one needs to have sufficient cognition, time, and information to reach a rational decision [100].

From our viewpoint, a critical observation made in these instances is that complex structures in our natural environment are highly redundant, and this redundancy can be used to simplify, model, and understand their description in economical terms [39]. However, as many authors of the era, including Shannon, indicated, one must find the right representation to achieve this simplification [37, 39].

3.3 Formal Treatment of Hierarchical Systems

Watanabe and others introduced a formal treatment technique for the analysis of hierarchical systems following Simon's definition [29, 32, 33, 35, 36, 41, 43]. Their preferred starting point was the mathematical notion of the structure of organization, which was conceptually identical to the systems view of Simon. However, Watanabe and others took into account in their analysis the notion of uncertainty in the representation of their models [30, 32, 33, 34, 35]. Therefore, in this section we note that the mathematical tools developed in Chapter 2 are used in analyzing the organizational structure existing in a group of stochastically behaving components.

In this perspective, a complex system is composed of correlated subsystems or elements [32, 33, 34]. The formal analysis of a complex system is therefore, related to the degree of correlation among its subsystems or elements [30, 32, 33, 34, 35]. Naturally, correlation can indicate the level of depth and breadth of interactions among subsystems or elements. Therefore, this correlation can be used to show the degree of interaction of subsystems (or elements, or components) constituting a system. If we use the example of Watanabe, one can say that gas molecules, naturally disorganized in a container, would show weak interaction as such indicating a weak structural organization [36]. Technically, each gas molecule can be considered to be nearly independent of the other gas molecules. However, crystal molecules are correlated with each other according to the structural properties of individual crystals, and as such, show a highly organized structure. An organized structure includes redundancy and the amount of redundancy reduces the information required to reveal that structure [29, 33, 35, 36]. Therefore, structure provides the information to locate the position of other molecules in a crystal while the determination of the position of each gas molecule in a disorganized system becomes difficult [36]. If we recall Shannon's results at this point, we observe that in communication theory, the structure of a system implies the structure of a message and redundancy within the message [37]. The gas and crystal analogies create a visual demonstration, in which exposing the structure of a disorganized

system requires more information than revealing the structure of a more organized system. It is observed that this fact mathematically corresponds to the amount of uncertainty [29, 33, 34].

However, as Watanabe recognized, one should distinguish the amount of uncertainty in individual components from the amount of uncertainty of the whole system [33]. The systemic organization is a function of these two different types of uncertainty. The amount of uncertainty of individual components simply quantifies our ignorance about each component, while the essence of organization or emergent behavior of the system is in the correlation among subsystems or elements [33, 34, 35, 36, 41, 43, 87]. Therefore, a significant amount of uncertainty among individual elements does not imply an unstructured system [33, 36]. In an organized system, the amount of uncertainty of the whole could be low despite high ignorance about each individual component [33].

In Chapter 2, entropy was shown to be a good measure of uncertainty. We discussed various relations governing these entropy functions, $H(X)$, $H(X, Y)$ and $H(X | Y)$. Following Watanabe [36], we provide a derivation of the formula of degree of organization. An entropy function is non-negative

$$H \geq 0. \quad (3.3.1)$$

The definition of Conditional Entropy entails

$$H(X) + H(Y | X) = H(X, Y). \quad (3.3.2)$$

Equation 3.3.2 implies that

$$H(X, Y) \leq H(X) + H(Y). \quad (3.3.3)$$

Combining 3.3.2 and 3.3.3, we obtain

$$H(X | Y) \leq H(X). \quad (3.3.4)$$

Combining 3.3.1 and 3.3.2, we obtain

$$H(X) \leq H(X,Y). \quad (3.3.5)$$

It is customary to attach intuitive meanings to these formulas using the notions of *ignorance* and *information*. Watanabe uses the term *indeterminacy* interchangeably with *ignorance* [36]. For example, the left half of equation 3.3.2 can be interpreted as the total ignorance concerning the system X,Y which is the sum of the initial ignorance about the system X and the average remaining ignorance about the other system Y after we have determined the state of X [36]. Therefore, the total information about the system X,Y is the information obtained by observing Y and the information obtained by observing X after having obtained information about Y [36]. For example, equation 3.3.3 says that the information obtained by observing the total system X,Y is less than the sum of the information obtained from X and the information from Y separately [36]. Because of interdependence, there is an overlap between the information associated with X and the information associated with Y . For example equation 3.3.4 can be interpreted that the ignorance about X is larger or at least equal to the average remaining ignorance about Y after the information about X is known [36]. For example the relation in equation 3.3.5 expresses that the ignorance about a system X,Y is larger than , or at least equal to, the ignorance about its part X , or Y [36].

Utilizing the inequality in equation 3.3.3, one can introduce a non-negative quantity $C(X,Y)$ by $C(X,Y) = H(X) + H(Y) - H(X,Y) = H(X) - H(X | Y)$ and $C(X,Y) \geq 0$ where equality holds if and only if X and Y are independent [36]. On the other hand $C(X,Y) \leq \min(H(X), H(Y))$ if X dependent on Y then $C(X,Y) = H(X)$ If Y is dependent on X and X is dependent on Y then $H(X) = H(Y) = C(X,Y)$. Therefore the lower and upper bounds correspond to the minimum and maximum interdependence between X and Y respectively [36]. Thus the quantity $C(X,Y)$ is called “interdependence between X and

Y ,” $H(X)$ is the information obtained by observing X only, while $H(X | Y)$ is the average additional information obtained by observing X when the outcome of Y is known [36]. Only $H(X | Y)$ of $H(X)$ is the information exclusively carried by X itself unobtainable through Y . Therefore, $H(X) - H(X | Y)$ is the part of information that is doubly carried by both X and Y . This redundancy is attributed to interdependence [36].

Naturally, the degree of organization increases if the degree of ignorance of the system decreases despite a large degree of ignorance on individual components [33, 36]. Like Watanabe states [36] “If there is no organization, a large amount of indeterminacy of individuals will result in a large amount of indeterminacy of the whole.” Thus the strength of organization is measured by the balance between the indeterminacy of the components with respect to the indeterminacy of the whole [36]. Since entropy is a measure of ignorance or indeterminacy, then the degree of organization [36] can be defined as

$$\text{Organization} = (\text{sum of entropies of parts}) - (\text{entropy of whole}). \quad (3.3.6)$$

This formula is equivalent to *correlation formula 2.2.14* given in Chapter 2.

3.4 Decomposition of Hierarchical Systems

Decomposition of a complex system, in a sense, is a matter of identification of its components and their interactions [35, 41, 43]. A many-component system interacting in a complex way is naturally not conducive to the observation of its component interactions. Under these circumstances, the observer limits the observation to a few variables at a time to decompose the system [30, 35, 101]. The difficulty rests in the identification of all the system components, which is a requirement of decomposing the system into loosely coupled subsystems or elements [30, 39].

One can define the system as a set of variables and observe the correlation between variables [29, 33]. It is assumed that the information flow within the system is

representative of the relations between the variables [33, 34, 35, 87]. Therefore, given a system of variables, how do we decompose it into a set of subsystems or elements to minimize the information flow between the subsystems? As we have mentioned in previous sections, the answer lies in the hierarchical decomposition of the total correlation [33, 34, 35, 87]. As Van Emden states, “The effect of the complete decomposition is that correlation of the system is found to be equal to a sum of interactions” [35]. There are multiple ways of producing such a decomposition scheme [35]. It is a matter of the “parameter of interest” [17] to decide, which depends on the purpose of the analysis. One natural purpose could be to identify subdivisions into subsystems in which there is little interaction among subsystems compared to the amounts of interaction within subsystems. Applying such a decomposition scheme can also be viewed as a “divide and conquer” or “divide and rule” [35].

Alexander introduces a general algorithm for partitioning a system into its successive subsystems [30]. The algorithm is a hill-climbing procedure consisting of producing one-element subsets and computing the value of correlation for this partition followed by comparing with it all partitions that can be obtained from it [30]. The partition having the lowest value of correlation is then substituted for the original partition and the procedure repeats. The termination criterion is such that the algorithm continues until it arrives to a partition whose value of correlation is lower than that of any partition that can be obtained from it by combining two sets. Another hill-climbing procedure used is based on finding a tree of partitions directly [30]. This algorithm breaks a complete set into its two most independent disjoint subsets, by computing correlation for a random two-way partition. It then repeats this process for each of the two subsets obtained, successively partitioning each of them into two smaller subsets until the entire set is decomposed [30].

3.5 Summary

In this chapter, an overview on hierarchy of systems was given, primarily through Herb Simon's perspective. Then, formalization and quantification of hierarchical systems were summarized. The last section detailed Information-Theoretical analysis and decomposition of hierarchical systems. The mathematical concept of organization and decomposition given in this chapter will be used to model software systems as hierarchical systems in Chapter 5.

Chapter 4

SOFTWARE DESIGN

In this chapter we review general design principles from the perspective of design decomposition and related concepts. Our motivation to emphasize the design decomposition approach is to be able to represent design in terms of information theoretical notions. Since the heart of computing involves linguistic formalism we included linguistic approaches as well as the role of specification languages. We conclude with an assessment of the state of the art in design, open issues, and, a summary.

4.1 General Design Principles

It is well known that software design has benefited tremendously from the early considerations of engineering design methods, principles, and the concept of intellectual control over the developed design by human developers using various series of hierarchical abstractions. Engineering processes start with the problem definition and recognition of constraints (*e.g.*, economic, timeliness, technical) and conclude with a prototype production of desired artifact(s). Within an engineering process, a clear distinction is made between *design* and *manufacturing* [5, 12]. Design Engineers/Designers are responsible for the abstract creation of a product that involves proof-of-concept prototyping or various forms of simulation and modeling [12, 13]. On the other hand, the actual realization of a product is the task of manufacturing engineers [3].

Following Smith and Browne [19], *design problems* consists of five elements: goals, constraints, alternatives, representations, and solutions [19]. While *goals* comprise the specification of needs, *solutions* provide satisfaction of those goals. Designers normally generate various alternative approaches in order to solve the given problem. *Alternatives* are the possibilities designers identify and evaluate [18]. During evaluation, designers consider what is feasible and work to narrow the space of alternative designs [5, 6, 7, 8, 19]. *Constraints* specify the feasibility of alternatives [19]. There usually are a number of feasible alternatives. The designer is required to make decisions based on many parameters and to choose among possible alternatives, while evaluating the feasibility of each choice [5, 8]. This situation reflects the overall *uncertainty* and decision process that designers encounter in obtaining a solution. Every effective design decision resolves some part of an unclear situation and reduces the number of possible alternatives, but may also limit optimality. In the face of uncertainty, a designer is obliged to evolve a design so that if an artifact were to be produced according to that design, it would meet the requirements and satisfy the stated constraints [3]. However, the design may not be the optimal. For the sake of completeness, it is relevant to mention Herb Simon's notion of *bounded rationality* [100, 102] as an observation of human decision making. The *bounded rationality* notion is that rationality of individuals is bounded by the information available to them, their cognitive limitations, as well as time available to make their decision. Herb Simon coined the term "satisfice" by combining words *satisfactory* and *sufficient* [100]. As such, a decision-maker becomes a "satisficer," one seeking a satisfactory and sufficient solution rather than an optimal one.

A discussion of the psychology of problem solving may also help to evaluate the design process more clearly. Following Pahl [5], problem solving commences with factual knowledge about a given problem domain. In a design process, goals and constraints comprise the factual knowledge about the design problem (and there is normally uncertainty). This factual knowledge is conceptualized in the designer's mind. In this

process, short-term and long term-memory are distinguished. It is assumed that short-term memory retains a limited quantity of facts on a temporary basis [101]. On the other hand, long-term memory retains the selectively obtained knowledge units for long-term usage of the entity [5, 103].

Limitations of short-term memory and associative structure requirements in long-term memory naturally lead designers to search for organized structures within a given problem area [5, 104]. A structure consists of components and the relationships among them [5, 36]. Humans use decomposition techniques to find components and relationships [105]. Decomposition is the partition of anything into its elements and the study of their relationships [5]. It is generally understood that, humans are able to recognize the following types of relationships:

- concrete-abstract relationships,
- part-whole relationships, and
- space-time relationships.

4.2 Software Design Principles

Just as in “hard” engineering disciplines, in *Software Engineering* software design is considered a fundamental activity. However software development is in a “pre-engineering” phase analogous in many respects to the pre-engineering phases of long established engineering disciplines [26]. Software-engineering activities are the techniques used by humans to solve problems, while computers implement the solution. Software engineering consists of a series of steps [106]. It has been demonstrated repeatedly in software development that it is a challenging task to produce a high quality artifact within the cost and scheduled time parameters, especially for large complex projects. Systematic software design methodologies reduce the cost of software development and improve the quality of software products [3, 27, 107, 108]. Tools and effective methods for the activities

in software development have been developed and lead to a better understanding of the software development process [108, 109].

A deeper analysis of design reveals that designers evidently focus on abstractions such as data, function, and control abstractions in order to master the complexity in software systems [17]. Abstraction of data is the determination of elements, their values, properties and operations [110]. The result is the *decomposition* of items that constitutes the software system [44]. Data are abstracted so that access to the raw data is provided through a set of predefined operations [111, 112]. Abstraction of function is the functional decomposition of software systems [110, 112]. The domain, range, and transformation of functions are determined using function abstraction [110]. Abstraction of control is the determination of the sequence of data and function activities [110, 112].

These abstractions have been provided in the form of programming language constructs and design tools [108, 112]. Significant effort has been expended over several decades to find new design techniques, programming languages, and other strategies for the production of software [106, 108, 109]. In the early days, programs were implemented as a single block of instructions [110, 112]. Over time, as problems became more complex and computers became more powerful, the size of the programs have correspondingly increased [104, 106, 109, 113]. Controlling the large blocks of instructions proved to be difficult for developers [110, 112, 114, 115]. Naturally, to tackle the complexities, language designers started applying hierarchical decompositions techniques [22, 116, 117]. Large programs were organized into subprograms. Therefore, as a decomposition strategy, numerous approaches were introduced [116, 117, 118, 119]. These approaches, which can generally be grouped under the category of *module-based programming*, amounted to the development of constructs, such as function, subroutine, and modules [17, 110]. Further developments in hardware technologies and changing requirements led to the need for implementing even more sophisticated programs. Therefore, the shortcomings of module-based abstractions and decomposition eventually became pronounced [22, 120].

As such, the need for even more advanced decomposition techniques resulted in the development of yet newer programming languages, newer development paradigms, and programming constructs [121, 122]. *Object-Oriented techniques* [22, 115, 120, 123] and *Aspect-Oriented techniques* [124, 125, 126] are among those facilitating further abstraction and decomposition.

4.2.1 Design Specification Languages

The foregoing approaches have been successful in their own right to provide strong support for human designers but they were also supported by specification languages as well as other forms of linguistic approaches. Therefore; we briefly discuss specification languages although over the years specification languages gained complete independence from many abstract design notions.

Specification languages are utilized to represent software design. A software specification is a description of a collection of software components, defining their boundaries within software systems and capturing interconnection relationships among them [127]. Abstract definitions of these components are defined using specification languages during software development [107]. There are a variety of specification languages targeting different phases of software development, such as Z [128], Clear [129], CSP [130], and Larch [131]. These languages provide abstraction techniques to cope with the software system's complexity [111, 112, 127]. Specification languages are used to define structural and behavioral abstractions [27, 107, 127]. Structural abstractions provide the details for the composition of the software system [112, 127]. On the other hand, behavioral abstractions specify the execution time details such as the input-output relations of the software systems [107, 127].

Informal or natural languages enable one to specify software systems in a systematic way, lacking formal mathematical underpinnings [107, 121]. Formal languages use mathematical structures and provide reasoning about design [107, 127]. Formal languages

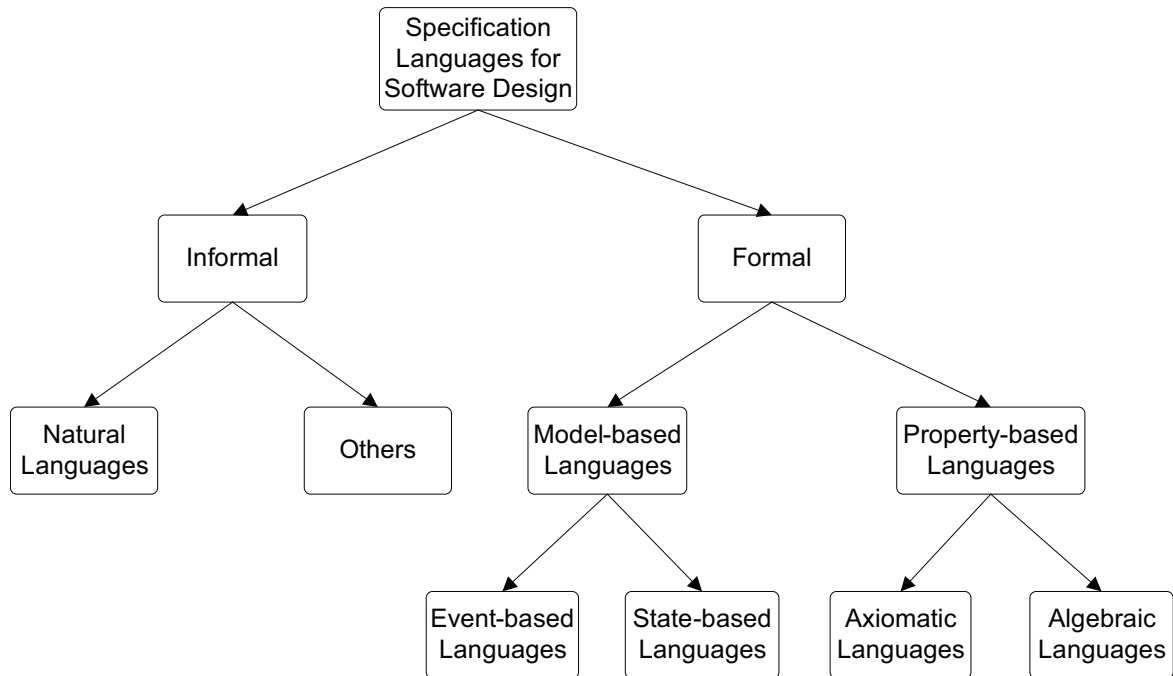


Figure 4.1: Classification of Specification Languages (Adapted from [107, 127])

are classified into two principal categories [107]. First, the *model-based languages* provides techniques to construct a model of the software system using structures such as sets, functions, and sequences. Model-based specification languages concentrate on describing how a software system is to operate. These languages are categorized into two subgroups. In an event-based languages, sequence events are used to represent system behavior [127]. While in state-based languages, software systems are represented by a sequence of states [127].

Second the *property-based languages* enable designers to specify the properties of software systems using logic and equations [127]. These languages are concerned with what the system is to do. They are based on procedural and data abstractions. Axiomatic languages, which are used for procedural abstraction, are classified within the property-based languages [127]. They use first order predicate logic. On the other hand, algebraic languages use equations to define data abstractions [127]. Figure 4.1 shows these languages.

4.2.2 Methods for Representing Semantics/Behavior

There are several different methods that have been proposed to describe the semantics of programming languages in a concise and formal manner [132]. These methods also have been successfully applied to define the semantics for Software. The following lists some of the main approaches to specify programming language semantics that may also be useful for defining the semantics of software.

Denotational Semantics: Denotational semantics is a semantic definition technique based on formal constructs [133]. In denotational semantics, each language element is associated to a mathematical object by mapping functions. Denotational semantics provides concise and rigorous definitions to represent the meaning of constructs [133]. Although denotational semantics of software systems could be defined in terms of state changes, manipulating mathematical objects rather than software constructs lead to difficulties and complexity during implementation.

Operational Semantics: Operational semantics can be used to specify the meaning of a programming language in terms of program execution on abstract machines [134]. Semantic definitions are comprised of rules, which describe specific effects of language constructs on an abstract machine. Each rule consists of preconditions that have to be met for the rule to apply and affects the transformation of the current state in some way. The final states of this transition system only contain values; they represent the result of the specification. Most software design platforms that provides a formal way of specifying the semantics have employed an operational style of the semantics definition.

Attribute Grammar: An attribute grammar [135] is another formal technique used to specify static semantics as an extension of a context-free grammar. Attribute grammars are mainly used to check the correctness of the static semantics such as variable type checking, and compatibility between procedure definitions and calls. Attribute grammars form one of the essential parts of compilers and offers benefits such as automatic construction of compilers, interpreters, and other language-based tools [136]. However, the large number

of rules required for a complete definition of a language may offer challenges when using attribute grammars to define a software systems.

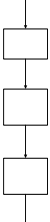
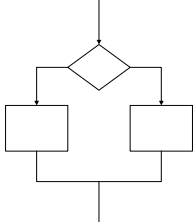
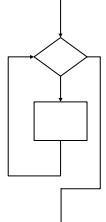
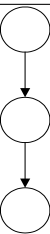
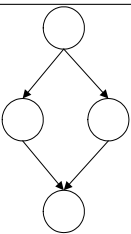
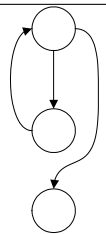

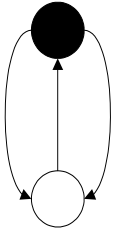
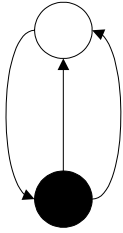
Graph Grammars : Graph Grammars is a formal technique used to divide all semantic concerns into discrete states and transition relations [137]. Graph grammars provide visual rules that specify in-place transformations based on precondition and postcondition steps. An in-place model transformation rule is defined as $L: [NAC]*LHS \rightarrow RHS$, where L is the rule label, LHS denotes the left-hand side rule stating the precondition pattern to trigger the rule; the RHS represents the right-hand side rule that specifies the final model part after execution of a rule. NAC is the optional negative condition that disables the rule if it is satisfied. Each graph transformation specifies the runtime behavior for one of the state transitions [137, 138, 139].

4.2.3 Control Abstractions

Linguistic approaches although successful in their own right they are not sufficient to further improve the art of software design. Significant benefits could accrue to designer if linguistic information can be quantized. If successful it enables computational modeling of design. Furthermore, it provides a possibility of developing a deductive reasoning mechanism to reach different design alternatives.

In order to support human intuition and provide documentation, behavioral aspects are typically represented by a diagrammatical system such as an *activity diagram* [140] and then transformed into a computer program [141]. Although *activity diagrams* document the logical flow of design behavior [140], they are not formal tools providing a mathematical foundation for analysis of design. Traditionally, compiler implementers have used graph-theoretical systems such as control flow graphs for analysis [142]. Converting programs or activity diagrams to control flow graphs can be achieved by introducing junction nodes where two lines meet without a vertex (node) [143]. The resulting directed graph is now a mathematical object and is called a *program control graph* or *control*

Table 4.1: The Three Constructs of Programming and Their Equivalent CCFGs (Adapted from [143])

	Sequence Construct	Selection Construct	Repetition Construct
Activity Diagram			
FlowGraph			
Cubic Graph			

flow graph [142]. Control flow graphs have been useful in analyzing programs and their algorithms. They not only represent the logic flow of the program but also the overall control structure. Table 4.1 shows the three basic constructs as activity diagrams and their equivalent flow graphs. Representing programs with graphs provides software developers with a graph theoretical toolbox to solve problems associated with programming systems [142, 144]. However, we need graph formalism that provides composition/decomposition principles in a rigorous way.

A specific class of control flowgraphs, called *Cubic Control Flow Graphs* (CCFG), was introduced to study decomposition properties of complex programs [45]. A CCFG is a strongly connected directed graph in which every vertex has a degree of three. Every CCFG has an even number of vertices that are colored half-black and half-white in order to represent decision and junction nodes respectively. Decision vertices have

indegree one and outdegree two, while junction vertices have indegree two and outdegree one. CCFGs, when used to represent the skeleton of a program, enable composition and decomposition of the structural system skeleton of associated programs [145]. A CCFG can be reduced to its components, namely the CCFGs of the three constructs [45]. Then, composition/decomposition principles derived from the cubic graph formalism are used to investigate the integration of the system [45, 142].

The decomposition process divides a cubic graph into at least two cubic graphs. The key idea of the decomposition is to remove some edges from a given cubic graph and to add some edges to decomposed components so the original graph is disconnected while the resulting components remain cubic graphs [45]. The composition of two cubic graphs is the process of combining two arbitrary cubic graphs into one cubic graph. This procedure is explained in detail in [45] and used in [146].

The decomposition of cubic graphs classifies them into two different classes: *decomposable* and *non-decomposable* cubic graphs. A cubic graph is said to be a *prime cubic graph* if it is non-decomposable; otherwise, it is said to be a *non-prime cubic graph* [45, 147]. Generation of prime cubic graphs and enumeration of them are discussed in [147].

4.2.4 Coupling and Cohesion

The way in which a software system is decomposed significantly affects the complexity of software [17, 44, 106]. There is a need to specify decomposition criteria that can guide the designer in design activities. There are two basic criteria for assessing the decomposition of a system: *cohesion* and *coupling* [116, 118, 148]. These criteria have been used for evaluating software design in Structured design and Object-oriented design [22, 27, 109, 116, 119].

Coupling is a measure of the strength of interconnection among the decomposed elements [116, 143]. Coupling is one way to evaluate the decomposition of a system. It

defines the degree of independence among the decomposed elements [149]. Highly coupled elements are joined by strong interconnections, while low coupling elements are joined by relatively weak interconnections [115, 150]. Degree of coupling is an important factor in systems complexity [143]. Making decomposed elements as independent as possible is one of the major objectives when dealing with system complexity [22, 27, 106, 110, 116]. Interconnections determine how well the system can be maintained or changed. It is hard to make changes to one decomposed element without affecting the others for highly coupled systems [17, 22]. Because of strong interconnection, an update within a decomposed element leads to additional updates in the other units; this is undesirable.

Cohesion is a measure of the strength of associations of elements within a single decomposed element [119, 143]. Cohesion is another way to measure how well a system is decomposed into elements. Within highly cohesive units, elements are strongly related to one another. Placing strongly associated elements into same unit is another objective while dealing with complexity [27, 104, 110].

Coupling and cohesion are clearly interrelated [116]. The cohesion of decomposed elements often determines the coupling among the elements. The greater the cohesion of individual elements lead to the lower coupling among the elements [118, 149].

4.3 Contemporary Design Decomposition Approaches

We will review contemporary design decomposition techniques starting with Objects followed by a discussion of Design Patterns, Aspects, and Domain-Specific Languages.

4.3.1 Objects

Object-oriented techniques decompose the software system into a set of objects with well-defined interfaces [22, 115, 120, 123]. An object encapsulates data and services for manipulating data [109, 112, 115]. Objects call on the services provided by other objects.

Information hiding defines how objects interact. Inheritance and polymorphism provide extension mechanism for objects [112, 115, 121].

Intellectual distance was defined by Edsger Dijkstra to show the distance between the problem space decomposition and the design space decompositions [151]. Objects minimize this distance by defining design structure as close as possible to real-world structure. So the structure of the system is readily understandable.

4.3.2 Design Patterns

Christopher Alexander proposed the pattern and pattern language concepts to describe the solutions to the problems that occur over and over again [152]. His ideas were adopted by the object-oriented community and applied into software problems. Software designers developed over time design patterns to solve recurring software design problems [42].

Design patterns specify design abstractions in a simple and elegant way. Each pattern provides a specific decomposition technique for a specific case. Designers utilize patterns to choose design alternatives that make software systems highly reusable and easily maintainable. The collection of 23 design patterns are given in the so-called *Gang-of-Four* catalog [42]. The patterns are divided into three categories: *creational*, *structural*, and *behavioral*. Creational patterns provide solutions to problems related with object creation [42]. Structural patterns deal with composition of the classes [42]. Furthermore, behavior patterns defines techniques for the object interactions and their run-time responsibilities [42]. Within this catalog, each pattern has four essential elements: the pattern name defines the pattern in a word or two, the problem describes the cases to apply the pattern, the solution describes the template, which is a general decomposition technique, to solve the problem, and finally the consequences put results and trade-offs of applying the pattern [42].

4.3.3 Aspects

Although objects enable designers to decompose the system as close as possible to real world, there are elements within the system require further decomposition methods [124]. These elements, called cross-cutting concerns, scatter throughout the objects and affect all the objects in the systems [124, 125, 153]. Aspect-Oriented Software development (AOSD) provides decomposition techniques [125]. In aspect-oriented programming terminology, *Aspects* specify cross-cutting concerns, *Joint points* defines where aspect should be associated in source code, and *Aspect weaving* links aspects to the joint points [124, 153].

4.3.4 Domain-Specific Modeling Languages

Model-Driven Engineering (MDE) is a software methodology that utilize modeling techniques to raise the level of abstraction from the solution domain to problem domain [154]. It has been shown to increase productivity and reduce development costs [154]. The concepts advocated by MDE focus on abstractions tied to a specific domain that provide tailored modeling languages for domain experts. Domain-Specific Modeling Languages (DSMLs) [155], used within the MDE context, enable end-users who are domain experts to participate in software development tasks and to specify their own programs using domain concepts in the problem space, rather than programming language concepts in the technical solution space [156, 157].

DSMLs, like other programming languages, consist of definitions that specify the abstract syntax, concrete syntax, static semantics and behavioral semantics of a language [158]. Specification of abstract syntax includes the concepts that are represented in the language and the relationships between those concepts. In MDE, domain metamodels are often used to define the structural rules for the abstract syntax [157]. Concrete syntax definition provides a mapping between meta-elements and their textual or graphical representations. Well-formedness rules represent the static semantics of a language. Such rules are often specified in constraint languages (*e.g.*, OCL) that enforce rules among

metamodel elements. The runtime behavior of each syntactical meta-element defined in the DSML represents the behavioral semantics of the language.

4.4 Design Space Analysis

As we have mentioned, software designers have to consider multiple alternatives during design and reach a decision based on experience and the methodology that they employ. They may eliminate some design alternatives in early stages, although that may result in loss of information, and hence optimality [159]. Therefore, designers need a consistent technique to represent, compare, and select among design alternatives. Aksit and Tekinerdogan [44] provided an approach to rank various design alternatives based on quality factors, adaptability, and time. They provided algebraic techniques, called *Design Algebra*, to form a mathematical foundation for the design process [44]. Design Algebra can be viewed as a special form of *Relational Algebra* [160] that implements operation instructions such as *size*, *reduce*, *quantify*, and *generate* for design spaces [44]. Design space is a function that maps fundamental concepts to the design properties [44]. Fundamental concepts of the domain are identified with domain analysis. Design properties include quality factors and implementation details that cover functional and non-functional requirements [44]. Each design alternative in a design space can be considered as one specific mapping of concepts to design properties. Aksit and Tekinerdogan [44] specify design algebra operations are as follows:

- Size, defines the total number of alternatives that can be generated from design space.
- Generate, identifies all possible design alternatives in a design space.
- Reduce, enables a condition to reduce the size of the design space. (Reduction is the mechanism that provides elimination of alternatives based on required criteria. Unfeasible and uninterested alternatives are discarded by the reduce operation.)

- Quantify, measures the design alternatives to provide a means with which to differentiate criteria. (The most relevant design alternative has the highest value after the quantification process.)

4.5 Summary

In this chapter, we reviewed general design principles from the perspective of design decomposition and related concepts. We followed with engineering design as well as software design principles and approaches. We included the role of specification languages as well as other forms of linguistic approaches. We also reviewed structured design decomposition including notions of data, function, and control based abstractions and their role in decomposition including coupling and cohesion. Control flow graph, a formal version of flowchart, is a popular decomposition technique favored by compiler designers as well as static software analysis programs. A specific class of control flow graphs is Cubic Control Flow Graph (CCFG). We reviewed CCFG because it provides the most complete control flow formalism. Methods for representing semantics are also reviewed. Our review continued with domain specific modeling languages, design patterns, objects, and finally aspects.

We concluded with design space analysis, in which software designers have to consider multiple alternatives and reach a decision based on experience and the methodology that they plan to use. In other words the designer develops a parameter of interest for his design decomposition and performs the decomposition accordingly. The actual mechanism of achieving this goal and its information theoretical analysis will be explored in the next chapter.

The historical objective in developing these successive approaches and their associated methodologies has been that the human developer should maintain intellectual control over the developed design by hierarchical series of abstractions. Although these approaches has been successful in their own right in providing strong support for human designers, they

are fundamentally linguistic in nature and do not consider information flow in a manner in which we can make computational modeling of design. Furthermore, they do not present a possibility of developing a deductive reasoning mechanism to reason about different design alternatives.

By reviewing all prior techniques from the perspective of design decomposition, we wanted to expose these approaches in such a way that we can analyze all types of designs primarily by information theoretical means in the following chapters.

Chapter 5

INFORMATION THEORETICAL ANALYSIS OF SOFTWARE SYSTEMS

Our general thesis is that software designers can benefit from the experience of engineering designers and information theory formalism. Knowledge of information theory enables software designers to adopt a systems view which facilitates intellectual control over a given software design. Since design imposes organization through successive transformations in reaching the final product, it is possible to formalize it with information theory. This is where reduction of entropy concept and the associated mathematics, various decomposition techniques become useful. The following sections in this chapter will illustrate how communication channel formalism from information theory can help formalize these relationships.

Software designers can also improve organization and clarify their thinking by systematic design space analysis proposed by Aksit and Tekinerdogan [44]. Every design starts with uncertainty and the art of software design reduces uncertainty. We discern from Aksit and Tekinerdogan approach and further systematize the approach through information theory.

The concept of a system and information theoretical approach has been introduced in Chapters 1 through 4 from the perspective of design decomposition. After the presentation of the overall perspective in Chapter 1, we introduced formal study of information in Chapter 2, while reviewing the systems perspective in Chapter 3. Finally, we connected

to our main objective, software design, in Chapter 4. In order to maintain the historical perspective, we retained the terminology of the original sources in these chapters. For example, we maintained the term “element” for software modules following Ashby [34] and Watanabe [33].

In this chapter we introduce our initial assumptions in order to present a consistent approach for information theoretical analysis of software systems. After the introduction of our approach, we provide supporting examples and case studies in the next chapter. The key operational assumptions are as follows:

- Design is a hierarchical decomposition and covers all steps from requirements to the final product (Design is fundamental).
- Design imposes an organization through successive transformations, and therefore
- Formally, design reduces entropy.

Furthermore, we observe that these assumptions of design are shared by software design as well. In fact, following Simon we can state that software “design is the transformation of existing conditions into preferred ones.” This formulation is itself an early contribution of this work because this view of design is novel as far as we can ascertain from the literature.

It should be remembered that, historically, software design has not been studied as an entropy reduction process leading to the final product. Of course, many particular approaches have existed to decompose software design to gain some level of intellectual control of the overall process. It was even recognized that design is a central event in developing software and as such various methodologies were developed [17, 18, 107]. In some cases software has even been called “abstract design” [9, 10, 17] and design space techniques are proposed to aid the software development [44]. In fact, many levels of design have been recognized starting from overall design or architecture to detailed design [107]. However, a need to develop a new approach for consistent and systematic development of software design still exists.

Following Simon, Shannon, and Conant we investigate software design as a hierarchical decomposition and we represent software design as a communication channel, which is a novel strategy. As such, software design is represented as an entropy reduction process. Since this approach rests on a known mathematical formalism, information theory, the machinery of the theory can be used for further systematic study of software design.

5.1 Software Design is an Entropy-reduction Activity

Software designers generate various alternative approaches to decompose the given problem from requirements, finally yielding to final artifact. Usually several decomposition alternatives exist and the designer must make decisions based on many parameters and to make choices among possible decomposition alternatives, while evaluating the feasibility of each choice. The choice among them often is subtle. This situation reflects the *uncertainty* that designers encounter in finding a specific decomposition. Uncertainty exists in every step of software design, such as the clarification of requirements, mapping problem space concepts into solution space concepts, and transformation of solution space concepts into executable concepts.

It is our view that, every design decision resolves some part of an unclear situation and reduces the number of possible alternatives. Thus, each design activity is an *uncertainty-reduction* process. The following section demonstrates the entropy reduction activities during problem space and solution space mapping. Four different software design specifications are used to define software design space. Each design space captures all possible decomposition alternatives related with one particular aspect of software system. Design alternatives within spaces clearly show the uncertainty that designers encounter while progressing through steps.

Table 5.1: Mapping Between Problem Space Concepts and Solution Space Concepts

Problem Space Concepts	Structural	<i>Class</i>
		<i>Attribute</i>
		<i>Operation</i>
		<i>Relationships</i>
	Behavioral	<i>Sequence</i>
		<i>Branch</i>
		<i>Loop</i>

One responsibility of a software designer is to transform problem space concepts into solution space concepts. Problem space concepts are terms, definitions, and rules from business/customer domain which are independent of technical details. On the other hand, solution space concepts are technical terms that incorporate solution details. Solution space concepts for software systems can be categorized into two groups, *Structural and Behavioral Concepts* [44]. Table 5.1 shows the mapping between problem concepts and solution ones. In this section, *Class*, *Attribute*, *Operation*, *Relationships*, *Sequence*, *Branch*, and *Loop* concepts are used to decompose solution spaces.

As stated above, software design consists of structural and behavior specifications. All possible design alternatives for these specifications form a design space for the software. To identify software abstractions, and corresponding decomposition activities, four different design spaces are defined, a discernment we obtained in this research. These are as follows:

- Structural Entity Space,
- Structural Relation Space,
- Behavioral Flow Space, and
- Behavioral Expression Space.

5.1.1 Structural Spaces

Design space decomposition starts with the specification of structural spaces. *Structural spaces* define entities, attributes, and relationships between the concepts [44]. While entities within solution space are specified in a Structural Entity Space, relationships are given in a Structural Relation Space.

5.1.1.1 Structural Entity Space

Mappings between structural problem domain concepts (C_{Domain}) into structural solution concepts are shown in this space. Figure 5.1 demonstrates the space as a two dimensional space. Following Aksit and Tekinerdogan [44], definitions of Structural Entity Space and corresponding solution space concepts are given as follows :

- The predefined property P_{Entity} (represented as the y-axis in Figure 5.1) is a set of solution space alternatives for problem space concepts. $P_{Entity} = \{Class, Operation, Attribute\}$, and
- $S_{StructuralEntity}$ defines the design space that maps the concepts of C_{Domain} to the elements of P_{Entity} and as such represents the total set of alternatives of domain models.

5.1.1.2 Structural Relation Space

This space shows the relations between problem domain concepts in relational terms. Figure 5.2 demonstrates the two-dimensional space. Following Aksit and Tekinerdogan [44], definitions of Structural Relation Space and corresponding solution space concepts are given as follows :

- The predefined property $P_{Relation}$ (represented as the y-axis in Figure 5.2) is a set of alternatives for the relationships between concepts. $P_{Relation} = \{Association, Generalization, AttributeOf, MethodOf, NoRelation\}$, and

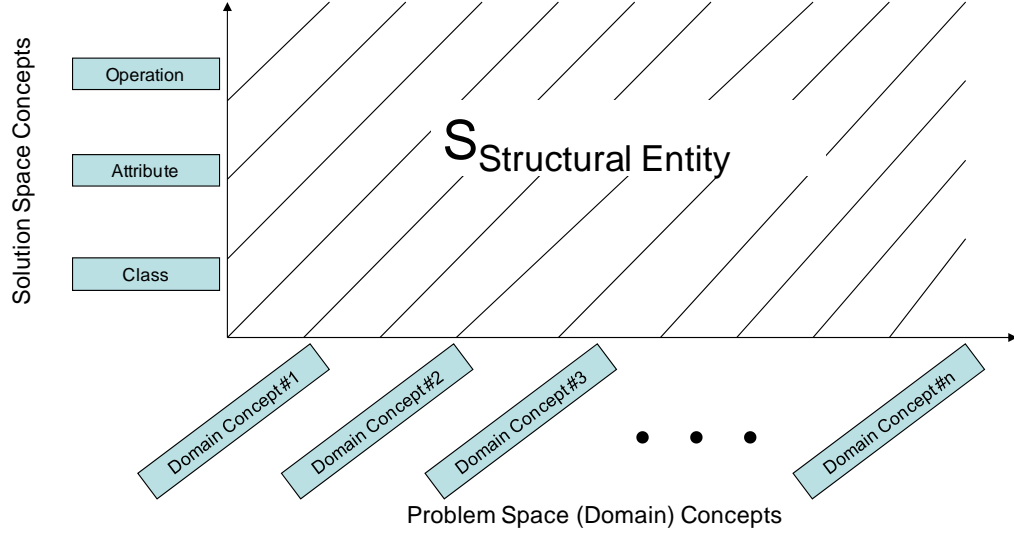


Figure 5.1: Structural Entity Space

- $S_{StructuralRelation}$ defines a design space that maps the 2-tuple concepts of C_{Domain} to the elements of $P_{Relation}$ and as such represents the total set of relationship alternatives of domain models.

5.1.2 Behavioral Spaces

As stated in previous sections, structural aspects of software are given with two structural design spaces [44]. However, design space specifications should also provide behavioral spaces. Representation of design alternatives for software behavior is the early contribution of this work. To represent behavioral design alternatives, behavior is decomposed into two spaces. The first space gives the flow alternatives of each design. The second space represents operation steps of behavior as expressions.

5.1.2.1 Behavioral Flow Space

Software programs have an equivalent representation composed of repetitive, selective, and sequential constructs [161]. In this study, behavior flow space enables behavioral mapping for the design. *Cubic Control Flow Graph (CCFG)* is the solution space concept for the

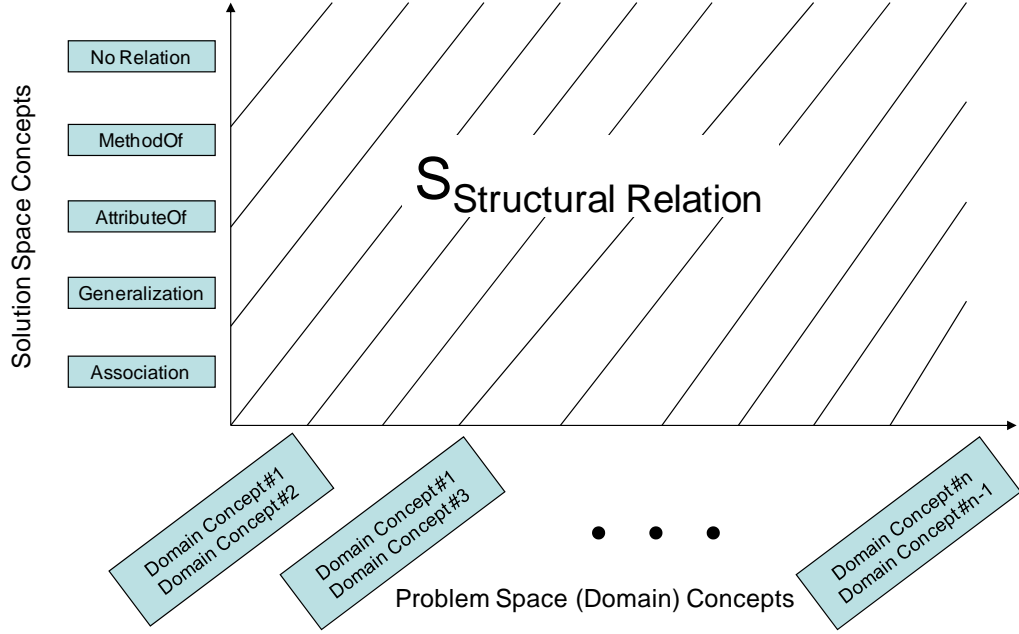


Figure 5.2: Structural Relation Space

decomposition of behavior flow space. This space shows the logical flow of operational domain concepts with CCFG [45, 145, 147]. CCFG is a specific class of control flow graphs providing mathematical convenience for design space decomposition and analysis. Each cubic control flow graph represents a flow, which is composed of if and/or loops, as a combination of decision nodes and junction nodes. Figure 5.3 demonstrates the two-dimensional space. Definitions of Behavior Flow Space and corresponding solution space concepts are given as follows:

- The predefined property P_{Flow} (represented as the y-axis in Figure 5.3) is a set of prime cubic graph alternatives for the flow of operational domain concepts.
- $S_{BehaviorFlow}$ defines design space that maps the operational concepts of C_{Domain} to the elements of P_{Flow} and as such represents the total set of flow alternatives of domain concepts.

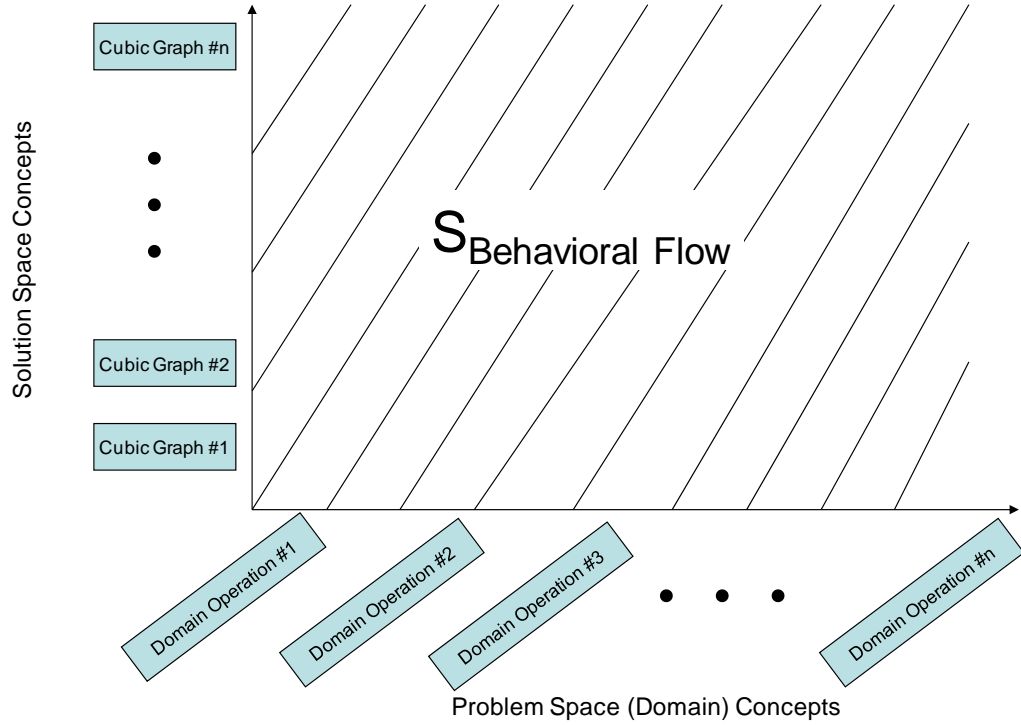


Figure 5.3: Behavioral Flow Space

5.1.2.2 Behavioral Expression Space

Behavior Flow Space represents the decomposition of logical flow. Additionally, to represent operational steps, cubic graph edge specifications as well as domain concept mappings are required. Control flowgraphs with proper labeling comprise the formal tool to complete behavioral specification. Graph grammar rules are used to decompose operational steps. *Expression Space* shows specification of cubic graph labels. Figure 5.4 demonstrates the two-dimensional behavioral flow space. Definitions of Behavior Expression Space and corresponding solution space concepts are given as follows:

- The predefined property $P_{Expression}$ (represented as y-axis in Figure 5.4) is a set of graph grammar rules for the state transition steps of domain operations.
- $S_{BehaviorExpression}$ defines design space that maps the labels defined on cubic graph edges to the elements of $P_{Expression}$ and as such represents the total set of state transition alternatives of domain operations.

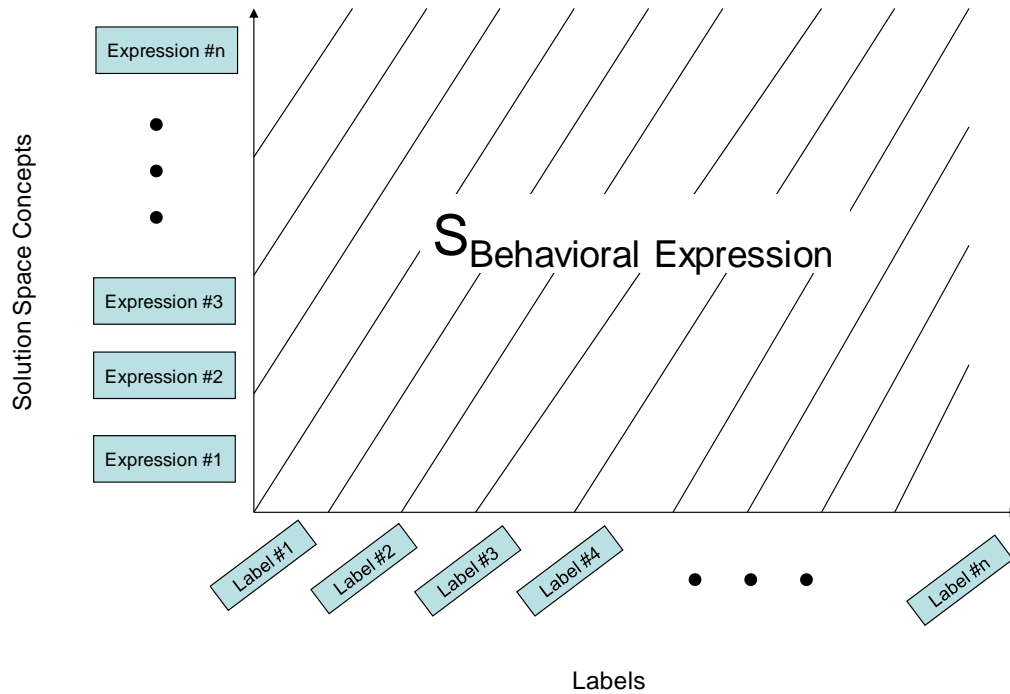


Figure 5.4: Behavioral Expression Space

Designers decompose these design spaces defined above using solution concepts to generate a design artifact. In the beginning there is minimal organization therefore high uncertainty exists (high entropy). The design decisions carrying out design activities reduce uncertainty and introduce comparatively higher organization (low entropy). In a pictorial form given in Figure 5.5, the design process is represented from the perspective of the designer to capture the uncertainty reduction process. In Chapter 6 a representative example of this process is presented in the form of a library example. All of the possibilities within the four spaces, given on the left part of Figure 5.5, demonstrate the uncertainty. On the other hand, the artifact, given on the right part of the figure, displays organization (low entropy). Representing software design as an uncertainty reduction process is one of the novel contributions of this work.

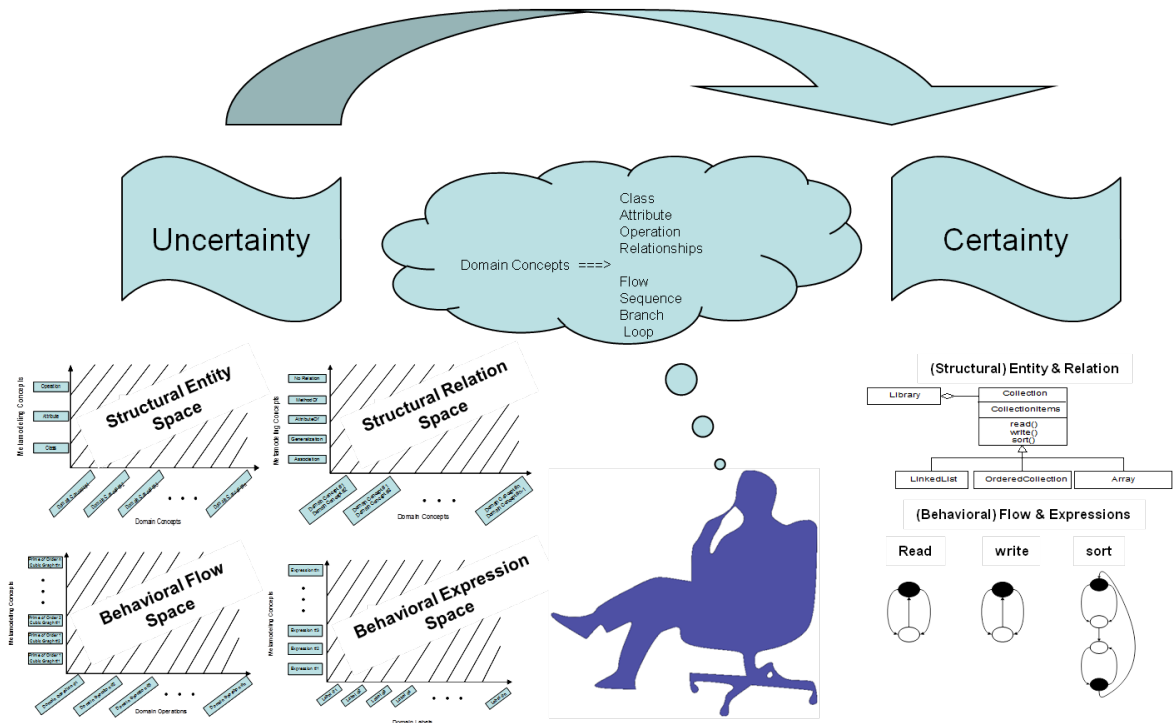


Figure 5.5: Software Design Activities Transforms Uncertainty to Certainty

5.2 Information Theoretical Approach

System decomposition provides a mechanism to deal with transforming uncertainty to certainty during design. The process of transforming uncertainty to certainty is viewed as a process towards organization. In turn, design as a fundamental process is a hierarchical decomposition and covers all steps from requirements to the final artifact(s). Therefore, design imposes an organization through successive transformations, and formally speaking, reduces entropy. Furthermore, software design possesses these key assumptions as well.

In fact, software design is an entropy reduction process leading to the final artifact. Therefore, following Simon, we analyzed software design as a hierarchical decomposition and following Shannon and Conant, we represented software design as a communication channel which is the interaction between elements. Since information theory is a well-known engineering formalism, it can usefully be applied to advance our viewpoint.

We developed the communication channel representation of software through a process of 1) Set-theoretical representation, 2) Mapping to communication channel formalism, and 3) through hierarchical decomposition leading to entropy reduction. This representation enables us to study software systems as communication channels, and is a novel contribution of this work.

5.2.1 Set-theoretical Representation of Software Systems

We start with mapping of software systems to set-theoretical representations. Software systems are represented with an arbitrary number of variables. We define a set of K variables for a given software system. The variables represent elements, such as identifiers defined within programs, data values from data segment, function return values, and code segment addresses. Each variable is denoted by X_j where $1 \leq j \leq K$. Software system is a set of X_j , denoted by the set $S = \{X_1, \dots, X_K\}$.

X_j 's values are taken from the set $P_j = \{X_j^1, X_j^2, \dots, X_j^{n_j}\}$. P_j is a finite set, and its elements depends on the software elements which X_j represents. The set P_j forms a partition associated with variable X_j .

For example, when an *integer* type identifier defined within a given program is associated with X_1 , P_1 takes a set of valid values associated with this integer type. Considering another level of abstraction, we can see that for an integer type with n bits, *unsigned type* represents the non-negative values 0 through $2^n - 1$, so that $P_1 = \{0, \dots, 2^n - 1\}$, on the other hand, *signed integer type* represents numbers from $-2^{(n-1)}$ through $2^{(n-1)} - 1$, therefore the set P becomes $P_1 = \{-2^{(n-1)}, \dots, 0, \dots, 2^{(n-1)} - 1\}$.

The next step is the observation of values associated with variables of the system. Therefore, values associated with each variable are obtained and observed once per cycle, for example, at the end of a user event for an application. The cycle represents the stable states within a software system. The cycle should allow the variables a chance of changing values so the stable states of software system can be observed. In terms of the

communication channel, this maps input variables of a channel to output variables. We observe the K variables for N cycles, and obtain a total of $K \cdot N$ different values. Therefore, observed values for each variable is denoted by $O_j = \{O_j^1, \dots, O_j^N\}$. Observed number of occurrences of the event in consideration $X_j = X_j^i$ is denoted by $n_{X_j^i}$, such that $\sum_{i=1}^{n_j} n_{X_j^i} = N$. Number of occurrences associated with partition P_j is denoted by $F_j = [n_{X_j^1}, \dots, n_{X_j^{n_j}}]$. The variables X_j is grouped into sets to demonstrate decomposition steps during software design. The set $S_i = \{S_i^1, \dots, S_i^{n_i}\}$ represents a subsystem of given software system, where $\cup_{i=1}^r S_i = S$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$.

A representative example of variables and observations is shown in Figure 5.6. In this example, eight variables are defined for a given software system. These eight variables are observed within seven cycles as listed in Figure 5.6. Therefore, in this example, $K = 8$, $N = 7$, $S = \{V1, V2, V3, V4, V5, V6, V7, V8\}$. The values of variables are shown in Table 5.2.

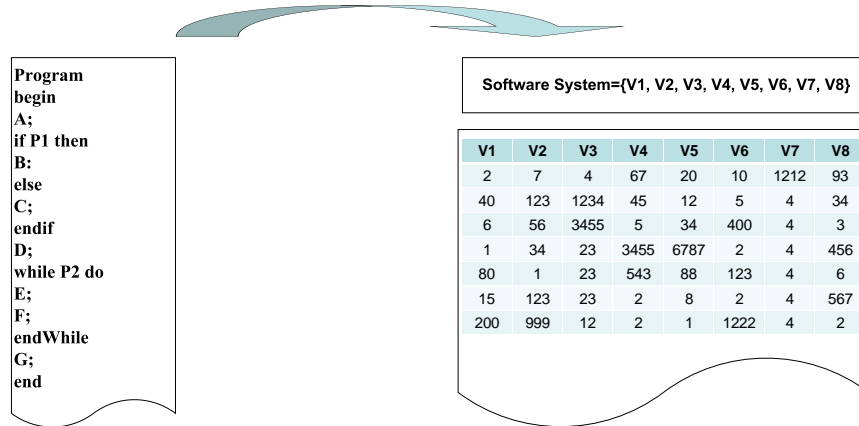


Figure 5.6: Conceptual Representation of Set-theoretical Representation of Software Systems

Table 5.2: Values of $V1 - V8$

Variables	P_i	n_i	F_i
V1	$P_1 = \{1, 2, 6, 15, 40, 80, 200\}$	$n_1 = 7$	$F_1 = [1, 1, 1, 1, 1, 1, 1]$
V2	$P_2 = \{1, 7, 34, 56, 123, 999\}$	$n_2 = 6$	$F_2 = [1, 1, 1, 1, 2, 1]$
V3	$P_3 = \{4, 12, 23, 1234, 3455\}$	$n_3 = 5$	$F_3 = [1, 1, 1, 3, 1]$
V4	$P_4 = \{2, 5, 45, 67, 543, 3455\}$	$n_4 = 6$	$F_4 = [2, 1, 1, 1, 1, 1]$
V5	$P_5 = \{1, 8, 12, 20, 34, 88, 6787\}$	$n_5 = 7$	$F_5 = [1, 1, 1, 1, 1, 1, 1]$
V6	$P_6 = \{2, 5, 10, 123, 400, 1222\}$	$n_6 = 6$	$F_6 = [2, 1, 1, 1, 1, 1]$
V7	$P_7 = \{4, 1212\}$	$n_7 = 2$	$F_7 = [6, 1]$
V8	$P_8 = \{2, 3, 6, 34, 93, 456, 567\}$	$n_8 = 7$	$F_8 = [1, 1, 1, 1, 1, 1, 1]$

5.2.2 Channel Formalism of Software Systems

As presented in the above section, set-theoretical representation of a software system and the corresponding observed values reveal that there are varieties in observed values, and there are relationships between the values. To analyze these relationships, we use communication channel formalism. The set theoretical decomposition of the software system lead us to expose the relationships between variables. This observation naturally lead us to capture this relationship in the formalism of communication channel. As can be discerned from the information theory literature the communication channel, as a mathematical object, connects input variables to output variables in a probabilistic manner.

A communication channel is represented by an input set $S = \{S_1, \dots, S_n\}$, an output set $R = \{R_1, \dots, R_m\}$, and a set of conditional probabilities $P(S_k | R_l)$ for all k, l [61]. A schematic of a communication channel is shown in Figure 5.7. Here S_1, S_2 are the input to the channel and R_1, R_2 are the output.



Figure 5.7: A Communication Channel

The interaction between elements in a given software system is represented using communication channel formalism. To represent the interaction between two system variables for example, X_i and X_j :

- the value set , P_i , which is associated with X_i , is taken as a channel input set S ,
- the value set , P_j , which is associated with X_j , is taken as a channel output set R , and
- then channel probability is, $P(S_k, R_l) = \frac{n_{S_k R_l}}{n_{R_l}}$, where observed number of occurrences of the event in consideration $\{R_l = P_j^l\}$ is denoted by n_{R_l} , the number of occurrences of the event $\{S_k R_l = P_i^k P_j^l\}$ is denoted by $n_{S_k R_l}$.

For example, Figure 5.8 shows the communication between $V3$ and $V4$ for the software system given in Figure 5.6.

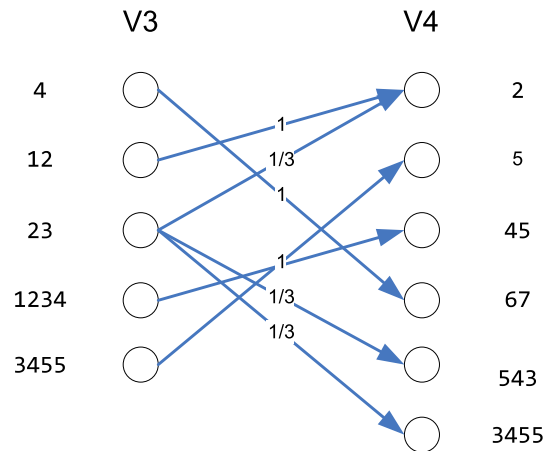


Figure 5.8: Communication Between $V3$ and $V4$

Communication channel representation is the mathematical modeling of the interactions among the software elements. This modeling enables us to demonstrate the organization and the entropy-reduction process during the software design.

To calculate the interaction in the channel we need the entropy calculation since entropy of a variable, X_j , denoted $H(X_j)$, is used as a measure of the variability of X_j . As stated in the second chapter, this value is calculated for each variable of a given software system by the formula [37] :

$$H(X_j) = - \sum_{i=1}^{n_j} \frac{n_{X_j^i}}{N} \log \frac{n_{X_j^i}}{N}. \quad (5.2.1)$$

The observed transmission between two variables, X_i , and X_j , is defined as follows:

$$T(X_i : X_j) = H(X_i) + H(X_j) - H(X_i X_j). \quad (5.2.2)$$

This value is equivalent to the *transmission formula* given in Chapter 2. We therefore use this formula for each interaction to measure the organization within software design, thus leading to the observation for entropy reduction.

In summary, the mapping of set-theoretical representation to channel formalism is needed to show the interaction between variables. As a result of this observation, we now have obtained a novel communication channel representation of the software system in Figure 5.9 produced from Figure 5.6. In Chapter 6, we present representative examples.

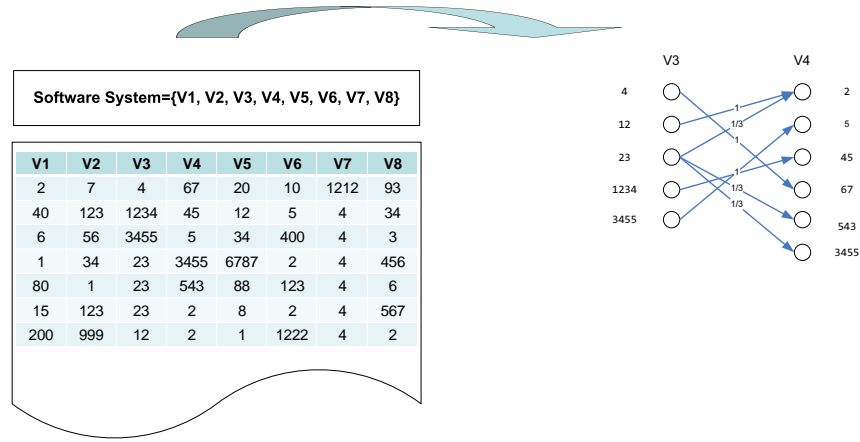


Figure 5.9: Conceptual Representation of Channel Representation of Software Systems

5.2.3 Hierarchical Decomposition of Software Systems

Hierarchical decomposition, through a process partitioning interactions in a channel, produces subsets of channel elements. In Figure 5.10, this process is represented as a transformation between channels and set-subsets producing a hierarchical combinations of software elements. A complete hierarchical decomposition example is given in Chapter 6.

Following the notation introduced above, software design decomposes the given software system $S = \{X_1, \dots, X_K\}$ into r elements, such that the variables X_j is grouped into sets, $S_i = \{S_i^1, \dots, S_i^{n_i}\}$ which represents an element of a given software system, where $\cup_{i=1}^r S_i = S$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$.

The total interaction is decomposed into transmission such that

$$C_{Total}(X_1 X_2 \dots X_K) = \sum_{i=1}^r C_{Total}(S_i) + C(S_1, S_2, \dots, S_r) \quad (5.2.3)$$

where $C_{Total}(S_i)$ is the transmission within an element, S_i , and $C(S_1, S_2, \dots, S_r)$, *correlation formula 2.2.14* given in chapter 2, is the transmission among elements. As a result, software system is decomposed into r elements with the total amount of transmission $C(S_1 S_2 \dots S_r)$. The example in Chapter 6 shows the actual decomposition using a specific example.

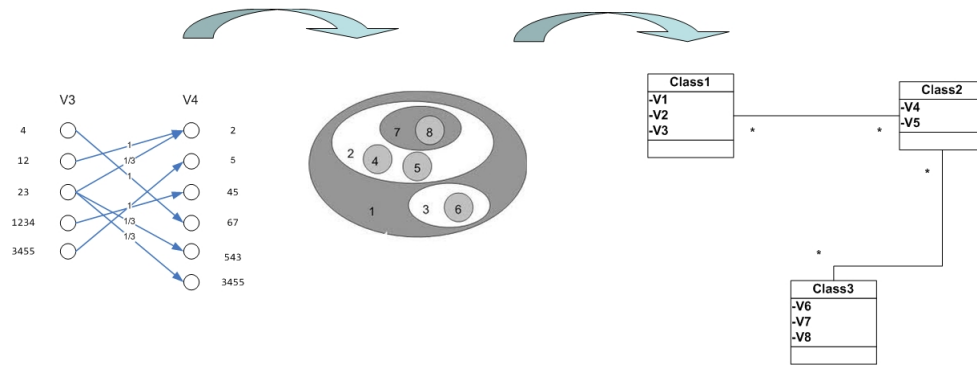


Figure 5.10: Conceptual Representation of Hierarchical Decomposition of Software Systems

5.3 Summary

In this chapter, we developed a novel communication channel formalism for software systems. We used a set-theoretical representation to produce a mapping from software to a communication channel, leading to entropy reduction. This representation enabled us to study software systems as communication channels. We formally demonstrated that

- software design is a hierarchical decomposition and covers all steps from requirements to the final product, and
- software design imposes an organization and reduces entropy through successive transformations.

The communication channel representation of software systems opens up useful possibilities for applying engineering mathematical analysis to software development. This means that current understanding and informal representations of software design, using our results, can now evolve into the type of inquiry involving classical engineering mathematics. These results offer an opportunity for transformative impact that could influence the area of software engineering by providing a relatively more stable context for discussing the representation of software design than is currently available. In summary, our contributions and their impacts are as follows:

- Formal representation of hierarchical decomposition of software and entropy-reduction view of software design, including :
 - better understanding of the design process in software engineering based on classical engineering principles,
 - establishing a new formal foundation for future software research, and
 - enabling the use of a known mathematical formalism of information theory, and therefore opening up possibilities to investigate software design with established engineering techniques and mathematics.
- Communication channel representation of software:
 - opening a new possibilities of software research, connecting software with information and coding theory, and
 - providing stronger bridge between established engineering methods and software design.

Together these theoretical contributions advance the state of software design theory as hypothesized in Chapter 1.

Chapter 6

EXAMPLES, RESULTS, AND ANALYSIS

In Chapter 5, we introduced an information-theoretical analysis of software systems based on the communication-channel formalism. In this chapter, we provide supporting examples associated with the theory introduced in Chapter 5. We begin with a representative example of design space analysis in which we demonstrate the organization process [44]. We follow up with an information-theoretical example in which we demonstrate the progress of discovery from set-theoretical representation to communication-channel representation, and finally to hierarchical representation. All these representations are necessary to map software systems to the communication-channel formalism.

For convenience, we recapitulate key assumptions as introduced before. We hypothesize that design, as a hierarchical decomposition, imposes organization through successive transformations, while covering all aspects from requirements leading to the final artifact(s). It should be noted that the design-space-decomposition example demonstrates the principle of imposing order throughout design. The other key idea is that design reduces entropy. The communication-channel example demonstrates the nature of entropy reduction in design, demonstrating a promised result of this dissertation. Therefore, with representative examples, we expose software design as an entropy-reduction process. Furthermore, we show the connection between the communication-channel formalism and software systems. We conclude with an analysis of the results.

6.1 Design Space Example

As explained in Section 5.1, making successive design decisions by decomposing the design spaces, produces higher levels of organization and leads to lower levels of entropy. This section demonstrates the application of the design-space decomposition to a library example provided by Aksit and Tekinerdogan [44]. The example is the design of a set of collection classes, such as *LinkedList*, *OrderedCollection*, and *Array* to be a part of an object-oriented library. These classes should provide the needed operations to read and write the elements stored in collection objects. Furthermore, the sorting operation is needed to sort items within collection objects.

An overall view of the application of the design-space decomposition to a library example and corresponding final artifact is provided in Figure 6.1. As stated in Section 5.1, software design consists of structural and behavior specifications. For the identification of the software abstractions, and the corresponding decomposition activities for the library example, four design spaces, *Structural Entity Space*, *Structural Relation Space*, *Behavioral Flow Space*, and *Behavioral Expression Space*, are demonstrated in this section.

The structural model of the library example is composed of the concepts of the domain (C_{Domain}) and the relationships in the domain (R_{Domain}). They are listed below:

- $C_{Library} = \{Library, Collection, LinkedList, Array, OrderedCollection, collectionItems, sort, read, write\},$
- $R_{Library} = \{(Library, Collection), (Library, LinkedList), (Library, OrderedCollection), (Library, Array), (Collection, LinkedList), (Collection, OrderedCollection), (Collection, Array), (sort, Collection), (sort, LinkedList), (sort, OrderedCollection), (sort, Array)\}.$

The structural decompositions are depicted in Figure 6.2 and Figure 6.3. The two types of structural decompositions, entity design space and relation design space, are shown in Figure 6.2 and Figure 6.3, respectively.

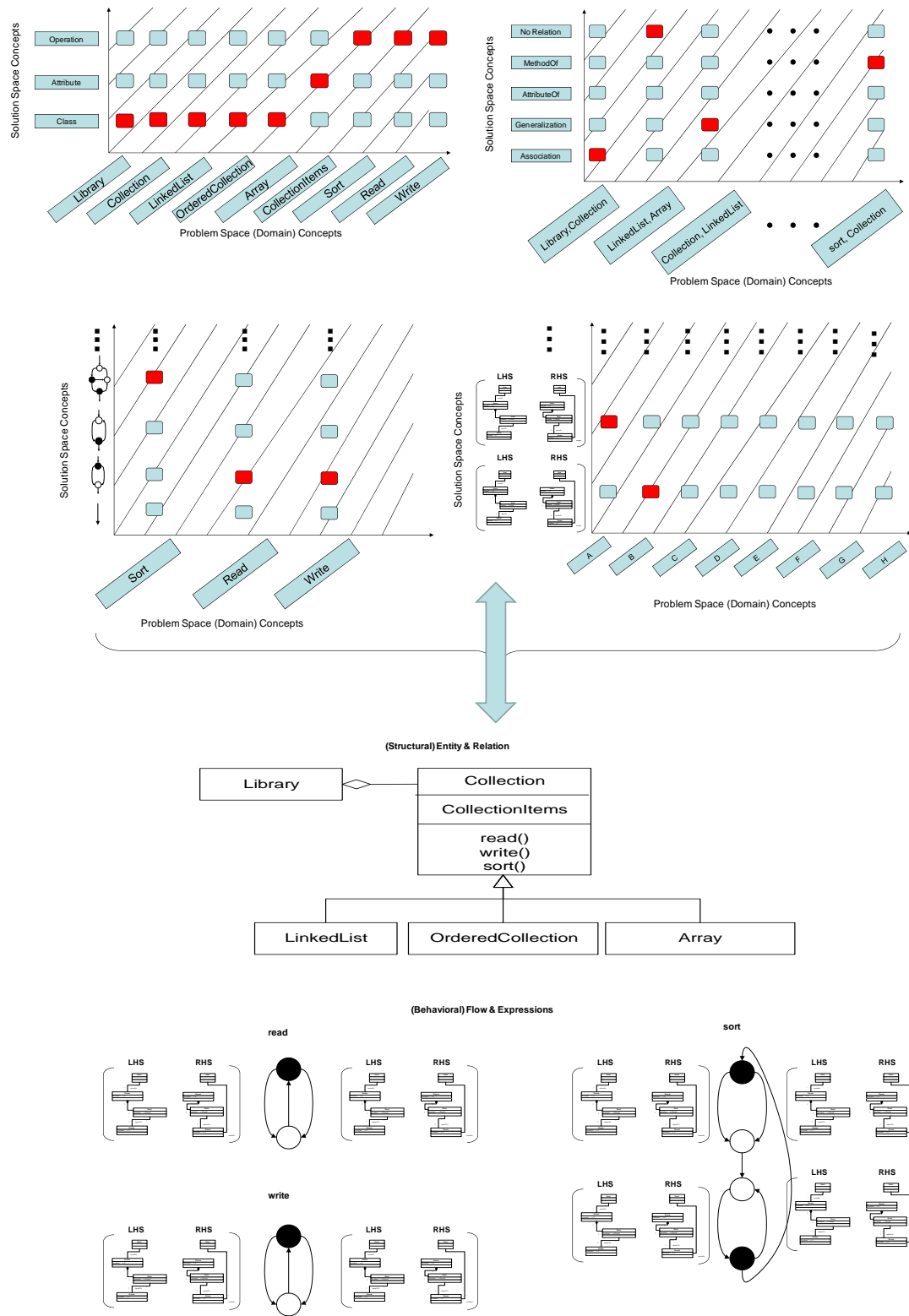


Figure 6.1: An Overall Diagram of Successive Design Decisions Leading to the Final Product

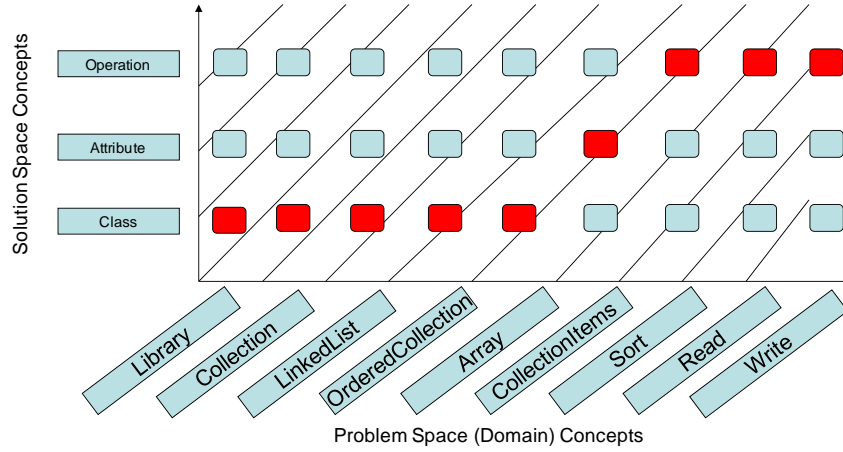


Figure 6.2: Structural Entity Design Space for the Library Example

Designer decisions in entity design space (represented in two dimensions) are marked in red in Figure 6.2. On the other hand, the decisions in relation design space are shown in red in Figure 6.3.

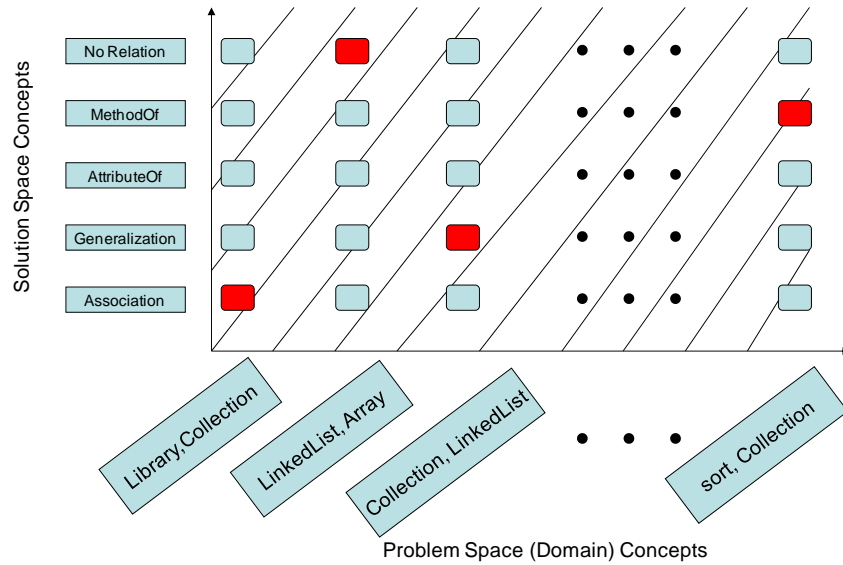


Figure 6.3: Structural Relation Design Space for the Library Example

The behavioral decompositions are given in Figure 6.4 and Figure 6.5. The selected design alternatives concerning Flow Design Space and Expression Space are shown in red in Figure 6.4 and Figure 6.5 respectively. As defined in Section 5.1.2, the icons on the y-axis in Figure 6.4 represent the predefined property P_{Flow} as *cubic graph* alternatives

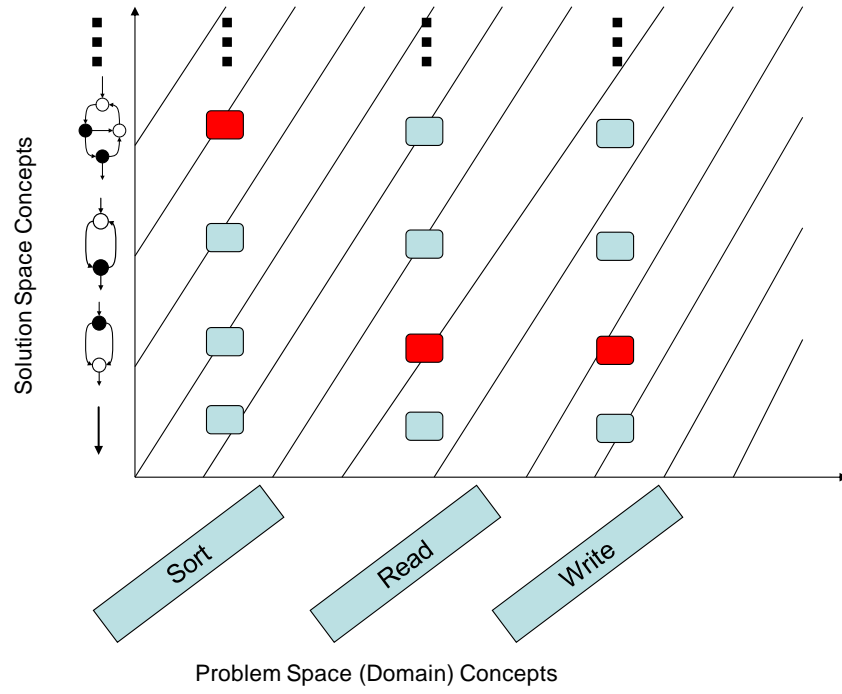


Figure 6.4: Flow Design Space for the Library Example

for the flow of operational domain concepts. On the other hand, the icons on the y-axis in Figure 6.5 represent the predefined property $P_{Expression}$ as *graph grammar rules* for the state transition steps of domain operations.

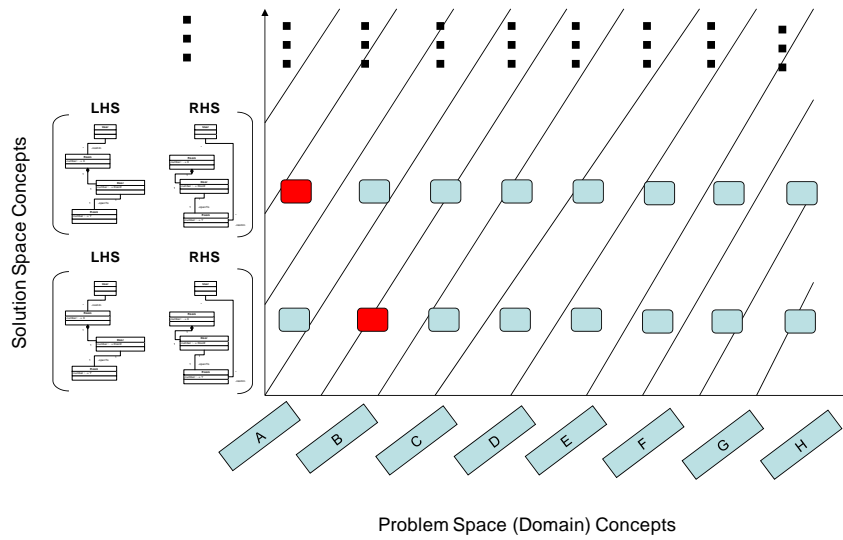
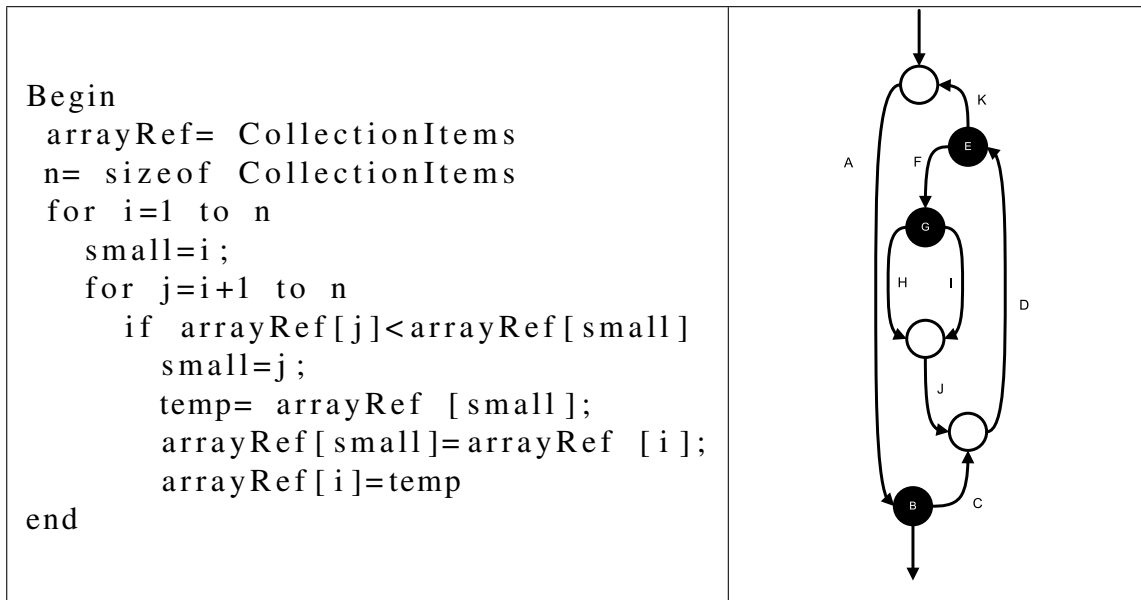


Figure 6.5: Expression Design Space for the Library Example

The mapping of problem space concepts to solutions space concepts as shown in Figure 6.4 and Figure 6.5 warrants an example for clarification. Therefore, we include an example design of the sort operation for an *Array* class and its corresponding cubic-graph representation in Table 6.1. Graph grammar examples are found in [157, 162]. The graph grammar examples provided there illustrate how the author and others specified software behavior using preconditions and postconditions.

Table 6.1: A Design of Sort Behavior for an Array Class



The analysis of this example implies that the Library designer decomposes these design spaces using solution space concepts to generate a library system. This process is viewed in our model as an effort towards producing an organized system. As emphasized throughout this dissertation in general and in Chapter 5 in particular, design spaces consist of all possible alternatives. This fact is shown in figures associated with the library system in this section. Therefore, the initial state of the library design spaces represent all possible alternatives with minimal organization (implying high entropy). The successive design decisions, shown as red markings within design spaces of the library example, introduce comparatively higher organization (implying low entropy).

6.2 Information Theoretical Analysis Example

As explained in Section 5.2, a software system is represented with an arbitrary number of interacting variables, in which the process of transforming uncertainty to certainty is an activity towards higher level of organization. Thus, design by imposing organization leads to lower levels of entropy. This example, through the use of hierarchical organization, demonstrates the utilization of communication channel in design.

Table 6.2 shows the specification of the representative example, with which we demonstrate information theoretical analysis of software design. In this example, the abstraction level and style selected by the designer is *Object-Oriented*. In summary, following the theory developed in Chapter 5, we first map *Class Attributes* into set variables (*cf*, Table 6.3, Figure 6.6), producing software system $S = \{V1, V2, V3, V4\}$. Then, we observe the run-time values of these variables for quantitative analysis (*cf*, Table 6.4). Next, we demonstrate the relationships between variables as communication channels (*cf*, Figure 6.7, Table 6.5, Figure 6.8). Using communication channels, we show the hierarchical decomposition and information theoretical quantification for this example (*cf*, Figure 6.9, Figure 6.10, Table 6.6).

The representative example of this section is presented below, following the theoretical development as explained in Section 5.2. As stated in Section 5.2.1, we start with mapping of the software system to a set-theoretical representation. We define a set of four variables, $\{V1, V2, V3, V4\}$, for the software system. In this example, variables represent *Attributes* defined within *Class A* and *Class B*. Mapping between *Class Attributes* and set variables is given in Table 6.3. The example software system is represented as a set, $S = \{V1, V2, V3, V4\}$.

Table 6.2: Example Software System

<pre> public class A { private int x; private int d; public A(int p, int p2) { x=p; d=p2; } public int getX() { return x; } public int getD() { return d; } public int calculate() { return d*x/100; } public void changeValue(boolean b) { if(b) x=x+5; else d=d+110; d=calculate(); } } </pre>	<pre> public class B { private int s; private int z; public B(int p, int p2) { s=p; z=p2; } public int getS() { return s; } public int getZ() { return z; } public void calculate(A a) { z=a.calculate()+s; a.changeValue(s%10<5?true:false); if(a.getD()%10<5) s=s+a.getX(); else s=s*a.getX(); } } </pre>
---	---

In this example, *Class* definitions demonstrate the hierarchy between *Classes* and *Attributes*. Figure 6.6a depicts the tree representation of the hierarchy of the example system. A set-theoretical representation of this hierarchy is shown in Figure 6.6b. *Class A* is mapped into a set $\{V1, V2\}$ and *Class B* is mapped into a set $\{V3, V4\}$.

Table 6.3: Software Concepts to Set Concepts

Software	Set
Example System	$S = \{V1, V2, V3, V4\}$
A.x	V1
A.d	V2
B.s	V3
B.z	V4

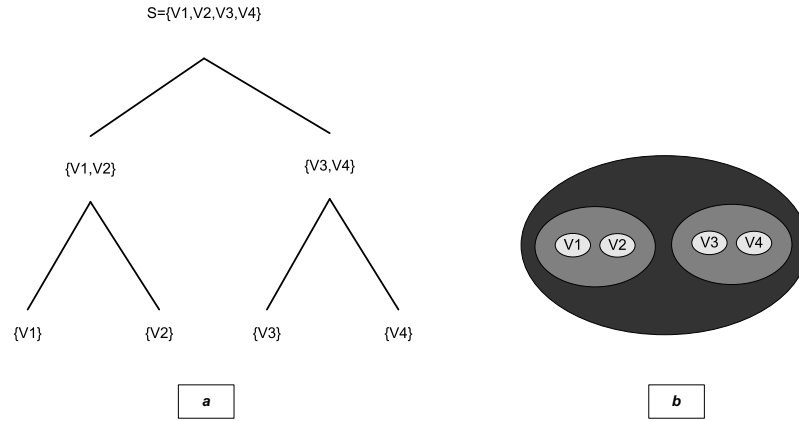


Figure 6.6: Hierarchical Representation of the Example: a) Tree Representation of the Hierarchy of the Example b) Set Representation of the Hierarchy of the Example

The next step is the observation of values associated with variables of the system. For the example system, values associated with each variable are observed with the execution of the *calculate* method shown in Table 6.2. Each *calculate* execution changes values of variables so that each transformation of the example system is observed. Observations of four variables for nine cycles are given in Table 6.4 ¹.

¹The complete observation values can be found in Appendix B.

Table 6.4: Part of the Observed Values

Cycle #	V1	V2	V3	V4
1	6	0	7	1
2	6	6	42	7
3	11	0	53	42
4	16	0	69	53
5	16	17	1104	69
6	21	3	1125	1106
7	21	23	1146	1125
8	21	27	24066	1150
9	21	28	505386	24071

As stated in Section 5.2.2, the interaction between variables in the example software system is represented using the communication channel formalism. In terms of mapping to the channel representation, the relationship between V1 and V2 based on observed values within nine cycles is shown in Figure 6.7. As shown in Table 6.4, while $V1 = 6$ in the first and second cycles, $V2 = 0$ for the first cycle and $V2 = 6$ for the second cycle. This relationship is shown in the communication channel of Figure 6.7 as two communication links starting from the first node of V1 and ending in the first and third nodes of V2.

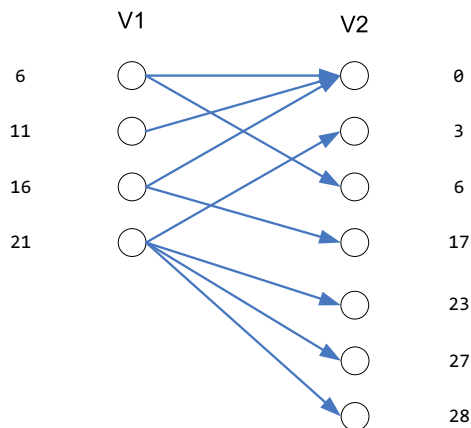


Figure 6.7: Actual Communication Between V1 and V2

Transmission between variables is calculated using the *transmission formula 5.2.2* given in Section 5.2.2. Pairwise relationships and corresponding transmission values for the example system are shown in Table 6.5. Diagrammatic representation of six pairwise relations are shown in Figure 6.8. In Figure 6.8, pairwise relations are indicated by arrows whose thickness is directly proportional to transmission values. For example, transmission value between $V3$ and $V4$ is 7.56 in Table 6.5, and it is shown as the strongest pairwise relationships in Figure 6.8 with the thickest arrow.

Table 6.5: Transmission Between Variables

Variables	Transmission
$V1 \Leftrightarrow V2$	2.81
$V1 \Leftrightarrow V3$	4.28
$V1 \Leftrightarrow V4$	3.91
$V2 \Leftrightarrow V3$	3.86
$V2 \Leftrightarrow V4$	3.68
$V3 \Leftrightarrow V4$	7.56

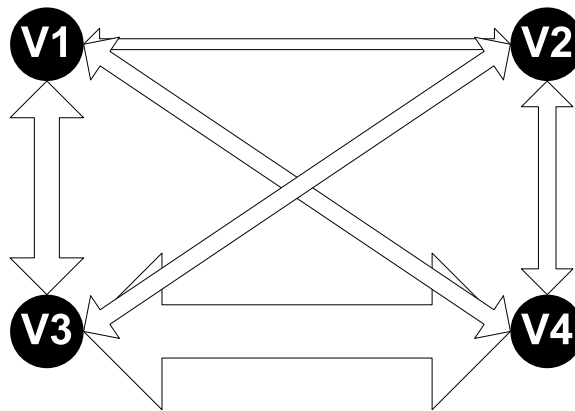


Figure 6.8: Transmission Between Variables

As stated in Section 5.2.3, partitioning of these interactions creates the hierarchy within software systems. Groups of highly interacted elements constitute the subsystems of the

system, as such producing the desired hierarchy. Following these principles, hierarchical decomposition of the example system is shown diagrammatically in Figure 6.9.

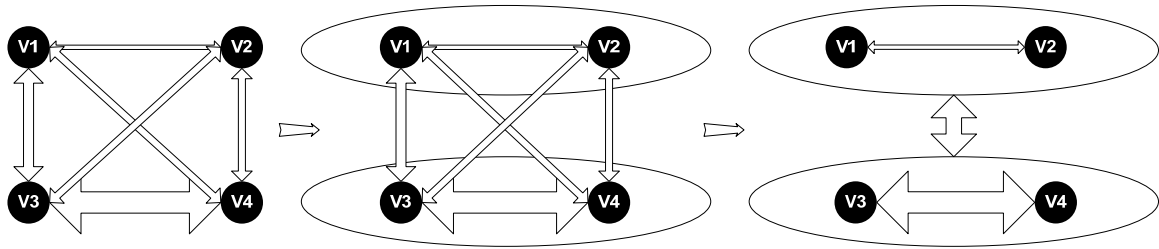


Figure 6.9: Decomposition of Interaction Among Variables

As seen in Figure 6.9, V1 and V2 are grouped into one subsystem and V3 and V4 are grouped into another subsystem. The interaction between two subsets is also represented as a communication channel in Figure 6.10.

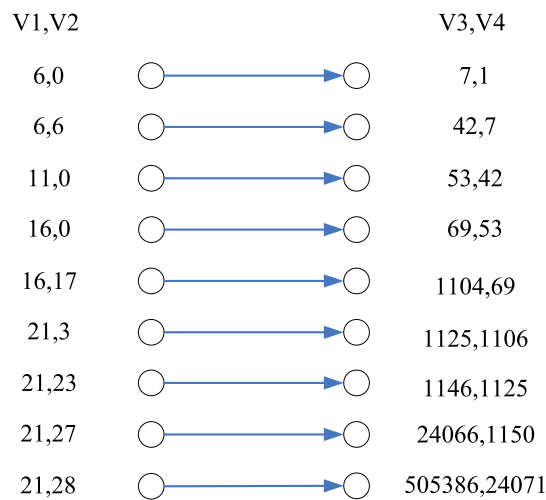


Figure 6.10: Communication Between Two Subsystems

Communication between variables within the example software system can be decomposed in many different ways. Table 6.6 shows various decomposition possibilities for the example system. For example, the decomposition $S = \{\{V2\}, \{V1, V3, V4\}\}$ partitions the example system into two subsystems. First subsystem, $\{V2\}$, consists of one software element with no internal communication. Second subsystem $\{V1, V2, V3\}$ is composed of three elements with total of 12.44 transmission value among the variables,

$V1, V3$, and $V4$. Transmission value between the subsystem $\{V2\}$ and the subsystem $\{V1, V3, V4\}$ is 4.27.

Table 6.6: Decomposition of Interaction Among Variables and Subsystems

Decomposition	Transmission among elements	Transmission within elements
$S = \{V1, V2, V3, V4\}$	16.71	16.71
$S = \{\{V1\}, \{V2\}, \{V3, V4\}\}$	9.15	0+0+7.56
$S = \{\{V2\}, \{V1, V3, V4\}\}$	4.27	0+12.44
$S = \{\{V1, V2\}, \{V3, V4\}\}$	6.34	2.81+7.56

As explained in Chapter 3 and Chapter 5, the communication between variables and decomposition of these communications create the organization within the example software system. The analysis of this example demonstrates that a communication channel representation is created for the example system. In the process, we used a set-theoretical representation and grouping of subsystems based on the observed interaction values. This activity produced the hierarchy and thus produced the communication channel representation of the example system.

Figure 6.11 demonstrates the broad disciplines from which we have obtained the appropriate formulas to represent software design as an entropy-reduction process. From information theory, we used the mathematical model of communication system defined by Shannon [37] to model communication or information exchange among software elements. We adopted the partition formalism definition of Shannon's entropy given by Papoulis [65]. Papoulis defines a *partition* as a collection of mutually exclusive events whose union is universal set. *Entropy of a partition* is defined as a measure of uncertainty about the occurrence or nonoccurrence of any event A_i of a partition U . It is denoted by $H(U)$. The postulates of the Entropy function, H are

1. $H(U)$ should be a continuous function of $p_i = P(A_i)$.

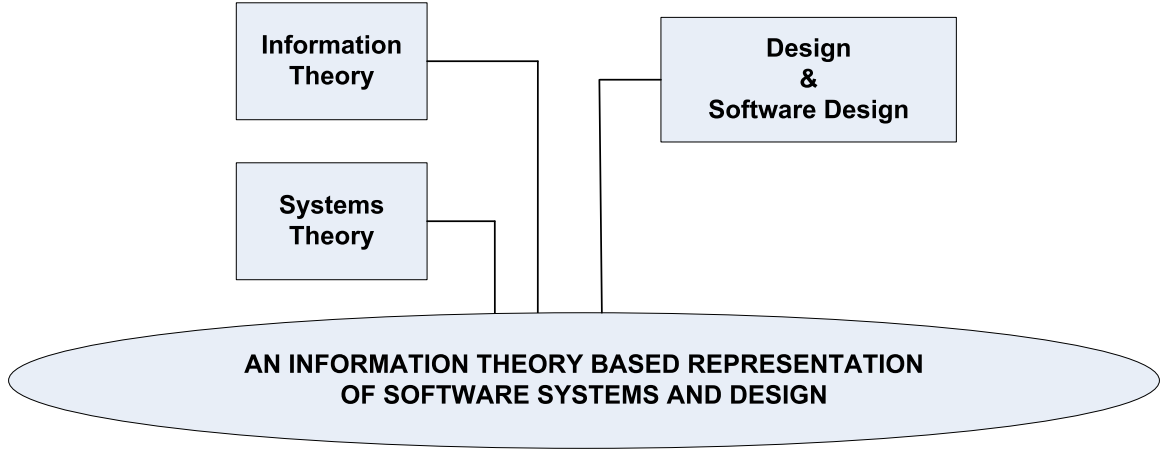


Figure 6.11: Conceptual Approach

2. If $p_1 = \dots = p_n = 1/N$, then $H(U)$ should be a monotonic increasing function of N ; that is, for $U = [A_1, \dots, A_m]$ $p(A_1) = \dots = p(A_m) = 1/M$ and $Y = [B_1, \dots, B_n]$ $p(B_1) = \dots = p(B_n) = 1/N$ $M < N$ implies $H(U) < H(Y)$.
3. If one of the elements of U be broken down into two successive events, then a new partition B is formed and $H(B) \geq H(U)$.

For multivariate interactions, we applied multivariate information transmission which was defined by McGill [38]. He defined correlation formula which is the extension of Shannon's communication system for multivariate cases. Correlation among partitions U_1, U_2, \dots, U_n is the total information transmission and by definition as follows:

$$C(U_1, U_2, \dots, U_n) = \sum_{i=1}^n H(U_i) - H(U_1 \cdot U_2 \cdot \dots \cdot U_n). \quad (6.2.1)$$

From systems theory, we used Ashby's systems approach. He defined complex systems as a set of variables with constraints [90]. As elaborated in Chapter 5, we used Ashby's technique to map a software system into a complex system and then we used the organization measurement technique defined by Rothstein [29], Watanabe [33] to map into the Entropy calculation of Shannon. Rothstein and Watanabe, being physicists in their original training, primarily investigated the organization from the point of view of the

interaction among speeding gas molecules. They demonstrated that order of gas molecules is related with uncertainty of their positions and entropy calculation is the measure the orderliness for gas molecules. The strength of organization is measured by the balance between the entropy of the components with respect to the entropy of the whole. Then the degree of organization can be defined as

$$\text{Organization} = (\text{sum of entropies of part}) - (\text{entropy of whole}). \quad (6.2.2)$$

As elaborated in Chapter 5, we applied this formula to the organization of software. We used hierarchical system definition from Simon to demonstrate that software design is a hierarchical decomposition of complex system [39]. Formal treatment of hierarchical decomposition was also studied by Conant [41]. Building on Simon and Ashby, Conant provided a technique detecting subsystems of a complex system using communication formalism of Shannon. As elaborated in Chapter 5 and as shown in Java implementation in Appendix A we applied the following steps to the decomposition of software system:

1. Define a set of K variables for a given software system. Each variable is denoted by X_j where $1 \leq j \leq K$. Software system is a set of X_j , denoted by the set $S = \{X_1, \dots, X_K\}$.
2. Observe the K variables for N cycles, and obtain a total of $K \cdot N$ different values. Observed number of occurrences of the event in consideration $X_j = X_j^i$ is denoted by $n_{X_j^i}$, such that $\sum_{i=1}^{n_j} n_{X_j^i} = N$.
3. Represent the interaction between two system variables, X_i and X_j using communication channel as follows:
 - the value set, P_i , which is associated with X_i , is taken as a channel input set S ,
 - the value set, P_j , which is associated with X_j , is taken as a channel output set R , and

- then channel probability is, $P(S_k, R_l) = \frac{n_{S_k R_l}}{n_{R_l}}$, where observed number of occurrences of the event in consideration $\{R_l = P_j^l\}$ is denoted by n_{R_l} , the number of occurrences of the event $\{S_k R_l = P_i^k P_j^l\}$ is denoted by $n_{S_k R_l}$.

4. Calculate the interaction in the channel using entropy of a variable, X_j , denoted $H(X_j)$, is used as a measure of the variability of X_j . This value is calculated for each variable of a given software system by the formula:

$$H(X_j) = - \sum_{i=1}^{n_j} \frac{n_{X_j^i}}{N} \log \frac{n_{X_j^i}}{N}. \quad (6.2.3)$$

The observed transmission between two variables, X_i , and X_j , is defined as follows:

$$T(X_i : X_j) = H(X_i) + H(X_j) - H(X_i X_j). \quad (6.2.4)$$

5. Group the variables X_j into sets to decompose software. The set $S_i = \{S_i^1, \dots, S_i^{n_i}\}$ represents a subsystem of given software system, where $\cup_{i=1}^r S_i = S$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$. The total interaction is decomposed into transmission such that

$$C_{Total}(X_1 X_2 \dots X_K) = \sum_{i=1}^r C_{Total}(S_i) + C(S_1, S_2, \dots, S_r) \quad (6.2.5)$$

where $C_{Total}(S_i)$ is the transmission within an element, S_i , and $C(S_1, S_2, \dots, S_r)$, correlation formula.

In conclusion, based on Simon's [18] design definition, we posited that software design is the transformation of current conditions into preferred conditions. We mapped transformation idea to software design. For this mapping we used Tanik's abstract design definition [13, 9, 17] which addresses all design steps from requirements to final product. To demonstrate software design as an entropy-reduction process, we modeled software design using design spaces which were introduced by Aksit and Tekinerdogan [44]. This modeling technique enable us to represent software design as hierarchical decomposition.

As a result of the application of these techniques, we provided a novel modeling for information theoretical representation of software systems and design. The results given in this chapter show that information theoretical representation of software systems and design formalize software design decomposition. The results demonstrate that software design fundamentally is an entropy-reduction process. The results explicitly show that classification of software elements into composite software elements, such as class, module, provide an organization which provide entropy reduction.

6.3 Summary

In this chapter we showed two classes of representative software examples which provide calculation instances for the formulas presented in Chapter 5. The Appendix A shows the Java program for the calculations and Appendix B contains the associated data. The first class of representative examples involve design space decomposition leading to entropy reduction of software design. The second class of examples is the communication-channel representation of software design. With these examples, we demonstrated that software design is a hierarchical decomposition. This implies that in software design the successive design decisions leads to reduced entropy and as such allows for analysis of all kinds of designs with information theoretical mechanisms.

Chapter 7

SUMMARY, CONCLUSIONS, AND FUTURE WORK

In this chapter, we provide an overall summary of the findings of this work while reviewing what was proposed in the first chapter. We offer conclusions on our outcomes and describe potential for future work in systematizing design in software engineering including additional incremental results.

7.1 Summary and Conclusions

According to Simon [18], “Design is the transformation of existing conditions into preferred ones.” Throughout the dissertation we assumed that software design is also the transformation of current conditions or state into preferred conditions or state. Initially, our intent was to formally analyze this transformation process for software design. As a result of this analysis, we hoped to develop a formal system perspective for software design [146]. We have achieved this objective by realizing software design as an entropy-reduction process and by representing software systems with communication channels.

We investigated software design as a hierarchical decomposition of design spaces. We realized that before initiating the software design process there is minimal initial organization (higher entropy) and therefore high uncertainty exists. The design decisions carrying out design activities (hierarchical decompositions) reduce uncertainty and therefore introduce comparatively higher organization (lower entropy).

This observation described in Chapter 5 and associated experiments explained in Chapter 6 revealed that design is an entropy-reduction process. Probability and information theory deal with uncertainties and provide the mathematical framework for entropy calculations. Using these theories, we developed the communication channel formalism of software design and therefore we prepared the groundwork for the analysis of software designs as an entropy reduction process.

We started our approach by mapping software systems to set-theoretical representations. Then we showed the properties of the information transfer between variables and demonstrated the interactions among variables as representable by communication channels. Using the communication-channel representation, we explained hierarchical decomposition of software design. As a result, through hierarchical decomposition, we analyzed software design as an entropy-reduction process.

Our key goal was to systematize software design and understanding of the design process in software engineering based on first principle foundations of science and the practices of “hard” engineering disciplines. We achieved this goal with formal demonstration that:

- software design is a hierarchical decomposition and addresses all steps from requirements to the final product, and
- software design imposes an organization and reduces entropy through successive transformations.

The communication-channel representation of software systems opens up further useful possibilities for applying engineering mathematical analysis to software development. This indicates that current understanding and informal representations of software design, using our results, can further evolve into a type of inquiry involving classical engineering mathematics and concepts. The current results offer an opportunity for transformative impact that could influence the area of software engineering by providing a relatively more stable context for discussing the representation of software design than was currently available before our work. In summary, our contributions and their impacts are summarized

Table 7.1: Contributions and Associated Impacts

Contributions to Software Design	Explanations/Impacts
Entropy Reduction View (See also Section 5.3)	Better understanding of the design process in software engineering based on classical engineering principles
	Establishing a new formal foundation for future software research
	Enabling the use of a known mathematical formalism of information theory, and therefore opening up possibilities to investigate software design with established engineering techniques and mathematics
Communication Channel Representation (See also Section 5.3)	Opening new possibilities in software research, connecting software with information and coding theory
	Providing a stronger bridge between established engineering methods and software design

in Table 7.1. Together these theoretical contributions advance the state of software design theory as hypothesized in Chapter 1.

As explained in Chapter 4, we presume that Software development is in a “pre-engineering” phase that is analogous to many pre-engineering phases found in engineering disciplines of the past. As Royce [25], Yeh [163], Aksit [44], and Tanik [17, 164], stated from different perspectives, design principles similar to “hard” engineering principles are needed to overcome problems associated with software development. Future researchers by using these observations and our entropy reduction formalism could further improve and/or develop improved mathematical underpinnings for software development. Considering our information-theoretical formalism, further formalization of software development towards hard engineering disciplines is in relatively easy reach.

The communication-channel representation of software systems opens up possibilities for further axiomatization of software design process and thus improving the affinity of software design to classical engineering. On the other hand, viewing various dynamic complex processes such as life processes (*i.e.*, cell division) as a progression towards

organization (lower entropy) opens up broad applications areas to be studied as an entropy reduction process.

While Chapter 5 presents our formalism, Chapter 6 confirms the validity and achievability of the theory through representative examples.

7.2 Future Work

The act of design starts with recognition of a problem. A designer determines the problem according to his or *her parameter(s) of interest*. A parameter of interest corresponds to a designer's intention and includes the criteria that will drive the design. The designer is required to make decisions based on many parameters and to make choices among possible alternatives, while evaluating the feasibility of each choice. All these actions on the part of the designer, require measurement whether the designer is consciously aware or not. To make sense of a measurement one needs an agreed-upon metric. Therefore, the calculation of entropy by definition depends on the parameter of interest. Therefore, low entropy calculation concerning a design implies by definition organization with respect to the parameter of interest. Naturally, conflicting parameters of interest have different effects in an entropy reduction process. Investigation of software design with conflicting parameters of interest is a potential future work of this study. Software Quality factors and their impacts during software design could also be investigated in the same context of many parameter of interest problem.

Moreover, differences between decomposition spaces and decomposition tools, such as the semantic gap [20], the mapping of problem space concepts and solution space concepts, affect entropy reduction process. There are multiple ways of producing a decomposition scheme, which are each driven by parameter of interest. In other words, the designer develops a parameter of interest for his or her design decomposition and performs the decomposition accordingly. Different design techniques can be also interpreted as conflicting parameters of interest.

This research revealed the need for a mathematical theory to analyze the relationship between decomposition at run-time and programming statements, such as selective, repetitive, and assignment. Although we took useful steps toward that end here, more work remains to create a complete, self-consistent, testable theory. Additional theory could be created to explain and provide deeper guidance on developing programs using information and coding theory. In this context, here are ways that future work could logically proceed:

- Information Theory and Representation of Computer Programs as a Partition Transformation, and/or
- Information Theory and Representation of Computer Programs as a Generalized Communication System.

The following subsections highlight these two areas.

7.2.1 Information Theory and Representation of Computer Programs as A Partition Transformation

Computer programs can be described and analyzed in combinatorial terms. As computer scientists, we observe that the execution of computer programs are repeatable and the outputs are not randomly produced. When we omit all side effects in a specification, it is expected that given a specific input, one gets a specific output when executed with the same program. Software programs could be formulated as a process that maps input possibilities to output possibilities. Furthermore, this mapping could be represented in combinatorial terms and can subsequently be analyzed with an information-theoretical formalism. Interpretation of computer programs as a partition (decomposition) transformation is shown in Figure 7.1. All computer programs transform an existing decomposition of software system into a new decomposition of software system.

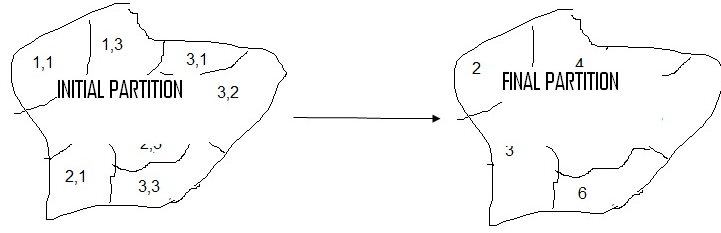


Figure 7.1: Interpretation of Computer Programs as a Partition Transformation

As a representative example to elaborate on this idea, we introduce an analysis of the program *ADD*, that calculates the summation of two integers. An instance of *ADD* program for $\{X, Y \mid 0 < X < 4 \text{ \& } 0 < Y < 4\}$, where X and Y are inputs to the program, is specified in Table 7.2:

Table 7.2: An Instance of *ADD* Program for $\{X, Y \mid 0 < x < 4 \text{ \& } 0 < y < 4\}$

X	Y	Output
1	1	2
1	2	3
1	3	4
2	1	3
2	2	4
2	3	5
3	1	4
3	2	5
3	3	6

The *ADD* program example then can be stated as a mapping from one partition to another partition. Figure 7.2a, Figure 7.2b, and Figure 7.2c, visually depict the prior partition, intermediate partition, and final partition, respectively. These transformations could be modeled as an entropy reduction process, and mapped into Communication channel formalism.

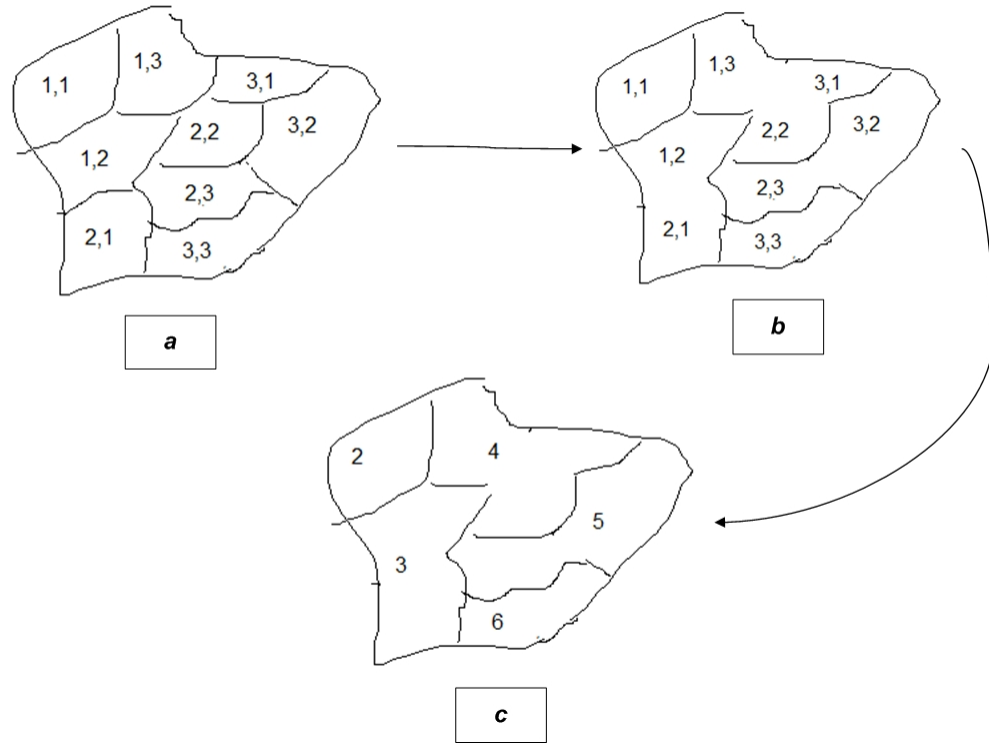


Figure 7.2: Partition Transformations for the ADD Program: a) Initial Partition for the ADD Program b) Intermediate Partition for the ADD Program c) Final Partition for the ADD Program

As a second representative example for partition transformation, we introduce modeling programs developed with structured constructs. As reviewed in Section 4.2.3 structured programs are realized with three fundamental constructs, namely sequence, selection, and repetition. Therefore, if a computer program is viewed as an integrated set of these three constructs then following our approach, incoming symbols are considered as the input, and the outgoing symbols stand for the output for the corresponding input. Therefore, realization of these constructs by using communication channels would allow us to represent programs as integrated communication systems or an integrated set of communication channels. Figure 7.3 diagrammatically summarizes each of the three constructs as communication channel models.

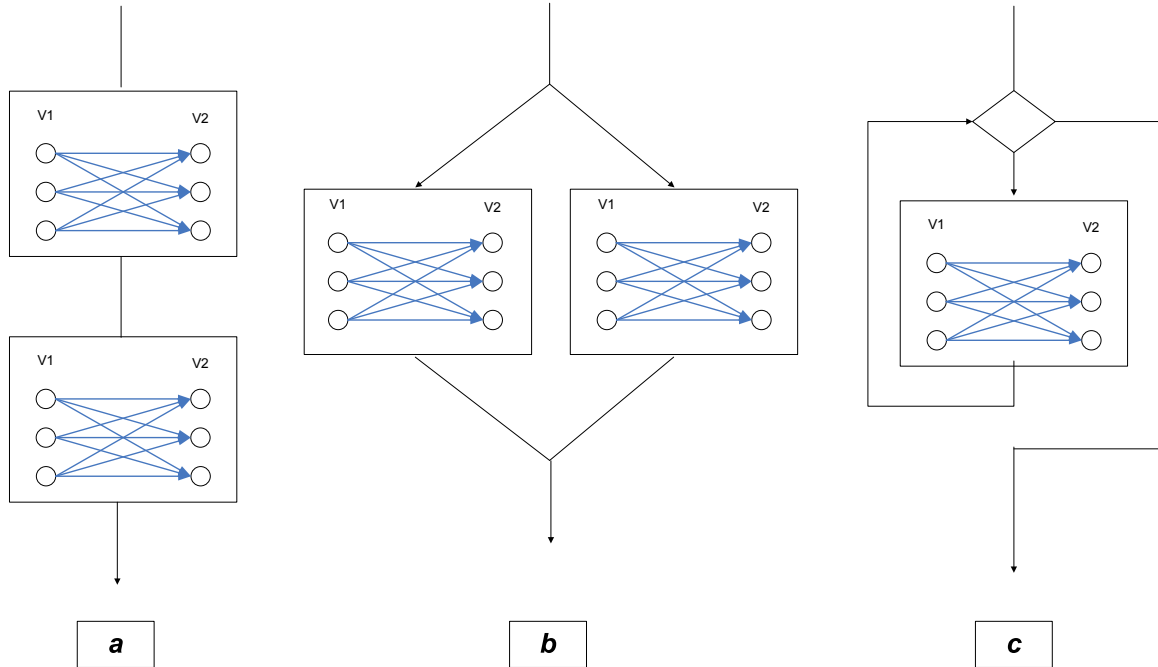


Figure 7.3: Communication Channel Representation of Program Constructs: a) Sequence b) Selection c) Repetition

7.2.2 Information Theory and Representation of Computer Programs as A Generalized Communication System

The transition from initial partition to final partition in theory can only have a communications channel as shown in the previous section. However, in practice an encoder and a decoder is needed for the formulation of a *parameter of interest* as shown in Figure 7.4. In this section, we provide a representative example in the form of a Fourier encoding of a representative problem, namely discovering a counterfeit coin among others.

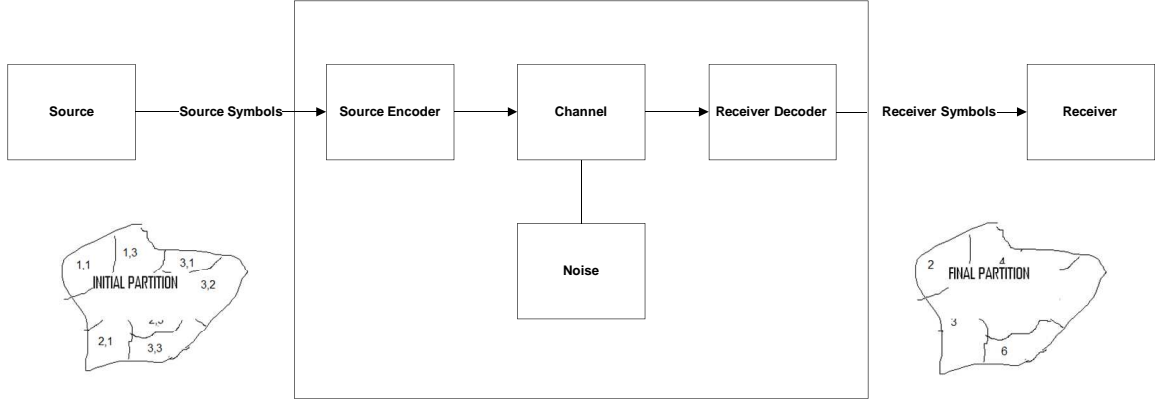


Figure 7.4: Representation of Computer Programs as A Generalized Communication System

The discovery of counterfeit coin problem can be specified as follows. Assume one is given a scale and four coins. Three of the coins are equal in weight, but the fourth is counterfeit or defective, and weighs different than other three coins. The objective is to devise a way to determine the counterfeit coin. The analysis of this problem proceeds as follows:

Let us assume that the weight of counterfeit coin is 'a' and the weight of other coins is 'b'. Possible Input configurations are $C1 = [a, b, b, b]$, $C2 = [b, a, b, b]$, $C3 = [b, b, a, b]$, and $C4 = [b, b, b, a]$ and corresponding output, as depicted in Figure 7.5, are $C1 = 1$, $C2 = 2$, $C3 = 3$, $C4 = 4$. It should be noted that Figure 7.5 demonstrates an instance of the problem for $a, b \in \{1, 2, 3\}$. As a result, Figure 7.5 demonstrates the transformation of the input configuration (initial partition) to the final partition. As outlined in previous chapters, this transformation can be studied as a communication channel. Furthermore, the utilization of encoding and decoding is computed with the Fourier Matrix as shown in Figure 7.6 and Figure 7.7 respectively. The Figure 7.8 depicts the complete example of communication channel representation including encoding and decoding.

$a, b \in \{1,2,3\}$

- Possible Configurations

Initial Partition

- [1,2,2,2]
- [1,3,3,3]
- [2,1,2,2]
- [2,2,1,2]
- [2,2,2,1]
- [2,3,3,3]
- [3,1,3,3]
- [3,3,1,3]
- [3,3,3,1]
- [3,2,3,3]
- [3,3,2,3]
- [3,3,3,2]



Final Partition

$[a,b,b,b]$ $[1,2,2,2]$ $[1,3,3,3]$ $[2,3,3,3]$ 1	$[b,b,a,b]$ $[2,2,1,2]$ $[3,3,1,3]$ $[3,2,3,3]$ 3
$[b,a,b,b]$ $[2,1,2,2]$ $[3,1,3,3]$ $[3,2,3,3]$ 2	$[b,b,b,a]$ $[2,2,2,1]$ $[3,2,2,1]$ $[3,3,1,2]$ 4

Figure 7.5: Partition Transformation for Coin Example

Fourier	Coins	R
$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$	$\begin{bmatrix} a \\ b \\ b \\ b \end{bmatrix} *$	$= \begin{bmatrix} a+3b \\ a-b \\ a-b \\ a-b \end{bmatrix}$
$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$	$\begin{bmatrix} b \\ a \\ b \\ b \end{bmatrix} *$	$= \begin{bmatrix} a+3b \\ (a-b)i \\ b-a \\ (b-a)i \end{bmatrix}$
$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$	$\begin{bmatrix} b \\ b \\ a \\ b \end{bmatrix} *$	$= \begin{bmatrix} a+3b \\ b-a \\ a-b \\ b-a \end{bmatrix}$
$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$	$\begin{bmatrix} b \\ b \\ b \\ a \end{bmatrix} *$	$= \begin{bmatrix} a+3b \\ (b-a)i \\ b-a \\ (a-b)i \end{bmatrix}$

Figure 7.6: Fourier Transformation of the Inputs

InverseFourier	R^*	Answer
$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$	$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \xrightarrow{[1,2,3,4]^*} = 1$
$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$	$\begin{bmatrix} 1 \\ i \\ -1 \\ -i \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \xrightarrow{[1,2,3,4]^*} = 2$
$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$	$\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \xrightarrow{[1,2,3,4]^*} = 3$
$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$	$\begin{bmatrix} 1 \\ -i \\ -1 \\ i \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \xrightarrow{[1,2,3,4]^*} = 4$

Figure 7.7: Inverse Transformation of the Output

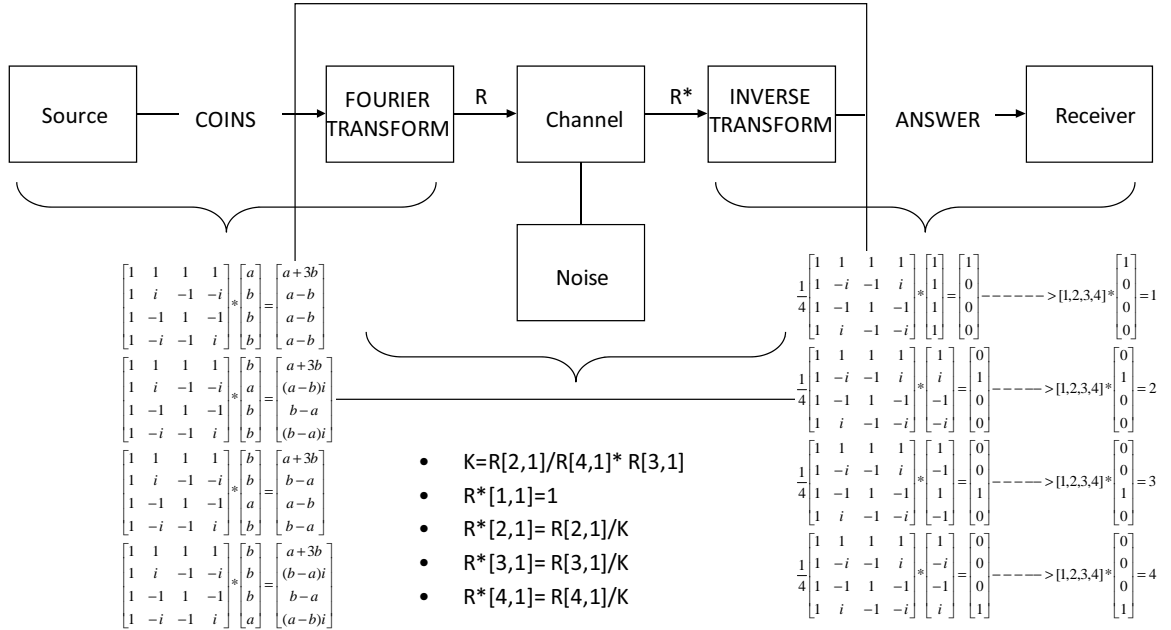


Figure 7.8: Communication-channel Representation of Coin Example

As a second example for the representation of programs as a generalized communication system, we discuss the implementation of Polynomial multiplication with Fast Fourier Transform (FFT). Polynomial Multiplication with FFT [165] can be viewed as a generalized communication system. Figure 7.9 shows the equality of ordinary

multiplication and the steps involved in the use of FFT. The use of FFT specifically involves Evaluation, Pointwise Multiplication, and Interpolation steps as depicted in Figure 7.9. These FFT steps corresponds to encoding, channel computation, and decoding respectively in the communication channel formalism and is shown in Figure 7.10.

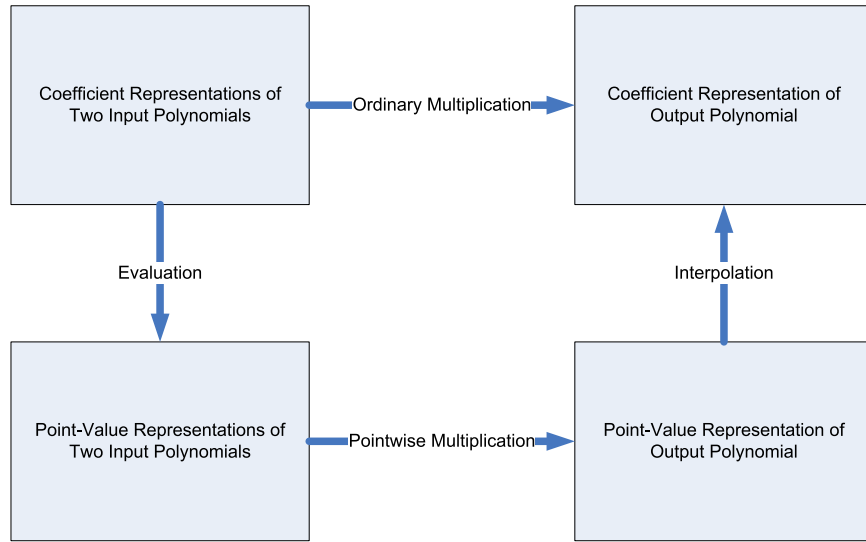


Figure 7.9: Polynomial Multiplication (Adapted From [165])

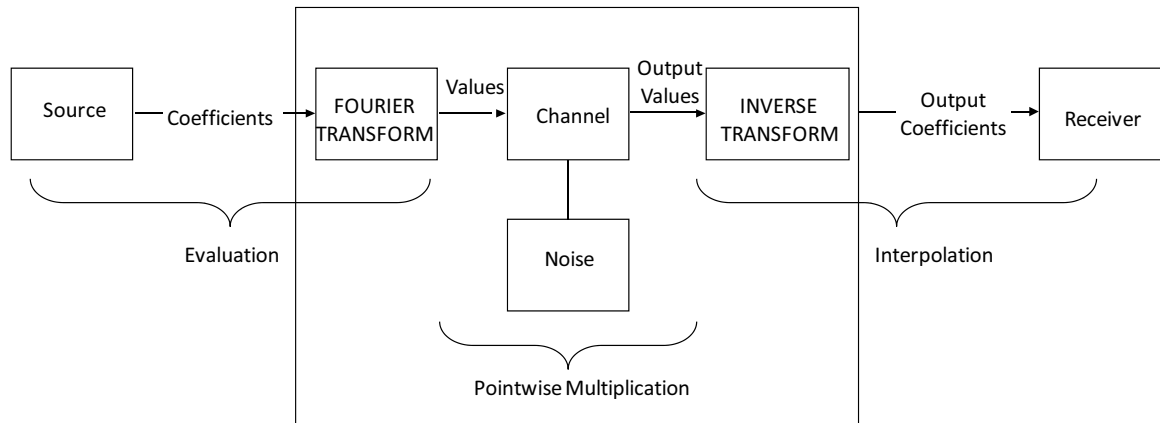


Figure 7.10: Communication-channel Representation of Polynomial Multiplication

Overall, these examples show the power of our approach and indicate the value of further exploration of software design via mathematical formalism based on information theory.

LIST OF REFERENCES

- [1] M. Asimow, *Introduction to Design*. Prentice-Hall Series in Engineering Design, Englewood Cliffs, N. J.: Prentice-Hall, 1962.
- [2] M. J. French, *Conceptual Design for Engineers*. London: Design Council, 2nd ed., 1985.
- [3] S. Dasgupta, *The Structure of Design Processes*, pp. 1–67. Academic Press Professional, Inc., 1989.
- [4] S. Pugh, *Total Design: Integrated Methods for Successful Product Engineering*. Wokingham, England ; Reading, Massachusetts: Addison-Wesley, 1991.
- [5] G. Pahl and W. Beitz, *Engineering Design: A Systematic Approach*. London ; New York: Springer, 1996.
- [6] V. Hubka and W. E. Eder, *Design Science: Introduction to Needs, Scope and Organization of Engineering Design Knowledge*. Berlin ; New York: Springer, 1996.
- [7] D. Braha and O. Z. Maimon, *A Mathematical Theory of Design: Foundations, Algorithms, and Applications*. Applied optimization, Boston: Kluwer, 1998.
- [8] G. E. Dieter, *Engineering Design : A Materials and Processing Approach*. McGraw-Hill series in mechanical engineering, Boston: McGraw-Hill, 3rd ed., 2000.
- [9] M. M. Tanik and A. Ertas, “Design as a basis for unification: System interface engineering,” in *Computer Applications and Design Abstractions (ASME)*, vol. 43, pp. 113–114, 1992.
- [10] M. M. Tanik, A. Ertas, and A. H. Dogru, “Techniques in abstract design methods in engineering design development,” *Control and Dynamic Systems*, vol. 61, pp. 285–328, 1994.
- [11] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Prentice-Hall Series in Automatic Computation, Englewood Cliffs, New Jersey: Prentice-Hall, 1973.
- [12] S. Dasgupta, *Design Theory and Computer Science : Processes and Methodology of Computer Systems Design*. Cambridge Tracts in Theoretical Computer Science, Cambridge ; New York: Cambridge University Press, 1991.
- [13] M. M. Tanik and R. T. Yeh, “Rapid prototyping in software development,” *Computer*, vol. 22, no. 5, pp. 9–11, 1989.

- [14] S. N. Delcambre and M. M. Tanik, "Using task system templates to support process description and evolution," *Journal of Systems Integration*, vol. 8, no. 1, p. 83, 1998.
- [15] N. P. Suh, *Axiomatic Design : Advances and Applications*. The MIT-Pappalardo Series in Mechanical Engineering, New York: Oxford University Press, 2001.
- [16] N. Cross, *Engineering Design Methods: Strategies for Product Design*. Chichester, England ; Hoboken, NJ: J. Wiley, 4th ed., 2008.
- [17] M. M. Tanik and E. S. Chan, *Fundamentals of Computing for Software Engineers*. Van Nostrand Reinhold, 1991.
- [18] H. A. Simon, *The Sciences of the Artificial*. Cambridge, Massachusetts: MIT Press, 3rd ed., 1996.
- [19] G. F. Smith and G. J. Browne, "Conceptual foundations of design problem solving," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 5, pp. 1209–1219, 1993.
- [20] V. N. Gudivada and V. V. Raghavan, "Content-based image retrieval systems," *Computer*, vol. 28, no. 9, pp. 18–22, 1995.
- [21] *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.
- [22] G. Booch, *Object-Oriented Analysis and Design with Applications*. The Addison-Wesley Object Technology Series, Upper Saddle River, New Jersey: Addison-Wesley, 3rd ed., 2007.
- [23] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley Object Technology Series, Reading, Massachusetts: Addison-Wesley, 1999.
- [24] C. L. Simons, I. C. Parmee, and P. D. Coward, "35 years on: To what extent has software engineering design achieved its goals?," *IEE Proceedings Software*, vol. 150, pp. 337–350, 2003.
- [25] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Proceedings of the 9th International Conference on Software Engineering*, (41801), pp. 328–338, IEEE Computer Society Press, 1987.
- [26] R. L. Baber, "Comparison of electrical "engineering" of heaviside's times and software "engineering" of our times," *IEEE Annals of the History of Computing*, vol. 19, no. 4, p. 5, 1997.
- [27] S. R. Schach, *Object-Oriented Software Engineering*. Boston, Massachusetts: McGraw-Hill, 1st ed., 2008.

- [28] K. Yang and B. El-Haik, *Design for Six Sigma : A Roadmap for Product Development*. New York: McGraw-Hill, 2003.
- [29] J. Rothstein, "Organization and entropy," *Journal of Applied Physics*, vol. 23, no. 11, pp. 1281–1282, 1952.
- [30] C. Alexander, *Notes on the Synthesis of Form*. Cambridge, Massachusetts: Harvard University Press, 1964.
- [31] Y. Tuncer, M. M. Tanik, and D. B. Allison, "An overview of statistical decomposition techniques applied to complex systems," *Computational Statistics & Data Analysis*, vol. 52, no. 5, pp. 2292–2310, 2008.
- [32] J. Rothstein, *Communication, Organization, and Science*. Keystone Series, Indian Hills, Colorado: Falcon's Wing Press, 1958.
- [33] S. Watanabe, "Information theoretical analysis of multivariate correlation," *IBM Journal of Research and Development*, vol. 4, no. 1, pp. 66–82, 1960.
- [34] W. R. Ashby, "Measuring the internal informational exchange in a system," *Cybernetica*, vol. 1, no. 1, pp. 5–22, 1965.
- [35] M. H. v. Emden, *An Analysis of Complexity*. Mathematical Centre tracts, Amsterdam: Mathematisch Centrum, 1971.
- [36] S. Watanabe, *Knowing and Guessing; A Quantitative Study of Inference and Information*. New York: Wiley, 1969.
- [37] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [38] W. J. McGill, "Multivariate information transmission," *Psychometrika Psychometrika*, vol. 19, no. 2, pp. 97–116, 1954.
- [39] H. A. Simon, "The architecture of complexity," *Proceedings of the American Philosophical Society*, vol. 106, no. 6, pp. 467–482, 1962.
- [40] J. Rothstein, "Information, organization and systems," *Information Theory, IRE Professional Group on*, vol. 4, no. 4, pp. 64–66, 1954.
- [41] R. C. Conant, "Detecting subsystems of a complex system," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 2, no. 4, pp. 550–553, 1972.
- [42] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Boston, Massachusetts: Addison-Wesley, 1995.
- [43] R. C. Conant, *Information Transfer in Complex systems, with Applications to Regulation*. PhD thesis, 1968.

- [44] M. Aksit and B. Tekinerdogan, *Deriving Design Alternatives Based on Quality Factors Software Architectures and Component Technology*, vol. 648 of *The Kluwer International Series in Engineering and Computer Science*, pp. 225–257. Springer US, 2002.
- [45] R. E. Prather, “Design and analysis of hierarchical software metrics,” *ACM Computing Surveys*, vol. 27, no. 4, pp. 497–518, 1995.
- [46] E. Fermi, *Thermodynamics*. New York: Dover Publications, 1956.
- [47] H. C. Van Ness, *Understanding Thermodynamics*. New York,: McGraw-Hill, 1969.
- [48] A. Einstein and L. Infeld, *The Evolution of Physics: From Early Concepts to Relativity and Quanta*. A Touchstone Book, New York: Simon and Schuster, 1966.
- [49] S. M. Carroll, *From eternity to Here: The Quest for the Ultimate Theory of Time*. New York: Dutton, 2010.
- [50] J. Gleick, *The Information: A History, A Theory, A Flood*. New York: Pantheon Books, 1st ed., 2011.
- [51] J. C. Maxwell and W. D. Niven, *The Scientific Papers of James Clerk Maxwell*. New York: Dover Publications, 1965.
- [52] L. Boltzmann, *Lectures on Gas Theory*. Dover Books on Physics Series, Dover Publications, 1995.
- [53] J. W. Gibbs, *Elementary Principles in Statistical Mechanics, Developed with Especial Reference to the Rational Foundations of Thermodynamics*. Yale Bicentennial Publications, New York: C. Scribner’s Sons, 1902.
- [54] H. S. Leff and A. F. Rex, *Maxwell’s Demon 2: Entropy, Classical and Quantum Information, Computing*. Bristol ; Philadelphia: Institute of Physics Publishing, 2nd ed., 2003.
- [55] L. Szilard, “On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings,” *Systems Research and Behavioral Science*, vol. 9, no. 4, pp. 301–310, 1964.
- [56] R. Landauer, “Irreversibility and heat generation in the computing process,” *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 183–191, 1961.
- [57] H. Nyquist, “Certain factors affecting telegraph speed,” *Transactions of the American Institute of Electrical Engineers*, vol. XLIII, pp. 412–422, 1924.
- [58] R. V. L. Hartley, “Transmission of information,” *Bell System Technical Journal*, vol. 7, no. 3, pp. 535–563, 1928.

- [59] A. Renyi, *Probability Theory*. North-Holland series in Applied Mathematics and Mechanics, Amsterdam ; New York: North-Holland Pub. Co. ; American Elsevier Pub Co., 1970.
- [60] J. R. Pierce, *An Introduction to Information Theory: Symbols, Signals & Noise*. New York: Dover Publications, 2nd, rev. ed., 1980.
- [61] N. Abramson, *Information Theory and Coding*. McGraw-Hill Electronic Sciences Series, New York: McGraw-Hill, 1963.
- [62] P. M. Woodward, *Probability and Information Theory, with Applications to Radar*. International Series of Monographs on Electronics and Instrumentation, Oxford, New York: Pergamon Press, 2d ed., 1964.
- [63] R. B. Ash, *Information Theory*. New York: Dover Publications, dover ed., 1990.
- [64] S. Guiasu, *Information Theory with Applications*. New York: McGraw-Hill, 1977.
- [65] A. Papoulis and S. U. Pillai, *Probability, Random Variables, and Stochastic Processes*. Boston: McGraw-Hill, 4th ed., 2002.
- [66] R. M. Fano, *Transmission of Information; A Statistical Theory of Communications*. MIT Press Publications, Cambridge, Massachusetts: M.I.T. Press, 1963.
- [67] F. M. Reza, *An Introduction to Information Theory*. New York: Dover, 1994.
- [68] G. A. Miller and F. C. Frick, "Statistical behavioristics and sequences of responses," *Psychological Review*, vol. 56, no. 6, pp. 311–24, 1949.
- [69] F. Attneave, *Applications of Information Theory to Psychology; A Summary of Basic concepts, Methods, and Results*. New York: Holt, 1959.
- [70] W. R. Garner, *Uncertainty and Structure as Psychological Concepts*. New York: Wiley, 1962.
- [71] W. R. Garner and H. W. Hake, "The amount of information in absolute judgements," *Psychological Review*, vol. 58, no. 6, pp. 446–59, 1951.
- [72] W. R. Garner and W. J. McGill, "The relation between information and variance analyses," *Psychometrika Psychometrika*, vol. 21, no. 3, pp. 219–228, 1956.
- [73] H. Quastler, *Information Theory in Psychology; Problems and Methods. Proceedings of a Conference on the Estimation of Information Flow, Monticello, Illinois, July 5-9, 1954, and Related Papers*. Glencoe, Illinois: Free Press, 1955.
- [74] A. N. Kolmogorov, *Foundations of the Theory of Probability*. New York: Chelsea Pub. Co., 1950.
- [75] B. V. Gnedenko and A. I. A. Khinchin, *An Elementary Introduction to the Theory of Probability*. Series of Undergraduate Books in Mathematics, San Francisco: W. H. Freeman, 1961.

- [76] W. Feller, *An Introduction to Probability Theory and its Applications*. Wiley series in Probability and Mathematical Statistics, New York: Wiley, 3d ed., 1968.
- [77] I. U. A. Rozanov and R. A. Silverman, *Probability Theory: A Concise Course*. New York: Dover Publications, rev. english ed., 1977.
- [78] B. V. Gnedenko, *The Theory of Probability*. New York: Chelsea Pub. Co., 4th ed., 1967.
- [79] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Hoboken, New Jersey: Wiley-Interscience, 2nd ed., 2006.
- [80] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press, 2010.
- [81] G. A. Miller, "What is information measurement?," *American Psychologist* *American Psychologist*, vol. 8, no. 1, pp. 3–11, 1953.
- [82] S. Kullback, *Information Theory and Statistics*. Wiley Publication in Mathematical Statistics, New York: Wiley, 1959.
- [83] F. Hausdorff, *Set Theory*. New York: Chelsea Pub. Co., 2d ed., 1962.
- [84] K. Krippendorff, *Information Theory: Structural Models for Qualitative Data*. Quantitative Applications in the Social Sciences, Beverly Hills, California: Sage, 1986.
- [85] S. Watanabe, "A note on the formation of concept and of association by information-theoretical correlation analysis," *Information and Control Information and Control*, vol. 4, no. 2-3, pp. 291–296, 1961.
- [86] J. Rothstein, "Information and organization as the language of the operational viewpoint," *Philosophy of Science*, vol. 29, no. 4, pp. 406–411, 1962.
- [87] W. R. Ashby, *Information Flows within Co-ordinated Systems*. Urbana, Illinois: Biological Computer Laboratory, Univ. of Illinois at Urbana-Champaign, 1969.
- [88] W. Ashby, "Two tables of identities governing information flows within large systems," *ASC Communications*, vol. 1, no. 2, pp. 3–8, 1969.
- [89] W. R. Ashby and R. Conant, *Mechanisms of Intelligence : Ashby's Writings on Cybernetics*. Seaside, California: Intersystems Publications, 1981.
- [90] W. R. Ashby, "Principles of the self-organizing dynamic system," *The Journal of General Psychology*, vol. 37, no. 2, pp. 125–128, 1947.
- [91] L. v. Bertalanffy, *General System Theory; Foundations, Development, Applications*. New York: G. Braziller, rev. ed., 1969.

- [92] G. J. Klir and E. C. Way, "Reconstructability analysis: Aims, results, open problems," *Systems Research*, vol. 2, no. 2, pp. 141–163, 1985.
- [93] G. J. Klir, "On the representation of activity arrays," *International Journal of General Systems*, vol. 2, no. 3, pp. 149–168, 1975.
- [94] G. Broekstra, "On the representation and identification of structure systems," *International Journal of Systems Science*, vol. 9, no. 11, pp. 1271–1293, 1978.
- [95] R. C. Conant, "Structural modelling using a simple information measure," *International Journal of Systems Science*, vol. 11, no. 6, pp. 721–730, 1980.
- [96] R. C. Conant, "Detection and analysis of dependency structures," *International Journal of General Systems*, vol. 7, no. 1, pp. 81–91, 1981.
- [97] B. R. Gaines, "System identification, approximation and complexity," *International Journal of General Systems*, vol. 3, no. 3, pp. 145–174, 1977.
- [98] R. Cavallo and J. De Voy, "Iterative and recursive algorithms for tree search and partition search of the lattice of structure models," *International Journal of General Systems*, vol. 20, no. 3, pp. 275–301, 1992.
- [99] G. J. Klir, *Applied General Systems Research : Recent Developments and Trends*. NATO conference series II, Systems Science, New York: Plenum Press, 1978.
- [100] H. A. Simon, *Models of Man: Social and Rational; Mathematical Essays on Rational Human Behavior in Society Setting*. New York: Wiley, 1957.
- [101] G. A. Miller, "The magical number seven plus or minus two: Some limits on our capacity for processing information," *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.
- [102] B. D. Jones, "Bounded rationality," *Annual Review of Political Science*, vol. 2, no. 1, pp. 297–321, 1999.
- [103] J. S. Bruner, J. J. Goodnow, and G. A. Austin, *A Study of Thinking*. New York: Science Editions, 1962.
- [104] D. Hamlet and J. Maybee, *The Engineering of Software: Technical Foundations for the Individual*. Boston: Addison-Wesley, 2001.
- [105] G. Polya, *How to Solve it; A New Aspect of Mathematical Method*. Princeton, New Jersey: Princeton University Press, 1945.
- [106] R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., 2010.
- [107] D. Budgen, *Software Design*. Addison-Wesley Longman Publishing Co., Inc., 2003.

- [108] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Upper Saddle River, New Jersey: Prentice Hall, 2nd ed., 2003.
- [109] I. Sommerville, *Software Engineering*. International computer science series, Boston: Pearson/Addison-Wesley, 7th ed., 2004.
- [110] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series, Cambridge, Massachusetts ;Â New York: MIT Press ; McGraw-Hill, 1986.
- [111] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Not.*, vol. 9, no. 4, pp. 50–59, 1974.
- [112] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*. New York: Wiley, 3rd ed., 1998.
- [113] B. I. Blum, *Software Engineering: A Holistic View*. Oxford University Press, Inc., 1992.
- [114] M. Shaw, "Prospects for an engineering discipline of software," *IEEE Software*, vol. 7, no. 6, pp. 15–24, 1990.
- [115] B. Meyer, *Object-Oriented Software Construction*. Upper Saddle River, New Jersey: Prentice Hall, 2nd ed., 1997.
- [116] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems*, vol. 13, no. 2, pp. 115–139, 1974.
- [117] J. R. Cameron, "An overview of jsd," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 222–240, 1986.
- [118] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. New York: Yourdon Press, 2nd ed., 1978.
- [119] G. J. Myers, *Reliable Software Through Composite Design*. New York,: Petrocelli/Charter, 1st ed., 1975.
- [120] J. Rumbaugh, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.
- [121] B. I. Blum, *Beyond Programming: To a New Era of Design*. Oxford University Press, Inc., 1996.
- [122] R. Wieringa, "A survey of structured and object-oriented software specification methods and techniques," *ACM Computing Survey*, vol. 30, no. 4, pp. 459–527, 1998.

- [123] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. New York Wokingham, England ; Reading, Massachusetts: ACM Press ; Addison-Wesley Pub., 1992.
- [124] G. Kiczales, “Aspect oriented programming,” *ACM SIGPLAN notices : a monthly publication of the Special Interest Group on Programming Languages.*, vol. 32, no. 10, p. 162, 1997.
- [125] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher, “Discussing aspects of aop,” *Communications of the ACM*, vol. 44, no. 10, pp. 33–38, 2001.
- [126] L. Bergmans and M. Aksit, “Composing crosscutting concerns using composition filters,” *Communications of the ACM*, vol. 44, no. 10, pp. 51–57, 2001.
- [127] D. Cooke, A. Gates, E. Demirors, O. Demirors, M. M. Tanik, and B. Kramer, “Languages for the specification of software,” *Journal of Systems and Software*, vol. 32, no. 3, pp. 269–308, 1996.
- [128] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1989.
- [129] R. M. Burstall and J. A. Goguen, “The semantics of clear, a specification language,” in *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, Springer-Verlag, 1980.
- [130] C. A. R. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [131] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [132] H. R. Nielson and F. Nielson, *Semantics with Applications. A Formal Introduction*. John Wiley & Sons, 1992.
- [133] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [134] G. D. Plotkin, “The origins of structural operational semantics,” *Journal of Logic and Algebraic Programming*, vol. 60, pp. 3–15, 2004.
- [135] D. E. Knuth, “Semantics of context-free languages,” *Theory of Computing Systems*, vol. 2, no. 2, pp. 127–145, 1968.
- [136] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer, “Lisa: An interactive environment for programming language development,” in *CC02 Proceedings of the 11th International Conference on Compiler Construction*, vol. 2304 of *Lecture notes in computer science*, pp. 1–4, Springer-Verlag, 2002.
- [137] R. Grzegorz, *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

- [138] J. de Lara and H. Vangheluwe, “Defining visual notations and their manipulation through meta-modelling and graph transformation,” *Journal of Visual Languages & Computing*, vol. 15, no. 3-4, pp. 309–330, 2004.
- [139] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, “Analyzing rule-based behavioral semantics of visual modeling languages with maude,” vol. 5452 of *Lecture Notes in Computer Science*, pp. 54–73, Springer-Verlag, 2008.
- [140] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley object technology series, Reading, Massachusetts: Addison-Wesley, 1999.
- [141] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston: Addison-Wesley, 3rd ed., 2004.
- [142] S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*. Prentice-Hall Software Series, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
- [143] O. Aktunc, *An Entropy-based Measurement Framework for Component-based Hierarchical Systems*. PhD thesis, 2007.
- [144] C. V. Ramamoorthy, “Analysis of graphs by connectivity considerations,” *Journal of the ACM*, vol. 13, no. 2, pp. 211–222, 1966.
- [145] R. E. Prather, “On hierarchical software metrics,” *Software Engineering Journal*, vol. 2, no. 2, pp. 42–45, 1987.
- [146] Z. Demirezen, B. R. Bryant, A. Skjellum, and M. M. Tanik, “Design space analysis in model-driven engineering,” *Journal of Integrated Design and Process Science*, vol. 14, no. 1, pp. 1–15, 2010.
- [147] R. E. Prather, “Hierarchical metrics and the prime generation problem,” *Software Engineering Journal*, vol. 8, no. 5, pp. 246–252, 1993.
- [148] A. P. Sage and J. D. Palmer, *Software Systems Engineering*. Wiley-Interscience, 1990.
- [149] M. Page-Jones, *The Practical Guide to Structured Systems Design*. Yourdon Press Computing series, Englewood Cliffs, New Jersey: Prentice Hall, 2nd ed., 1988.
- [150] M. Page-Jones and L. L. Constantine, *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley Object Technology Series, New York; Reading, Massachusetts: Dorset House Pub. ; Addison-Wesley, 2000.
- [151] A. M. Davis, “Fifteen principles of software engineering,” *IEEE Software*, vol. 11, no. 6, pp. 94–96, 1994.
- [152] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

- [153] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting started with aspectj,” *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.
- [154] D. C. Schmidt, “Guest editor’s introduction: Model-driven engineering,” *Computer*, vol. 39, no. 2, pp. 25–31, 2006.
- [155] J. Sprinkle, M. Mernik, J.-P. Tolvanen, and D. Spinellis, “Guest editors’ introduction: What kinds of nails need a domain-specific hammer?,” *IEEE Software*, vol. 26, no. 4, pp. 15–18, 2009.
- [156] B. R. Bryant, J. Gray, and M. Mernik, “Domain-specific software engineering,” in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, (1882376), pp. 65–68, ACM, 2010.
- [157] Z. Demirezen, M. Mernik, J. Gray, and B. Bryant, “Verification of dsmls using graph transformation: A case study with alloy,” in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, (1656488), pp. 1–10, ACM, 2009.
- [158] I. Kurtev, J. Bezivin, F. Jouault, and P. Valduriez, “Model-based dsl frameworks,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ACM, 2006.
- [159] F. Marcelloni and M. Aksit, “Improving object-oriented methods by using fuzzy logic,” *SIGAPP Applied Computing Review*, vol. 8, no. 2, pp. 14–23, 2000.
- [160] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.
- [161] B. Corrado and J. Giuseppe, “Flow diagrams, turing machines and languages with only two formation rules,” *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.
- [162] E. Syriani and H. Vangheluwe, “Programmed graph rewriting with time for simulation-based design,” vol. 5063 of *Lecture Notes in Computer Science*, pp. 91–106, Springer-Verlag, 2008.
- [163] R. T. Yeh, *Applied Computation Theory: Analysis, Design, Modeling*. Prentice-Hall Series in Automatic Computation, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.
- [164] <http://www.sdpsnet.org>.
- [165] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series, Cambridge, Massachusetts ; New York: MIT Press; McGraw-Hill, 1990.

APPENDIX A

JAVA IMPLEMENTATION

Table A.1: List of Java Files

Java File	Explanation
<i>BasicInterface.java</i>	provides an interface for the collection operations, such as <i>getElements</i> , <i>jointElements</i> .
<i>BasicEntropyInterface.java</i>	provides an interface for the entropy functions, such as <i>Entropy</i> , <i>Transmission</i> and <i>Correlation</i> .
<i>BasicEntropyImplementation.java</i>	provides an implementation of <i>BasicEntropyInterface</i> such as implementation of Shannon's formula and McGill's multivariate formula.
<i>UncertainElement.java</i>	provides an implementation of <i>BasicInterface</i> and provides basic collection operations. Decomposition algorithms based on Conant's and Watanabe's implementations are provided in this java file.
<i>Variable.java</i>	is a concrete type of <i>UncertainElement</i> which is used for the representation of set variables.
<i>Set.java</i>	is a collection of <i>UncertainElements</i> to represent a set of <i>Variables</i> , a set of <i>Sets</i> . Decomposition algorithm based on Alexander's implementation is provided in this java file.
<i>FrequencyTable.java</i>	provides the database operations and instantiates <i>Sets</i> and <i>Variables</i> .

BasicInterface.java

```
package multivariateanalysis;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

public interface BasicInterface {

    public Iterator<UncertainElement> getElementsIterator();
    public List<UncertainElement> getElements();
    public UncertainElement getElement(int i);
    public int size();
    public void addUncertainElement(UncertainElement var);
    public boolean removeUncertainElement(UncertainElement s);
    public void addAllUncertainElement(Collection<UncertainElement> puncertainElements);
    public String getName();
    public String getViewName();
    public UncertainElement jointElements(UncertainElement part2);
    public ArrayList<UncertainElement> [] powerSet();
}
```

BasicEntropyInterface.java

```
package multivariateanalysis;

public interface BasicEntropyInterface extends BasicInterface{

    //ENTROPY*****

    //H(X) Shannon measure of Information
    //Measure of Uncertainty
    public double entropy();

    //H(Y|X)=Hx(Y)=H(X,Y)-H(X)
    public double conditionalEntropy(UncertainElement prior);

    //Hmax(X)
    public double maxEntropy();

    //H(x)/Hmax(x)
    //Standardized
    public double normalizedEntropy();

    //1-H(x)/Hmax(x)
    //C-Function
    public double redundancy();

    //TRANSMISSION*****

    // T(Input:Output)=H(X)+H(Y)-H(X,Y)
    //Measure of Relatedness
    public double transmission(UncertainElement part2);

    //Tmax=Min(H(X),H(Y))
    public double maxTransmission(UncertainElement part2);

    //T(X:Y)/Tmax(X:Y)
    //Index of Predictability Krippendorff pp. 24
    public double normalizedTransmission(UncertainElement part2);

    //R=Tmax-T (not standardized)
    public double redundancyinTransmission(UncertainElement part2);

    //T(Y:Z|X)=Tx(Y:Z)=H(Y|X)+H(Z|X)-H(Y,Z|X)
    public double conditionalTransmission(UncertainElement part2, UncertainElement prior);

    //the D function D(X:Y)=T(X:Y)/H(Y)= 1- H(Y|X)/H(Y)
    //Coefficient of Constraint
    //Relative Amount of Relatedness
    //Not Symmetrical
    public double relativeTransmission(UncertainElement part2);

    //D(X:Y|Z)=T(X:Y|Z)/H(Y|Z)
    public double relativeConditionalTransmission(UncertainElement part2, UncertainElement prior);

    //CORRELATION*****
```

BasicEntropyInterface.java

```
// C(XYZ....) or T(X:Y:Z;...)
//Measure of Relatedness
public double correlation();

//Cx(WYZ)  C(WYZ|X)
public double conditionalCorrelation(UncertainElement prior);

//INTERACTION*****

// Q(xyz)  or A(x:y:z)
//Measure of Partial Relatedness
public double interaction();
public double conditionalInteraction(UncertainElement prior);

}
```

BasicEntropyImplementation.java

```
package multivariateanalysis;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Iterator;

public abstract class BasicEntropyImplementation implements
    BasicEntropyInterface {

    static Hashtable<String, Double> hashEntropy = new Hashtable<>();
    public static String condition = "";

    public static void clearCache() {
        hashEntropy = new Hashtable<>();
    }

    public static double getLogTwo(double inputValue) {
        if (inputValue > 0) {
            return Math.Log(inputValue) / Math.Log(2);
        } else {
            return 0.0;
        }
    }

    // ENTROPY*****
    // H(X) Shannon-Wiener measure of Information

    public double entropy() {
        double d = 0;
        ArrayList<Integer> list = FrequencyTable.instance.getFrequency(
            getViewName(), getName(), condition);
        double total = FrequencyTable.instance.getTotal(getViewName(),
            condition);
        for (int i = 0; i < list.size(); i++) {
            Integer freq = list.get(i);
            d += -1 * (((double) freq) / total)
                * getLogTwo(((double) freq) / total);
        }
        DecimalFormat f = new DecimalFormat("#.##");
        return new Double(f.format(d));
    }

    //  $H(Y|X)=H_X(Y)=H(X,Y)-H(X)$ 
    public double conditionalEntropy(UncertainElement prior) {
        return jointElements(prior).entropy() - prior.entropy();
    }

    // Hmax(X)
    public double maxEntropy() {
        int itemCount = FrequencyTable.instance.getItemCount(
            getViewName(), getName());
        return getLogTwo(itemCount);
    }
}
```

BasicEntropyImplementation.java

```
//
public double normalizedEntropy() {
    return entropy() / maxEntropy();
}

//
public double redundancy() {
    return 1 - normalizedEntropy();
}

// TRANSMISSION*****

// T(Input:Output)
public double transmission(UncertainElement part2) {
    return this.entropy() + part2.entropy()
        - jointElements(part2).entropy();
}

// Tmax=Min(H(X),H(Y))
public double maxTransmission(UncertainElement part2) {
    double inEntropy = this.entropy();
    double outEntropy = part2.entropy();
    return inEntropy < outEntropy ? inEntropy : outEntropy;
}

// T(X:Y)/Tmax(X:Y)
// Index of Predictability Krippendorff pp. 24
public double normalizedTransmission(UncertainElement part2) {
    return transmission(part2) / maxTransmission(part2);
}

// R=Tmax-T (not standardized)
public double redundancyinTransmission(UncertainElement part2) {
    return 1 - normalizedTransmission(part2);
}

// T(Y:Z|X)=Tx(Y:Z)=H(Y|X)+H(Z|X)-H(Y,Z|X)
public double conditionalTransmission(UncertainElement part2,
    UncertainElement prior) {
    return this.conditionalEntropy(prior) + part2.conditionalEntropy(prior)
        - jointElements(part2).conditionalEntropy(prior);
}

// the D function D(X:Y) McGill Quastler pp. 89
public double relativeTransmission(UncertainElement part2) {
    return this.transmission(part2) / part2.entropy();
}

// D(X:Y|Z)=T(X:Y|Z)/H(Y|Z)
public double relativeConditionalTransmission(UncertainElement part2,
    UncertainElement prior) {
    return this.conditionalTransmission(part2, prior)
        / part2.conditionalEntropy(prior);
}

// CORRELATION*****
```

BasicEntropyImplementation.java

```
// C(xyz....)
public double correlation() {
    double result = 0;
    if (this.size() == 1)
        return 0;
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        Double d = hashEntropy.get(type.getName());
        if (d == null) {
            d = type.entropy();
            hashEntropy.put(type.getName(), d);
        }
        result += d;
    }
    Double d = hashEntropy.get(this.getName());
    if (d == null) {
        d = this.entropy();
        hashEntropy.put(this.getName(), d);
    }
    result -= d;
    return result;
}

public double conditionalCorrelation(UncertainElement prior) {
    double result = 0;
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        result += type.conditionalEntropy(prior);
    }
    result -= this.conditionalEntropy(prior);
    return result;
}

// INTERACTION*****

// Q(xyz)

public double interaction() {
    double d = 0;
    ArrayList<UncertainElement>[] subsets = this.powerSet();
    for (int i = 0; i < subsets.length; i++) {
        ArrayList<UncertainElement> arrayList = subsets[i];
        double totalEntropy = 0;
        for (Iterator iterator = arrayList.iterator(); iterator.hasNext();) {
            UncertainElement uncertainElement = (UncertainElement) iterator
                .next();
            totalEntropy += uncertainElement.entropy();
        }
        if ((this.size() - (i + 1)) % 2 == 0)
            d -= totalEntropy;
        else
            d += totalEntropy;
    }
}
```

BasicEntropyImplementation.java

```
        return d;
    }

    public double conditionalInteraction(UncertainElement prior) {
        double d = 0;
        ArrayList<UncertainElement>[] subsets = this.powerSet();
        for (int i = 0; i < subsets.length; i++) {
            ArrayList<UncertainElement> arrayList = subsets[i];
            double totalEntropy = 0;
            for (Iterator iterator = arrayList.iterator(); iterator.hasNext();) {
                UncertainElement uncertainElement = (UncertainElement) iterator
                    .next();
                totalEntropy += uncertainElement.conditionalEntropy(prior);
            }
            if (this.size() - (i + 1) % 2 == 0)
                d -= totalEntropy;
            else
                d += totalEntropy;
        }
        return d;
    }
}
```

UncertainElement.java

```

package multivariateanalysis;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Hashtable;
import java.util.Iterator;

public abstract class UncertainElement extends BasicEntropyImplementation {
    protected String viewName = FrequencyTable.VIEWNAME;

    public static final int CONANT = 0;
    public static final int WATANABE = 1;

    public abstract UncertainElement getTimeIncrementElement(int tick);

    public String getViewName() {
        return viewName;
    }

    public UncertainElement jointElements(UncertainElement part2) {
        Set result = new Set();
        result.addAllUncertainElement(getElements());
        result.addAllUncertainElement(part2.getElements());
        return result;
    }

    public String getName() {
        String str = "";
        for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
            .hasNext();) {
            UncertainElement type = iterator.next();
            str += type.getName() + ",";
        }
        str = str.substring(0, str.length() - 1);
        return str;
    }

    // CALCULATIONS*****
    // CONANT Dissertation
    // pp 552 tij
    public double calculateNormalizedTransmission_OneTimeIncrementLater(
        UncertainElement element) {
        return this.transmission(element.getTimeIncrementElement(1))
            / element.getTimeIncrementElement(1).entropy();
    }

    // CONANT Dissertation
    // pp 552 Twj
    public double calculateConstraintHoldingOverOneTimeIncrementWithInSubsystem() {
        Set s = new Set();
        s.addUncertainElement(this);
        s.addUncertainElement(this.getTimeIncrementElement(1));
        return s.correlation();
    }
}

```


UncertainElement.java

```
// CONANT Dissertation
// pp 552 Tbij
public double calculateTheStrenghtofRelationBetween(Set element) {
    Set s1 = new Set();
    s1.addUncertainElement(this);
    s1.addUncertainElement(this.getTimeIncrementElement(1));

    Set s2 = new Set();
    s2.addUncertainElement(element);
    s2.addUncertainElement(element.getTimeIncrementElement(1));

    return s1.transmission(s2);
}

// CONANT Dissertation
// pp 552 Tb
public double calculateTheConstraintBetweenAll_OverOneTimeIncrement() {
    Set result = new Set();
    for (Iterator<UncertainElement> iterator = this.getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        Set subSet = new Set();
        subSet.addUncertainElement(type);
        subSet.addUncertainElement(type.getTimeIncrementElement(1));
        result.addUncertainElement(subSet);
    }
    return result.correlation();
}

// CONANT Dissertation
// pp 552 Tw1+Tw2
public double calculateTOTALConstraintHoldingOverOneTimeIncrementWithInSubsystem() {
    double d = 0;
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        d += type
            .calculateConstraintHoldingOverOneTimeIncrementWithInSubsystem();
    }
    return d;
}

// CONANT Dissertation
// pp 552 Tb<<(Tw1+Tw2)
// Tb/(Tb+Tw1+Tw2)
public double verify() {
    return (calculateTheConstraintBetweenAll_OverOneTimeIncrement() /
        (calculateTOTALConstraintHoldingOverOneTimeIncrementWithInSubsystem()
            + calculateTheConstraintBetweenAll_OverOneTimeIncrement())) * 100;
}

//
*****

// DECOMPOSITION
// *****
```

UncertainElement.java

```

public static class PairWiseInteraction implements
    Comparable<PairWiseInteraction> {
    Double value;
    UncertainElement[] elements = new UncertainElement[2];
    boolean equal;

    PairWiseInteraction(Double d, UncertainElement[] pelements,
        boolean pequal) {
        value = d;
        elements = pelements;
        equal = pequal;
    }

    @Override
    public int compareTo(PairWiseInteraction o) {
        // TODO Auto-generated method stub
        return -1 * value.compareTo(o.value);
    }
}

public static ArrayList<PairWiseInteraction> getPairwiseInteractionMatrix(
    Collection<UncertainElement> elements, int decType) {

    ArrayList<PairWiseInteraction> pairwiseInteractionMatrix =
        new ArrayList<PairWiseInteraction>();

    int x = 0;
    for (Iterator<UncertainElement> iterator = elements.iterator(); iterator
        .hasNext();) {
        UncertainElement uncertainElement = iterator.next();
        x++;
        int y = 0;
        for (Iterator<UncertainElement> iterator2 = elements.iterator(); iterator2
            .hasNext();) {
            UncertainElement uncertainElement2 = iterator2.next();
            y++;

            if (x != y) {
                UncertainElement[] row = new UncertainElement[2];
                row[0] = uncertainElement;
                row[1] = uncertainElement2;

                if (decType == CONANT)
                    pairwiseInteractionMatrix
                        .add(
                            new PairWiseInteraction(
                                uncertainElement.
                                    calculateNormalizedTransmission_OneTimeIncrementLater(uncertainElement2),
                                    row, x == y));
                else if (decType == WATANABE)
                    pairwiseInteractionMatrix.add(new PairWiseInteraction(
                        uncertainElement
                            .transmission(uncertainElement2), row,
                            x == y));
            }
        }
    }
}

```

UncertainElement.java

```
}
Collections.sort(pairwiseInteractionMatrix);
return pairwiseInteractionMatrix;
}

public void printMatrix(
    ArrayList<PairWiseInteraction> pairwiseInteractionMatrix) {
    for (Iterator<PairWiseInteraction> iterator = pairwiseInteractionMatrix
        .iterator(); iterator.hasNext();) {
        PairWiseInteraction type = iterator.next();
        System.out.println(type.elements[0] + "--" + type.elements[1] + "*"
            + type.value);
    }
}

public UncertainElement decompose(int decType) {

    if (this.size() == 1)
        return this;
    Hashtable<UncertainElement, UncertainElement> subSystemTracking = new Hashtable<>();
    ArrayList<PairWiseInteraction> pairwiseInteractionMatrix = getPairwiseInteractionMatrix(
        this.getElements(), decType);
    printMatrix(pairwiseInteractionMatrix);

    UncertainElement result = new Set();
    int variableCount = size();
    for (Iterator<PairWiseInteraction> iterator = pairwiseInteractionMatrix
        .iterator(); iterator.hasNext();) {
        PairWiseInteraction pairInteract = iterator.next();

        UncertainElement sub1 = subSystemTracking
            .get(pairInteract.elements[0]);
        UncertainElement sub2 = subSystemTracking
            .get(pairInteract.elements[1]);

        if (sub1 == null && sub2 == null) {
            UncertainElement subNew = new Set();
            variableCount = variableCount - 1;
            subNew.addUncertainElement(pairInteract.elements[0]);
            subSystemTracking.put(pairInteract.elements[0], subNew);
            result.addUncertainElement(subNew);
            if (!pairInteract.equal) {
                subNew.addUncertainElement(pairInteract.elements[1]);
                subSystemTracking.put(pairInteract.elements[1], subNew);
                variableCount = variableCount - 1;
            }
        }
        else if (sub1 == null) {
            variableCount--;
            sub2.addUncertainElement(pairInteract.elements[0]);
            subSystemTracking.put(pairInteract.elements[0], sub2);
            subSystemTracking.put(pairInteract.elements[1], sub2);
        }
        else if (sub2 == null) {
            variableCount--;
            sub1.addUncertainElement(pairInteract.elements[1]);
            subSystemTracking.put(pairInteract.elements[1], sub1);
            subSystemTracking.put(pairInteract.elements[0], sub1);
        }
    }
}
```

UncertainElement.java

```

    } else {
        if (result.removeUncertainElement(sub2)) {
            if (result.removeUncertainElement(sub1)) {
                sub1 = sub1.joinElements(sub2);
                result.addUncertainElement(sub1);
                for (Iterator<UncertainElement> iterator2 = sub1
                    .getElementsIterator(); iterator2.hasNext();) {
                    UncertainElement type = iterator2.next();
                    subSystemTracking.put(type, sub1);
                }
            } else {
                result.addUncertainElement(sub2);
            }
        }

        if (variableCount == 0) {
            break;
        }
        // }
    }
    return result;
}

//
*****

public String toString() {
    String str = "";
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        str += type;
    }
    return str;
}

@Override
public boolean equals(Object obj) {
    // TODO Auto-generated method stub
    return this == obj;
}
}

```

Variable.java

```
package multivariateanalysis;

import java.util.*;

public class Variable extends UncertainElement {
    private String name;

    public Variable(String nm) {
        name = nm;
    }

    public String getName() {
        return name;
    }

    public UncertainElement getTimeIncrementElement(int tick) {
        return new Variable(name + tick);
    }

    public String toString() {
        return " " + getName() + " ";
    }

    public Iterator<UncertainElement> getElementsIterator() {
        ArrayList<UncertainElement> elements = new ArrayList<UncertainElement>();
        elements.add(this);
        return elements.iterator();
    }

    public List<UncertainElement> getElements() {
        ArrayList<UncertainElement> elements = new ArrayList<UncertainElement>();
        elements.add(this);
        return elements;
    }

    public int size() {
        return 1;
    }

    public void addUncertainElement(UncertainElement var) {
    }

    public boolean removeUncertainElement(UncertainElement s) {
        return false;
    }

    public void addAllUncertainElement(
        Collection<UncertainElement> puncertainElements) {
    }

    public UncertainElement getElement(int i) {
        return null;
    }

    public ArrayList<UncertainElement>[] powerSet() {
        return new ArrayList[0];
    }}
}
```

Set.java

```

package multivariateanalysis;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

public class Set extends UncertainElement {

    private ArrayList<UncertainElement> uncertainElements = new ArrayList<UncertainElement>();

    public Set() {
        // TODO Auto-generated constructor stub
    }

    public Set(ArrayList<UncertainElement> elems) {
        uncertainElements = elems;
    }

    public Set(UncertainElement elem) {
        uncertainElements.add(elem);
    }

    public static void comb1(Set prefix, Set s) {
        if (s.size() > 0) {
            Set nSet = new Set(prefix.uncertainElements);
            nSet.addUncertainElement(s.getElement(0));
            System.out.println(nSet);
            Set kSet = new Set(s.uncertainElements);
            kSet.uncertainElements.remove(0);
            comb1(nSet, kSet);
            comb1(prefix, kSet);
        }
    }

    public static void comb2(ArrayList<UncertainElement>[] subsets, Set prefix,
        Set s) {
        if (prefix.size() != 0) {
            if (subsets[prefix.size() - 1] == null)
                subsets[prefix.size() - 1] = new ArrayList<>();
            subsets[prefix.size() - 1].add(prefix);
        }

        for (int i = 0; i < s.size(); i++) {
            Set s1 = new Set();
            s1.addAllUncertainElement(prefix.uncertainElements);
            s1.addUncertainElement(s.getElement(i));
            Set s2 = new Set();
            s2.addAllUncertainElement(s.uncertainElements.subList(i + 1,
                s.uncertainElements.size()));
            comb2(subsets, s1, s2);
        }
    }

    public ArrayList<UncertainElement>[] powerSet() {
        ArrayList<UncertainElement>[] subsets = new ArrayList[size()];
    }
}

```

Set.java

```
Set lattice = new Set();
Set.comb2(subsets, lattice, this);
return subsets;
}

private static ArrayList<Set> getSubsets(ArrayList<UncertainElement> set) {
    ArrayList<Set> subsetCollection = new ArrayList<Set>();

    if (set.size() == 0) {
        subsetCollection.add(new Set());
    } else {
        ArrayList<UncertainElement> reducedSet = new ArrayList<UncertainElement>();

        reducedSet.addAll(set);

        UncertainElement first = reducedSet.remove(0);
        ArrayList<Set> subsets = getSubsets(reducedSet);
        subsetCollection.addAll(subsets);

        subsets = getSubsets(reducedSet);

        for (Set subset : subsets) {
            subset.addUncertainElement(first);
        }

        subsetCollection.addAll(subsets);
    }

    return subsetCollection;
}

public void addAllUncertainElement(
    Collection<UncertainElement> puncertainElements) {
    uncertainElements.addAll(puncertainElements);
}

public Iterator<UncertainElement> getElementsIterator() {
    return uncertainElements.iterator();
}

public boolean removeUncertainElement(UncertainElement s) {
    return uncertainElements.remove(s);
}

public int size() {
    return uncertainElements.size();
}

public void addUncertainElement(UncertainElement var) {
    boolean b = uncertainElements.add(var);
    if (!b) {
        System.out.println(var);
        System.out.println(uncertainElements);
    }
}
```

Set.java

```

public Set unionElements() {
    Set result = new Set();
    String str = "";

    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        str += type.toString().trim();
    }
    String lViewName = "(";
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        lViewName += " SELECT " + type + " as " + str + " FROM "
            + FrequencyTable.TABLENAME
            + (iterator.hasNext() ? " union all" : "");
    }

    lViewName += ") as unionof" + FrequencyTable.TABLENAME;
    Variable var = new Variable(str);
    result.addUncertainElement(var);
    result.viewName = lViewName;
    return result;
}

@Override
public UncertainElement getTimeIncrementElement(int tick) {
    Set result = new Set();
    for (Iterator<UncertainElement> iterator = this.uncertainElements
        .iterator(); iterator.hasNext();) {
        UncertainElement type = iterator.next();
        result.uncertainElements.add(type.getTimeIncrementElement(tick));
    }
    return result;
}

@Override
public List<UncertainElement> getElements() {
    // TODO Auto-generated method stub
    return uncertainElements;
}

public String toString() {
    String str = " ( ";
    for (Iterator<UncertainElement> iterator = getElementsIterator(); iterator
        .hasNext();) {
        UncertainElement type = iterator.next();
        str += type;
    }
    str += " )";
    return str;
}

@Override
public UncertainElement getElement(int i) {
    // TODO Auto-generated method stub

```


Set.java

```
        return this.uncertainElements.get(i);
    }

    // LATTICE OPERATIONS*****

    public Set createMaxSet() {
        Set s = new Set();
        for (Iterator iterator = uncertainElements.iterator(); iterator
            .hasNext();) {
            UncertainElement uncertainElement = (UncertainElement) iterator
                .next();
            Set sub = new Set();
            sub.addAllUncertainElement(uncertainElement.getElements());
            s.addUncertainElement(sub);
        }
        return s;
    }

    // *****

    // **
    public ArrayList<Set> generateNeighbours() {

        ArrayList<Set> result = new ArrayList<>();
        for (int i = 0; i < this.size(); i++) {
            for (int j = i + 1; j < this.size(); j++) {
                UncertainElement newSubItem = this.getElement(i).jointElements(
                    this.getElement(j));
                ArrayList<UncertainElement> items = new ArrayList<>(this.size());
                items.addAll(this.getElements());
                items.remove(this.getElement(i));
                items.remove(this.getElement(j));
                Set s = new Set();
                s.addUncertainElement(newSubItem);
                s.addAllUncertainElement(items);
                result.add(s);
            }
        }

        return result;
    }

    public void printInternal() {
        Iterator<UncertainElement> it = this.getElementsIterator();
        double totalCor = 0;
        DecimalFormat f = new DecimalFormat("#.##");

        while (it.hasNext()) {
            UncertainElement uncertainElement2 = (UncertainElement) it.next();
            System.out.print(f.format(uncertainElement2.correlation()) + "+");
            totalCor += uncertainElement2.correlation();
        }
        System.out.println("=" + f.format(totalCor));
        System.out.println();
    }
}
```

Set.java

```
private double fitnessFunction(Set uncertainElement) {
    return uncertainElement.correlation();
}

public Set decomposeAlexandar() {
    Set lowest = this.createMaxSet();
    double min = Double.MAX_VALUE;
    boolean k = true;
    while (k) {
        k = false;
        ArrayList<Set> neighbours = lowest.generateNeighbours();
        if (neighbours.size() == 0)
            break;
        for (Iterator<Set> iterator = neighbours.iterator(); iterator
            .hasNext();) {
            Set uncertainElement = (Set) iterator.next();
            double tmp = fitnessFunction(uncertainElement);
            if (tmp < min) {
                lowest = uncertainElement;
                min = tmp;
                k = true;
            }
        }
        DecimalFormat f = new DecimalFormat("#.##");
        System.out.println("*" + lowest + "="
            + f.format(lowest.correlation()));
        lowest.printInternal();
    }
    return lowest;
}

public void printLattice() {
    ArrayList<Set> subsets = getSubsets(this.uncertainElements);
    for (Iterator iterator = subsets.iterator(); iterator.hasNext();) {
        Set arrayList = (Set) iterator.next();
        System.out.println(arrayList);
    }
}

public void printAllInfo() {
    System.out.println(this);
    System.out.println("\tEntropy:" + this.entropy());
    System.out.println("\tCorrelation:" + this.correlation());
    System.out.println("\tInteraction:" + this.interaction());
}
}
```

FrequencyTable.java

```
package multivariateanalysis;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.lang.System;

public class FrequencyTable {

    public static String VIEWNAME = "mvc_view";
    public static String TABLENAME = "MVCEXample";

    public static String SCHEMANAME = "test";

    Connection conn = null;
    public static FrequencyTable instance = new FrequencyTable();

    private FrequencyTable() {

        try {
            conn = DriverManager.getConnection("jdbc:mysql://localhost/test?"
                + "user=root&password=alabama01");
        } catch (SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("VendorError: " + ex.getErrorCode());
        }

    }

    public ArrayList<UncertainElement> getVariablesTESTDATA(int count) {
        ArrayList<UncertainElement> result = new ArrayList<UncertainElement>();
        for (int i = 0; i < count; i++) {
            result.add(new Variable("X" + (i + 1)));
        }
        return result;
    }

    public ArrayList<UncertainElement> getVariablesFromDB() {

        ArrayList<UncertainElement> result = new ArrayList<UncertainElement>();

        Statement stmt = null;
        ResultSet rs = null;
        try {
            stmt = conn.createStatement();
            rs = stmt
                .executeQuery("SELECT Column_name FROM information_schema.`COLUMNS`"
                    + " where table_name = '"
                    + TABLENAME
                    + "' and Column_Key<>'PRI'");
            while (rs.next()) {
                result.add(new Variable(rs.getString(1)));
            }
        }

    }

}
```

FrequencyTable.java

```

} catch (SQLException ex) {
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
} finally {

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) {
        } // ignore
        rs = null;
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) {
        } // ignore
        stmt = null;
    }
}
return result;
}

public int getNumberOfItems(String pviewName, String cmpVal) {

    int result = 0;

    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        rs = stmt
            .executeQuery("SELECT COUNT(*) from (select distinct * FROM "
                + pviewName + " Group By " + cmpVal + ") as aa");
        while (rs.next()) {
            result = (rs.getInt(1));
        }
    } catch (SQLException ex) {
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException sqlEx) {
            } // ignore
            rs = null;
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) {
            } // ignore
        }
    }
}

```

FrequencyTable.java

```

        stmt = null;
    }
}
return result;
}

public ArrayList<Integer> getFrequency(String pviewName, String cmpVal,
    String condition) {
    ArrayList<Integer> result = new ArrayList<Integer>();

    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT COUNT(*) FROM " + pviewName + " "
            + condition + " " + " Group By " + cmpVal);
        while (rs.next()) {
            result.add(rs.getInt(1));
        }
    } catch (SQLException ex) {
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    } finally {

        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException sqlEx) {
            } // ignore
            rs = null;
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) {
            } // ignore
            stmt = null;
        }
    }
    return result;
}

public long getTotal(String pviewName, String condition) {
    int total = 0;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT COUNT(*) FROM " + pviewName + " "
            + condition);
        while (rs.next()) {
            total = rs.getInt(1);
        }
    }
}

```

FrequencyTable.java

```

} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
} finally {

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) {
        } // ignore
        rs = null;
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) {
        } // ignore
        stmt = null;
    }
}

return total;
}

public void executeSql(String sql) {
    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate(sql);

    } catch (SQLException ex) {
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    } finally {

        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException sqlEx) {
            } // ignore
            rs = null;
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) {
            } // ignore
            stmt = null;
        }
    }
}
}

```

APPENDIX B

COLLECTED DATA

Cycle #	V1	V2	V3	V4
1	6	0	7	1
2	6	6	42	7
3	11	0	53	42
4	16	0	69	53
5	16	17	1104	69
6	21	3	1125	1106
7	21	23	1146	1125
8	21	27	24066	1150
9	21	28	505386	24071
10	6	0	7	1
11	6	6	42	7
12	11	0	53	42
13	16	0	69	53
14	16	17	1104	69
15	21	3	1125	1106
16	21	23	1146	1125
17	21	27	24066	1150
18	21	28	505386	24071
19	6	0	7	1
20	6	6	42	7
21	11	0	53	42
22	16	0	69	53
23	16	17	1104	69
24	21	3	1125	1106
25	21	23	1146	1125
26	21	27	24066	1150
27	21	28	505386	24071
28	6	0	7	1
29	6	6	42	7
30	11	0	53	42
31	16	0	69	53
32	16	17	1104	69
33	21	3	1125	1106
34	21	23	1146	1125
35	21	27	24066	1150
36	21	28	505386	24071
37	6	0	7	1
38	6	6	42	7
39	11	0	53	42
40	16	0	69	53
41	16	17	1104	69
42	21	3	1125	1106
43	21	23	1146	1125

Cycle #	V1	V2	V3	V4
44	21	27	24066	1150
45	21	28	505386	24071
46	6	0	7	1
47	6	6	42	7
48	11	0	53	42
49	16	0	69	53
50	16	17	1104	69
51	21	3	1125	1106
52	21	23	1146	1125
53	21	27	24066	1150
54	21	28	505386	24071
55	6	0	7	1
56	6	6	42	7
57	11	0	53	42
58	16	0	69	53
59	16	17	1104	69
60	21	3	1125	1106
61	21	23	1146	1125
62	21	27	24066	1150
63	21	28	505386	24071
64	6	0	7	1
65	6	6	42	7
66	11	0	53	42
67	16	0	69	53
68	16	17	1104	69
69	21	3	1125	1106
70	21	23	1146	1125
71	21	27	24066	1150
72	21	28	505386	24071
73	6	0	7	1
74	6	6	42	7
75	11	0	53	42
76	16	0	69	53
77	16	17	1104	69
78	21	3	1125	1106
79	21	23	1146	1125
80	21	27	24066	1150
81	21	28	505386	24071
82	6	0	8	2
83	6	6	48	8
84	6	6	288	48
85	6	6	1728	288
86	6	6	10368	1728

Cycle #	V1	V2	V3	V4
87	6	6	62208	10368
88	6	6	373248	62208
89	6	6	2239488	373248
90	6	6	13436928	2239488
91	6	0	8	2
92	6	6	48	8
93	6	6	288	48
94	6	6	1728	288
95	6	6	10368	1728
96	6	6	62208	10368
97	6	6	373248	62208
98	6	6	2239488	373248
99	6	6	13436928	2239488
100	6	0	8	2
101	6	6	48	8
102	6	6	288	48
103	6	6	1728	288
104	6	6	10368	1728
105	6	6	62208	10368
106	6	6	373248	62208
107	6	6	2239488	373248
108	6	6	13436928	2239488
109	6	0	8	2
110	6	6	48	8
111	6	6	288	48
112	6	6	1728	288
113	6	6	10368	1728
114	6	6	62208	10368
115	6	6	373248	62208
116	6	6	2239488	373248
117	6	6	13436928	2239488
118	6	0	8	2
119	6	6	48	8
120	6	6	288	48
121	6	6	1728	288
122	6	6	10368	1728
123	6	6	62208	10368
124	6	6	373248	62208
125	6	6	2239488	373248
126	6	6	13436928	2239488
127	6	0	8	2
128	6	6	48	8
129	6	6	288	48

Cycle #	V1	V2	V3	V4
130	6	6	1728	288
131	6	6	10368	1728
132	6	6	62208	10368
133	6	6	373248	62208
134	6	6	2239488	373248
135	6	6	13436928	2239488
136	6	0	8	2
137	6	6	48	8
138	6	6	288	48
139	6	6	1728	288
140	6	6	10368	1728
141	6	6	62208	10368
142	6	6	373248	62208
143	6	6	2239488	373248
144	6	6	13436928	2239488
145	6	0	8	2
146	6	6	48	8
147	6	6	288	48
148	6	6	1728	288
149	6	6	10368	1728
150	6	6	62208	10368
151	6	6	373248	62208
152	6	6	2239488	373248
153	6	6	13436928	2239488
154	6	0	8	2
155	6	6	48	8
156	6	6	288	48
157	6	6	1728	288
158	6	6	10368	1728
159	6	6	62208	10368
160	6	6	373248	62208
161	6	6	2239488	373248
162	6	6	13436928	2239488
163	6	0	9	3
164	6	6	54	9
165	11	0	65	54
166	11	12	76	65
167	11	13	87	77
168	11	13	98	88
169	11	13	109	99
170	11	13	120	110
171	16	2	136	121
172	6	0	9	3

Cycle #	V1	V2	V3	V4
173	6	6	54	9
174	11	0	65	54
175	11	12	76	65
176	11	13	87	77
177	11	13	98	88
178	11	13	109	99
179	11	13	120	110
180	16	2	136	121
181	6	0	9	3
182	6	6	54	9
183	11	0	65	54
184	11	12	76	65
185	11	13	87	77
186	11	13	98	88
187	11	13	109	99
188	11	13	120	110
189	16	2	136	121
190	6	0	9	3
191	6	6	54	9
192	11	0	65	54
193	11	12	76	65
194	11	13	87	77
195	11	13	98	88
196	11	13	109	99
197	11	13	120	110
198	16	2	136	121
199	6	0	9	3
200	6	6	54	9
201	11	0	65	54
202	11	12	76	65
203	11	13	87	77
204	11	13	98	88
205	11	13	109	99
206	11	13	120	110
207	16	2	136	121
208	6	0	9	3
209	6	6	54	9
210	11	0	65	54
211	11	12	76	65
212	11	13	87	77
213	11	13	98	88
214	11	13	109	99
215	11	13	120	110

Cycle #	V1	V2	V3	V4
216	16	2	136	121
217	6	0	9	3
218	6	6	54	9
219	11	0	65	54
220	11	12	76	65
221	11	13	87	77
222	11	13	98	88
223	11	13	109	99
224	11	13	120	110
225	16	2	136	121
226	6	0	9	3
227	6	6	54	9
228	11	0	65	54
229	11	12	76	65
230	11	13	87	77
231	11	13	98	88
232	11	13	109	99
233	11	13	120	110
234	16	2	136	121
235	6	0	9	3
236	6	6	54	9
237	11	0	65	54
238	11	12	76	65
239	11	13	87	77
240	11	13	98	88
241	11	13	109	99
242	11	13	120	110
243	16	2	136	121
244	6	0	10	4
245	11	0	21	10
246	16	0	37	21
247	16	17	592	37
248	21	3	613	594
249	26	0	639	613
250	26	28	16614	639
251	31	8	515034	16621
252	36	2	515070	515036
253	6	0	10	4
254	11	0	21	10
255	16	0	37	21
256	16	17	592	37
257	21	3	613	594
258	26	0	639	613

Cycle #	V1	V2	V3	V4
259	26	28	16614	639
260	31	8	515034	16621
261	36	2	515070	515036
262	6	0	10	4
263	11	0	21	10
264	16	0	37	21
265	16	17	592	37
266	21	3	613	594
267	26	0	639	613
268	26	28	16614	639
269	31	8	515034	16621
270	36	2	515070	515036
271	6	0	10	4
272	11	0	21	10
273	16	0	37	21
274	16	17	592	37
275	21	3	613	594
276	26	0	639	613
277	26	28	16614	639
278	31	8	515034	16621
279	36	2	515070	515036
280	6	0	10	4
281	11	0	21	10
282	16	0	37	21
283	16	17	592	37
284	21	3	613	594
285	26	0	639	613
286	26	28	16614	639
287	31	8	515034	16621
288	36	2	515070	515036
289	6	0	10	4
290	11	0	21	10
291	16	0	37	21
292	16	17	592	37
293	21	3	613	594
294	26	0	639	613
295	26	28	16614	639
296	31	8	515034	16621
297	36	2	515070	515036
298	6	0	10	4
299	11	0	21	10
300	16	0	37	21
301	16	17	592	37

Cycle #	V1	V2	V3	V4
302	21	3	613	594
303	26	0	639	613
304	26	28	16614	639
305	31	8	515034	16621
306	36	2	515070	515036
307	6	0	10	4
308	11	0	21	10
309	16	0	37	21
310	16	17	592	37
311	21	3	613	594
312	26	0	639	613
313	26	28	16614	639
314	31	8	515034	16621
315	36	2	515070	515036
316	6	0	10	4
317	11	0	21	10
318	16	0	37	21
319	16	17	592	37
320	21	3	613	594
321	26	0	639	613
322	26	28	16614	639
323	31	8	515034	16621
324	36	2	515070	515036
325	1	1	6	5
326	1	1	7	6
327	1	1	8	7
328	1	1	9	8
329	1	1	10	9
330	6	0	16	10
331	6	6	96	16
332	6	6	576	96
333	6	6	3456	576
334	1	1	6	5
335	1	1	7	6
336	1	1	8	7
337	1	1	9	8
338	1	1	10	9
339	6	0	16	10
340	6	6	96	16
341	6	6	576	96
342	6	6	3456	576
343	1	1	6	5
344	1	1	7	6

Cycle #	V1	V2	V3	V4
345	1	1	8	7
346	1	1	9	8
347	1	1	10	9
348	6	0	16	10
349	6	6	96	16
350	6	6	576	96
351	6	6	3456	576
352	1	1	6	5
353	1	1	7	6
354	1	1	8	7
355	1	1	9	8
356	1	1	10	9
357	6	0	16	10
358	6	6	96	16
359	6	6	576	96
360	6	6	3456	576
361	1	1	6	5
362	1	1	7	6
363	1	1	8	7
364	1	1	9	8
365	1	1	10	9
366	6	0	16	10
367	6	6	96	16
368	6	6	576	96
369	6	6	3456	576
370	1	1	6	5
371	1	1	7	6
372	1	1	8	7
373	1	1	9	8
374	1	1	10	9
375	6	0	16	10
376	6	6	96	16
377	6	6	576	96
378	6	6	3456	576
379	1	1	6	5
380	1	1	7	6
381	1	1	8	7
382	1	1	9	8
383	1	1	10	9
384	6	0	16	10
385	6	6	96	16
386	6	6	576	96
387	6	6	3456	576

Cycle #	V1	V2	V3	V4
388	1	1	6	5
389	1	1	7	6
390	1	1	8	7
391	1	1	9	8
392	1	1	10	9
393	6	0	16	10
394	6	6	96	16
395	6	6	576	96
396	6	6	3456	576
397	1	1	6	5
398	1	1	7	6
399	1	1	8	7
400	1	1	9	8
401	1	1	10	9
402	6	0	16	10
403	6	6	96	16
404	6	6	576	96
405	6	6	3456	576
406	1	1	7	6
407	1	1	8	7
408	1	1	9	8
409	1	1	10	9
410	6	0	16	10
411	6	6	96	16
412	6	6	576	96
413	6	6	3456	576
414	6	6	20736	3456
415	1	1	7	6
416	1	1	8	7
417	1	1	9	8
418	1	1	10	9
419	6	0	16	10
420	6	6	96	16
421	6	6	576	96
422	6	6	3456	576
423	6	6	20736	3456
424	1	1	7	6
425	1	1	8	7
426	1	1	9	8
427	1	1	10	9
428	6	0	16	10
429	6	6	96	16
430	6	6	576	96

Cycle #	V1	V2	V3	V4
431	6	6	3456	576
432	6	6	20736	3456
433	1	1	7	6
434	1	1	8	7
435	1	1	9	8
436	1	1	10	9
437	6	0	16	10
438	6	6	96	16
439	6	6	576	96
440	6	6	3456	576
441	6	6	20736	3456
442	1	1	7	6
443	1	1	8	7
444	1	1	9	8
445	1	1	10	9
446	6	0	16	10
447	6	6	96	16
448	6	6	576	96
449	6	6	3456	576
450	6	6	20736	3456
451	1	1	7	6
452	1	1	8	7
453	1	1	9	8
454	1	1	10	9
455	6	0	16	10
456	6	6	96	16
457	6	6	576	96
458	6	6	3456	576
459	6	6	20736	3456
460	1	1	7	6
461	1	1	8	7
462	1	1	9	8
463	1	1	10	9
464	6	0	16	10
465	6	6	96	16
466	6	6	576	96
467	6	6	3456	576
468	6	6	20736	3456
469	1	1	7	6
470	1	1	8	7
471	1	1	9	8
472	1	1	10	9
473	6	0	16	10

Cycle #	V1	V2	V3	V4
474	6	6	96	16
475	6	6	576	96
476	6	6	3456	576
477	6	6	20736	3456
478	1	1	7	6
479	1	1	8	7
480	1	1	9	8
481	1	1	10	9
482	6	0	16	10
483	6	6	96	16
484	6	6	576	96
485	6	6	3456	576
486	6	6	20736	3456
487	1	1	8	7
488	1	1	9	8
489	1	1	10	9
490	6	0	16	10
491	6	6	96	16
492	6	6	576	96
493	6	6	3456	576
494	6	6	20736	3456
495	6	6	124416	20736
496	1	1	8	7
497	1	1	9	8
498	1	1	10	9
499	6	0	16	10
500	6	6	96	16
501	6	6	576	96
502	6	6	3456	576
503	6	6	20736	3456
504	6	6	124416	20736
505	1	1	8	7
506	1	1	9	8
507	1	1	10	9
508	6	0	16	10
509	6	6	96	16
510	6	6	576	96
511	6	6	3456	576
512	6	6	20736	3456
513	6	6	124416	20736
514	1	1	8	7
515	1	1	9	8
516	1	1	10	9

Cycle #	V1	V2	V3	V4
517	6	0	16	10
518	6	6	96	16
519	6	6	576	96
520	6	6	3456	576
521	6	6	20736	3456
522	6	6	124416	20736
523	1	1	8	7
524	1	1	9	8
525	1	1	10	9
526	6	0	16	10
527	6	6	96	16
528	6	6	576	96
529	6	6	3456	576
530	6	6	20736	3456
531	6	6	124416	20736
532	1	1	8	7
533	1	1	9	8
534	1	1	10	9
535	6	0	16	10
536	6	6	96	16
537	6	6	576	96
538	6	6	3456	576
539	6	6	20736	3456
540	6	6	124416	20736
541	1	1	8	7
542	1	1	9	8
543	1	1	10	9
544	6	0	16	10
545	6	6	96	16
546	6	6	576	96
547	6	6	3456	576
548	6	6	20736	3456
549	6	6	124416	20736
550	1	1	8	7
551	1	1	9	8
552	1	1	10	9
553	6	0	16	10
554	6	6	96	16
555	6	6	576	96
556	6	6	3456	576
557	6	6	20736	3456
558	6	6	124416	20736
559	1	1	8	7

Cycle #	V1	V2	V3	V4
560	1	1	9	8
561	1	1	10	9
562	6	0	16	10
563	6	6	96	16
564	6	6	576	96
565	6	6	3456	576
566	6	6	20736	3456
567	6	6	124416	20736
568	1	1	9	8
569	1	1	10	9
570	6	0	16	10
571	6	6	96	16
572	6	6	576	96
573	6	6	3456	576
574	6	6	20736	3456
575	6	6	124416	20736
576	6	6	746496	124416
577	1	1	9	8
578	1	1	10	9
579	6	0	16	10
580	6	6	96	16
581	6	6	576	96
582	6	6	3456	576
583	6	6	20736	3456
584	6	6	124416	20736
585	6	6	746496	124416
586	1	1	9	8
587	1	1	10	9
588	6	0	16	10
589	6	6	96	16
590	6	6	576	96
591	6	6	3456	576
592	6	6	20736	3456
593	6	6	124416	20736
594	6	6	746496	124416
595	1	1	9	8
596	1	1	10	9
597	6	0	16	10
598	6	6	96	16
599	6	6	576	96
600	6	6	3456	576
601	6	6	20736	3456
602	6	6	124416	20736

Cycle #	V1	V2	V3	V4
603	6	6	746496	124416
604	1	1	9	8
605	1	1	10	9
606	6	0	16	10
607	6	6	96	16
608	6	6	576	96
609	6	6	3456	576
610	6	6	20736	3456
611	6	6	124416	20736
612	6	6	746496	124416
613	1	1	9	8
614	1	1	10	9
615	6	0	16	10
616	6	6	96	16
617	6	6	576	96
618	6	6	3456	576
619	6	6	20736	3456
620	6	6	124416	20736
621	6	6	746496	124416
622	1	1	9	8
623	1	1	10	9
624	6	0	16	10
625	6	6	96	16
626	6	6	576	96
627	6	6	3456	576
628	6	6	20736	3456
629	6	6	124416	20736
630	6	6	746496	124416
631	1	1	9	8
632	1	1	10	9
633	6	0	16	10
634	6	6	96	16
635	6	6	576	96
636	6	6	3456	576
637	6	6	20736	3456
638	6	6	124416	20736
639	6	6	746496	124416
640	1	1	9	8
641	1	1	10	9
642	6	0	16	10
643	6	6	96	16
644	6	6	576	96
645	6	6	3456	576

Cycle #	V1	V2	V3	V4
646	6	6	20736	3456
647	6	6	124416	20736
648	6	6	746496	124416
649	1	1	10	9
650	6	0	16	10
651	6	6	96	16
652	6	6	576	96
653	6	6	3456	576
654	6	6	20736	3456
655	6	6	124416	20736
656	6	6	746496	124416
657	6	6	4478976	746496
658	1	1	10	9
659	6	0	16	10
660	6	6	96	16
661	6	6	576	96
662	6	6	3456	576
663	6	6	20736	3456
664	6	6	124416	20736
665	6	6	746496	124416
666	6	6	4478976	746496
667	1	1	10	9
668	6	0	16	10
669	6	6	96	16
670	6	6	576	96
671	6	6	3456	576
672	6	6	20736	3456
673	6	6	124416	20736
674	6	6	746496	124416
675	6	6	4478976	746496
676	1	1	10	9
677	6	0	16	10
678	6	6	96	16
679	6	6	576	96
680	6	6	3456	576
681	6	6	20736	3456
682	6	6	124416	20736
683	6	6	746496	124416
684	6	6	4478976	746496
685	1	1	10	9
686	6	0	16	10
687	6	6	96	16
688	6	6	576	96

Cycle #	V1	V2	V3	V4
689	6	6	3456	576
690	6	6	20736	3456
691	6	6	124416	20736
692	6	6	746496	124416
693	6	6	4478976	746496
694	1	1	10	9
695	6	0	16	10
696	6	6	96	16
697	6	6	576	96
698	6	6	3456	576
699	6	6	20736	3456
700	6	6	124416	20736
701	6	6	746496	124416
702	6	6	4478976	746496
703	1	1	10	9
704	6	0	16	10
705	6	6	96	16
706	6	6	576	96
707	6	6	3456	576
708	6	6	20736	3456
709	6	6	124416	20736
710	6	6	746496	124416
711	6	6	4478976	746496
712	1	1	10	9
713	6	0	16	10
714	6	6	96	16
715	6	6	576	96
716	6	6	3456	576
717	6	6	20736	3456
718	6	6	124416	20736
719	6	6	746496	124416
720	6	6	4478976	746496
721	1	1	10	9
722	6	0	16	10
723	6	6	96	16
724	6	6	576	96
725	6	6	3456	576
726	6	6	20736	3456
727	6	6	124416	20736
728	6	6	746496	124416
729	6	6	4478976	746496
730	6	0	7	1
731	6	6	42	7

Cycle #	V1	V2	V3	V4
732	11	0	53	42
733	16	0	69	53
734	16	17	1104	69
735	21	3	1125	1106
736	21	23	1146	1125
737	21	27	24066	1150
738	21	28	505386	24071
739	6	0	7	1
740	6	6	42	7
741	11	0	53	42
742	16	0	69	53
743	16	17	1104	69
744	21	3	1125	1106
745	21	23	1146	1125
746	21	27	24066	1150
747	21	28	505386	24071
748	6	0	7	1
749	6	6	42	7
750	11	0	53	42
751	16	0	69	53
752	16	17	1104	69
753	21	3	1125	1106
754	21	23	1146	1125
755	21	27	24066	1150
756	21	28	505386	24071
757	6	0	7	1
758	6	6	42	7
759	11	0	53	42
760	16	0	69	53
761	16	17	1104	69
762	21	3	1125	1106
763	21	23	1146	1125
764	21	27	24066	1150
765	21	28	505386	24071
766	6	0	7	1
767	6	6	42	7
768	11	0	53	42
769	16	0	69	53
770	16	17	1104	69
771	21	3	1125	1106
772	21	23	1146	1125
773	21	27	24066	1150
774	21	28	505386	24071

Cycle #	V1	V2	V3	V4
775	6	0	7	1
776	6	6	42	7
777	11	0	53	42
778	16	0	69	53
779	16	17	1104	69
780	21	3	1125	1106
781	21	23	1146	1125
782	21	27	24066	1150
783	21	28	505386	24071
784	6	0	7	1
785	6	6	42	7
786	11	0	53	42
787	16	0	69	53
788	16	17	1104	69
789	21	3	1125	1106
790	21	23	1146	1125
791	21	27	24066	1150
792	21	28	505386	24071
793	6	0	7	1
794	6	6	42	7
795	11	0	53	42
796	16	0	69	53
797	16	17	1104	69
798	21	3	1125	1106
799	21	23	1146	1125
800	21	27	24066	1150
801	21	28	505386	24071
802	6	0	7	1
803	6	6	42	7
804	11	0	53	42
805	16	0	69	53
806	16	17	1104	69
807	21	3	1125	1106
808	21	23	1146	1125
809	21	27	24066	1150
810	21	28	505386	24071
811	6	0	8	2
812	6	6	48	8
813	6	6	288	48
814	6	6	1728	288
815	6	6	10368	1728
816	6	6	62208	10368
817	6	6	373248	62208

Cycle #	V1	V2	V3	V4
818	6	6	2239488	373248
819	6	6	13436928	2239488
820	6	0	8	2
821	6	6	48	8
822	6	6	288	48
823	6	6	1728	288
824	6	6	10368	1728
825	6	6	62208	10368
826	6	6	373248	62208
827	6	6	2239488	373248
828	6	6	13436928	2239488
829	6	0	8	2
830	6	6	48	8
831	6	6	288	48
832	6	6	1728	288
833	6	6	10368	1728
834	6	6	62208	10368
835	6	6	373248	62208
836	6	6	2239488	373248
837	6	6	13436928	2239488
838	6	0	8	2
839	6	6	48	8
840	6	6	288	48
841	6	6	1728	288
842	6	6	10368	1728
843	6	6	62208	10368
844	6	6	373248	62208
845	6	6	2239488	373248
846	6	6	13436928	2239488
847	6	0	8	2
848	6	6	48	8
849	6	6	288	48
850	6	6	1728	288
851	6	6	10368	1728
852	6	6	62208	10368
853	6	6	373248	62208
854	6	6	2239488	373248
855	6	6	13436928	2239488
856	6	0	8	2
857	6	6	48	8
858	6	6	288	48
859	6	6	1728	288
860	6	6	10368	1728

Cycle #	V1	V2	V3	V4
861	6	6	62208	10368
862	6	6	373248	62208
863	6	6	2239488	373248
864	6	6	13436928	2239488
865	6	0	8	2
866	6	6	48	8
867	6	6	288	48
868	6	6	1728	288
869	6	6	10368	1728
870	6	6	62208	10368
871	6	6	373248	62208
872	6	6	2239488	373248
873	6	6	13436928	2239488
874	6	0	8	2
875	6	6	48	8
876	6	6	288	48
877	6	6	1728	288
878	6	6	10368	1728
879	6	6	62208	10368
880	6	6	373248	62208
881	6	6	2239488	373248
882	6	6	13436928	2239488
883	6	0	8	2
884	6	6	48	8
885	6	6	288	48
886	6	6	1728	288
887	6	6	10368	1728
888	6	6	62208	10368
889	6	6	373248	62208
890	6	6	2239488	373248
891	6	6	13436928	2239488
892	6	0	9	3
893	6	6	54	9
894	11	0	65	54
895	11	12	76	65
896	11	13	87	77
897	11	13	98	88
898	11	13	109	99
899	11	13	120	110
900	16	2	136	121
901	6	0	9	3
902	6	6	54	9
903	11	0	65	54

Cycle #	V1	V2	V3	V4
904	11	12	76	65
905	11	13	87	77
906	11	13	98	88
907	11	13	109	99
908	11	13	120	110
909	16	2	136	121
910	6	0	9	3
911	6	6	54	9
912	11	0	65	54
913	11	12	76	65
914	11	13	87	77
915	11	13	98	88
916	11	13	109	99
917	11	13	120	110
918	16	2	136	121
919	6	0	9	3
920	6	6	54	9
921	11	0	65	54
922	11	12	76	65
923	11	13	87	77
924	11	13	98	88
925	11	13	109	99
926	11	13	120	110
927	16	2	136	121
928	6	0	9	3
929	6	6	54	9
930	11	0	65	54
931	11	12	76	65
932	11	13	87	77
933	11	13	98	88
934	11	13	109	99
935	11	13	120	110
936	16	2	136	121
937	6	0	9	3
938	6	6	54	9
939	11	0	65	54
940	11	12	76	65
941	11	13	87	77
942	11	13	98	88
943	11	13	109	99
944	11	13	120	110
945	16	2	136	121
946	6	0	9	3

Cycle #	V1	V2	V3	V4
947	6	6	54	9
948	11	0	65	54
949	11	12	76	65
950	11	13	87	77
951	11	13	98	88
952	11	13	109	99
953	11	13	120	110
954	16	2	136	121
955	6	0	9	3
956	6	6	54	9
957	11	0	65	54
958	11	12	76	65
959	11	13	87	77
960	11	13	98	88
961	11	13	109	99
962	11	13	120	110
963	16	2	136	121
964	6	0	9	3
965	6	6	54	9
966	11	0	65	54
967	11	12	76	65
968	11	13	87	77
969	11	13	98	88
970	11	13	109	99
971	11	13	120	110
972	16	2	136	121
973	6	0	10	4
974	11	0	21	10
975	16	0	37	21
976	16	17	592	37
977	21	3	613	594
978	26	0	639	613
979	26	28	16614	639
980	31	8	515034	16621
981	36	2	515070	515036
982	6	0	10	4
983	11	0	21	10
984	16	0	37	21
985	16	17	592	37
986	21	3	613	594
987	26	0	639	613
988	26	28	16614	639
989	31	8	515034	16621

Cycle #	V1	V2	V3	V4
990	36	2	515070	515036
991	6	0	10	4
992	11	0	21	10
993	16	0	37	21
994	16	17	592	37
995	21	3	613	594
996	26	0	639	613
997	26	28	16614	639
998	31	8	515034	16621
999	36	2	515070	515036
1000	6	0	10	4