Communicating Software Design Patterns with InfoMaps

Athanassios A. Michailidis

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for

the Degree of Master of Computer Science at

Concordia University

Montréal, Quebéc, Canada

March 1995

ISBN 0-612-01364-2

Abstract

Communicating Software Design Patterns with InfoMaps

Athanassios A. Michailidis

Problems encountered during the design and analysis of software vary. The solutions applied to these problems can be used several times without performing the same task twice. Design Patterns describe problems which occur over and over again during the software design process. Several parts constitute a Design Pattern. These parts can be represented uniformly if the appropriate notation is available to us.

In this thesis we study the notation of the InfoMap representation methodology and its applications. Steps are presented for the construction of InfoMap models. The proposed framework is derived by the application of heuristics on the context-free grammar production rules designed specifically for the InfoMap notation. These rules provide automation for the conversion of the production rules to a framework structure. The design framework structure is compatible with rules for designing classes and frameworks found in the literature. The framework used to accomplish this task is also presented using its own notation, and modeled using the Rational Inc. CASE tool. The notation and the framework that accompanies the methodology are used as a common vocabulary for the presentation and communication of Design Patterns. Furthermore the characteristics of Design Patterns are compared to InfoMap/InfoSchema characteristics.

For the purpose of this study we present the InfoRun system, which simulates the execution of Control Flow Graphs presented in the InfoMap notation. Control Flow Graphs can be imported in the InfoRun system and manipulated during run-time. This allows us to experiment with Control Flow Graphs and model the source code of the Design Patterns in an executable fashion using the InfoMap notation.

Acknowledgments

This could have been an easier task, had I listened to the advice of my supervisor, Dr. W.M. Jaworski, on two occasions: the first in January of 1993 and the second in November of 1994. In both cases (totally unrelated events) I realized what it means to be working with him on and off campus, and seeking his academic and personal advice. Special thanks to him and to my family for their support. Finally, I would like to express my appreciation to Maria Skitzis, my LIFE 101 Instructor.

Epigram

"The only thing I know is that I know nothing."
Socrates

Table of Contents

List of Figures

Trademarks

CHAPTER 1: INTRODUCTION

1.1 Context

There exist a wide variety of design techniques and programming tools. These techniques and tools provide assistance to software developers and researchers, who live and breathe in a world that contains several notions that need to be related and classified in a meaningful manner. These notions include entities, processes, locations, people, times and purposes [3]. Therefore, producing quality software depends not only on tools and techniques for design and programming, but also on the several pieces of information that need to be arranged and viewed in a clear, non-ambiguous style. Most programming techniques and design methodologies provide ways for interrelating pieces of information for efficient and effective viewing and communication purposes.

The viewing of the several pieces of information that influence design and programming decisions can be performed in several ways. A common language that can describe each system viewing style is needed. A natural language such as English is capable of describing everything [7, 8, 14]. Yet, although English may be a good communication tool for discussion of design decisions in the early stages of design, it is not as precise as a more specialized notation that captures all the views of a system. Symbolic logic is precise and general enough to describe anything that can be implemented as a software entity [4]. But the usual predicate calculus notation for logic tends to become unreadable, even for simple examples. Conceptual graphs are a readable graphic notation for logic that is designed for translations to and from natural languages [5, 6]. Because of the generality and readability of conceptual graphs, they have become popular among developers of methodologies and systems. Nevertheless, in several cases they fail to capture the exact interrelationship of the components of a system, since they introduce unnecessary complexity due to misleading notations [9].

The primary goal of software engineering is high-quality software. Specifically, this means software that is appropriately compatible, correct, correctable, efficient, extensible, maintainable, modifiable, portable, reliable, reusable, robust, safe, secure, testable, understandable, user-friendly, validatable and verifiable

[2]. Therefore, in order to introduce all these qualities in software design and implementation, the display of the components that constitute a system should be crystal-clear. The InfoMap methodology, and its applications described in this thesis, have been proposed as a notation and representation approach for viewing overall structures of systems. Its abstraction levels provide the advantage of focusing on one aspect of a system while ignoring others, and its notation provides an implementable framework. Therefore we have chosen to describe and then use this methodology in order to analytically communicate ideas.

1.2 Objective

This work has multiple objectives. The main objective is to establish a representation methodology for the effective and efficient communication of design patterns [1]. In order to accomplish this objective, the methodology of producing InfoMap models is presented. The secondary objectives of this work are: first, to present the context-free grammar of the InfoMap methodology, and its transformation to a design framework; second, to present this framework in its own terms; and third, to present a mechanism for executing control flow graphs represented in the InfoMap methodology. The experience gained through this exercise of organizing a methodology, presenting it in terms of a framework, using it and designing tools for it tested the completeness of this methodology.

1.3 Organization

This thesis is organized into seven chapters and three appendices. In chapter 2 the basics of the methodology for the derivation of the InfoMap models are presented. The proposed methodology for deriving models is presented as a sequence of steps. These steps start with the definition of a context-free grammar for the methodology, continue with the description of a process for the application of the production rules of the context-free grammar, and finally apply algorithms for the conversion of terminal tokens to InfoMap models. In chapter 3 the design framework as well as framework related issues are presented. The proposed design framework is derived by applying certain heuristic rules to the context-free grammar that transforms the production rules into classes and their inheritance relationships. In chapter 4 the framework along with the basics of the methodology are used to provide a model for design patterns. This provides a demonstration of the framework and its capabilities. Chapter 5 offers a brief description of the basics and the framework of the methodology presented in its own terms. In chapter 6 the InfoRun system is presented. It is a tool for the execution of control flow graphs, presented in terms of the InfoMap methodology. The conclusion of the thesis follows chapter 6. In the conclusion we summarize the findings and suggest further research in the areas of design pattern representation, the InfoMap methodology, and the improvement of the InfoRun tool. In the first appendix of this thesis we present the specifications of the design framework, discussed in chapter 3, produced by the Rational Inc. CASE tool [15]. In the second appendix we present the same specifications modeled using the InfoMap methodology. In the third appendix, we present the same specifications given in the first appendix using the "export" function provided by the Rational Inc. CASE tool. This function exports the specifications for further manipulation by other systems.

CHAPTER 2: METHODOLOGY

2.1 Introduction, Motivation

Models are abstractions that provide problem analysts with the capability of having a diagrammatic, textual, and/or mathematical view of a given problem. Models in general are used to represent specific domains of knowledge. In the software community several kinds of models are used to represent the software knowledge specific domain. This domain includes documentation (Software Requirements Documents and Software Specifications Documents [11]) and design (Data Flow Diagrams and Control Flow Diagrams [12, 13]). Entities contained in this domain need to be modeled in a clear, non-ambiguous way. This clarity should guarantee the visibility of every aspect within these entities. One approach is to partition a specific knowledge domain into sets of items, define the relationships between and within these sets of items, and determine levels of representation abstraction.

The InfoMap approach transforms a given 2-dimensional screen into an *n*-dimensional space by grouping the rows and columns of a spreadsheet. The idea that this representation is possible was initiated in the late 1960's at CROPI [54]. CROPI was an institute responsible for the organization and coordination of the Polish heavy industry in the late 1960's. This organization and coordination of the Polish industry (mining and manufacturing) was targeted towards 900,000 employees in total. The advancement of the InfoMap methodology is mainly attributed to Dr. W.M. Jaworski at Concordia University, who was a founder member of CORPI, and an advisor of the minister of the Polish heavy industry in the late 1960's[53]. There have been several thesis completed under his supervision. These theses focused on software production related problems and concerns. The most important follow in chronological order.

In 1983, L. Ficocelli [49] studied the minimization of the inherited difficulties while transforming problems to programs. This study showed how it is possible to use the InfoMap methodology (previously known as ABL: Alternative Based Language) in order to exploit several problems. The same year, M. Kronick [48] studied the InfoMap methodology from the point of view of a computer-based automatic dialing system. His research emphasized more the structured design techniques, the modulization and the step-wise construction of the InfoMap models. In 1988 another two thesis were presented. The first thesis, by D. Eddy [52], presented the InfoMap methodology in terms of the software life cycle. In his thesis database models were used to demonstrate the automatic verification of models for redundancy and inconsistency related issues. The second thesis, by K. Finkelstein [51], presented a first attempt into creating an editor for the tabular structure of the InfoMap models. This was the first attempt to built tools for the manipulation of the InfoMap tabular structure. Later on, in 1993 another two thesis were presented. The first thesis, by B. Deslauriers [50], focused on the inspection of software deliverables. These deliverables were presented, and inspected according to several industry quality standards with the help of the InfoMap methodology. The second thesis, by T. Cummings [43], used the InfoMap methodology on several algorithms. The results of his research showed how it is possible to structure and present algorithms in a state-transition environment presented using the InfoMap methodology. Furthermore, his research showed how it is possible to use this methodology in order to optimize algorithms with the elimination and/or merging of states and transitions. Later on, in 1992, S. Kattou [18], used the InfoMap methodology to present the synthetic and reusable products of the software process. In his work the focus was mainly on the modeling and the synthesis of the several existing software models (*i.e.* : Behavior, Object-Oriented).

In addition to the various thesis published at Concordia University, there have been several publications in journals and conferences on the subject of the InfoMap methodology. The following are the most important. In 1987 the InfoMap methodology is presented as an environment that provides a powerful, language independent, infrastructure for the transformation of problems into programs [55]. This work focused on the life cycle software engineering of expert systems, and showed how it is possible to provide solutions for common problems found in the area of expert systems. In the same year a system's methodology evolution is presented in terms of the InfoMap methodology [56]. This evolution is based on the idea that the InfoMap methodology can be used to built evolving models, therefore whole systems may be built, analyzed and evolved around the InfoMap methodology. Another attempt, as in [51], to built automation tools for the InfoMap methodology was presented in 1970 [57]. In this work, a decision table simulator represented using the InfoMap methodology is given. The tool built for this purpose uses a selection strategy in order to choose among candidate transitions, one transition that will reach faster the desire goal or solution to a given problem. The most recent publication, by Dr. W. M. Jaworski and the author of this thesis, was in 1994 [10]. In this paper the basics as well as the "query-by-structure" notion was presented. The "query-by-structure" notion extracts only the essential information from the InfoMap model of a body of knowledge while ignoring the rest.

In this chapter we present only the basics of the InfoMap knowledge representation methodology. These basics are: sets, relationships and levels of abstraction. Based on these, we present the syntactical issues and the production methodology for InfoMap models. We specify the concepts that relate sets to the InfoMap models. These concepts are summarized by four attributes: SetName, SetMember, SetRole, and SetMemberRole. These attributes assist the creation, description and naming of relationships at two abstraction levels: the general level or InfoSchema, and the detailed level or InfoMap. Following the discussion on sets we present the rationale behind specific kinds of set roles. These roles are attached to sets in order to provide meaning to relationships between and within sets. The production of models using the InfoMap methodology is based on the idea that the sets, relationships and levels of abstraction can be represented using context-free grammars. Processes for the application of the context-free grammars are presented and applied in order to derive terminal symbols which are used as input to algorithms that produce the InfoMap models at the general and detailed level of abstraction.

2.2 Methodology Definitions

Theory and practice in the area of software engineering evolve around statements made about objects [2]. These objects have special properties. For example, numbers may be odd or even, letters may be upper or lower case, and database records may contain strings of characters or multimedia objects such as images and/or sounds. Properties like these may be used as a guide in order to group objects together and form sets. A set is a collection of objects that share one or more common properties [16]. Set objects may overlap when one or more objects in a given set are also present in another set. Unique sets do not share their set members with other sets. The overlap is attributed to properties of different sets that bring objects together. These properties that place objects together in the same set also place some of these objects together in a different set. Using this statement as an inspiration for our approach, we identify the sets and the sets of sets that may exist in a world describing objects and being described by objects, in terms of the InfoMap methodology.

2.2.1 Methodology Attributes

Each SetName contains SetMembers. A SetName is related to other SetNames in a given Partition by its assigned SetRoles. The contents of all SetNames are selectively related to each other by their assigned SetMemberRoles. Furthermore, SetMemberRoles may also selectively relate SetMembers from different SetNames. This preliminary description of the relationships between SetName, SetMember, SetRole and SetMemberRole is presented in figure 2.1.

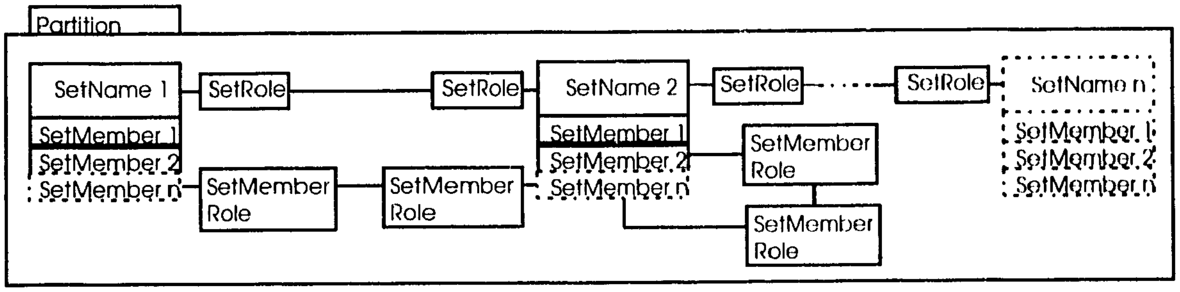


Figure 2.1. Relationship Between SetName, SetRole, SetMember, and SetMemberRole: A Diagrammatic Description of a Partition

A SetName is an abstract definition of a set. It keeps its contents hidden. Furthermore, it requires a specific role in order to have a meaning. This role is described by its relationship to other SetNames within a collection of several SetNames. The collection of several SetNames is a SetColumn.

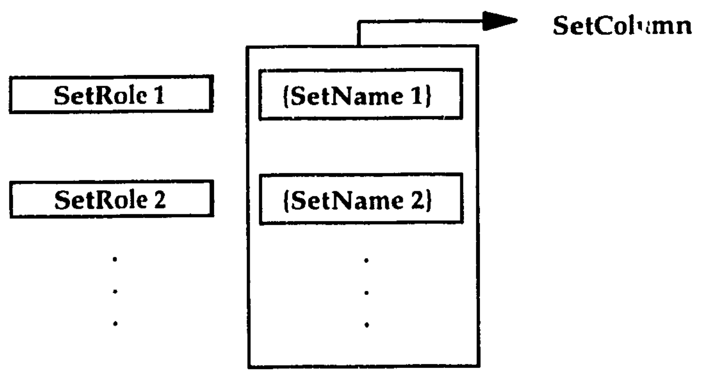


Figure 2.2. Diagrammatic Description of SetName

In figure 2.2 a SetName belongs in the SetColumn collection of several SetNames. It is related to other SetNames in the SetColumn through the SetRole attached on its left-hand side. Furthermore, the SetRole provides an indication about the relationships that exist among the SetMembers of a SetName.

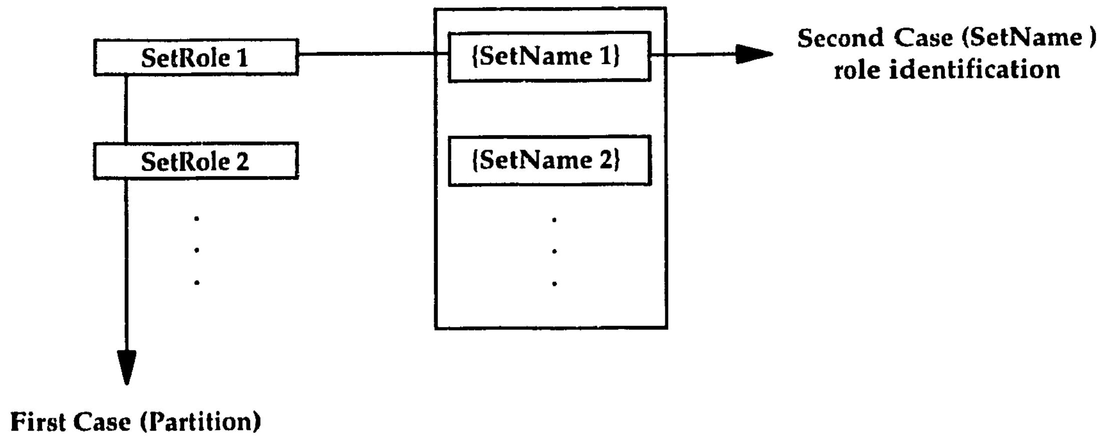


Figure 2.3. Diagrammatic Description of SetRole

A SetRole is a relationship identification for a SetName. It describes the specific handling that was mentioned in the previous definition of the SetName. This identification card may be used for two kinds of relationship identifications:

[1] The relationship that may exist between two or more SetNames.
[2] The relationship that may exist between the contents of one SetName.

The first case is called a Partition. Each time a new relationship situation is encountered between two or more SetNames, a new SetRole is assigned to each SetName. Therefore the SetRoles that are attached to the several SetNames of a given Partition provide a meaningful relationship between SetNames. In the description of a SetRole we limit the number of SetRoles by defining a restricted collection of valid SetRoles. This is done for semantic purposes. The valid SetRoles are presented later in our discussion.

The second case is a simple SetRole attached to a SetName. It describes the existence of a relationship between the SetMembers of that SetName. These two cases of SetRoles are presented in figure 2.3.

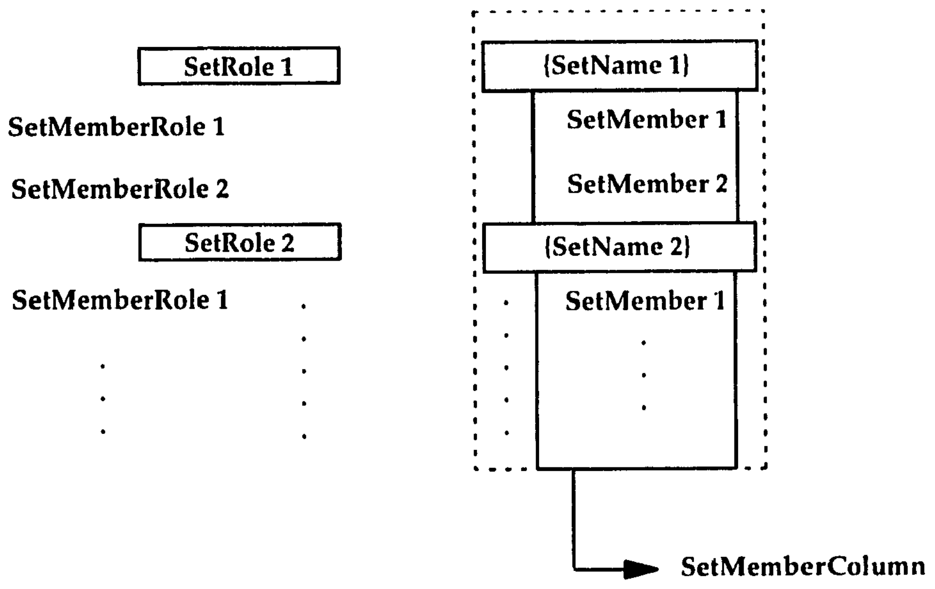


Figure 2.4. Diagrammatic Description of SetMember

The elements contained in SetNames are called SetMembers. They are located in the SetMemberColumn. In each SetName, SetMembers are unique. These SetMembers need not be interrelated in any way with one another. On the other hand, there may be a relationship between them. In figure 2.4, a SetMember belongs in a SetName. It is related to other SetMembers in other SetNames or within its own SetName by a SetMemberRole.

So far we have seen SetNames and SetMembers as collections in the SetColumn, and SetMemberColumn. What remains to be seen is what assists us in relating these two levels. It would be helpful to have a convenient way of representing these relationships. Therefore we use the SetRole to describe the relationships that exist at the general level (collection of SetNames in SetColumn). On the other hand, at the detailed level (a collection of SetMembers in SetMemberColumn) we use the SetMemberRole to describe relationships which are partitioned into the following three cases:

First Case :	SetMembers of the same SetName
Second Case :	SetMembers of different SetNames
Third Case :	Both first and second cases

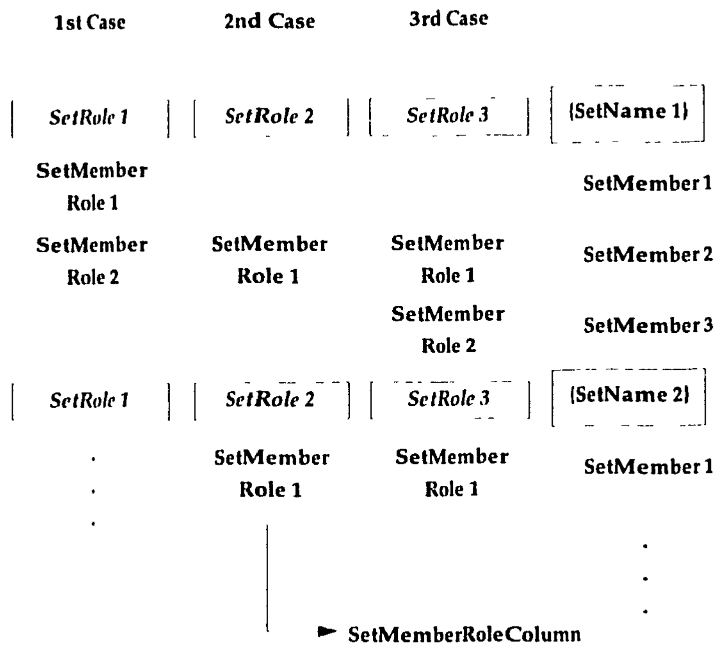


Figure 2.5. Diagrammatic Description of SetMemberRole

The three cases are shown in figure 2.5. As in the description of the SetRole (figure 2.3), the SetMemberRole is also an identification attached to a SetMember. The purpose of this identification card is to establish a meaningful relationship between SetMembers. In figure 2.5, a SetMemberRole belongs in a SetMemberRoleColumn which is a collection of SetMemberRoles. There are also only a few valid SetMemberRoles.

2.2.2 Valid Role Definitions

As we mentioned in the previous section, there exists a limitation in the number of valid SetRoles and SetMemberRoles. There are two reasons why this limitation exists:

- We should identify sets of relationships and sets of SetRoles. It is obvious that while we use SetRoles attached to SetNames in order to analyze real world situations, at the same time certain SetRoles may be further decomposed into more primitive SetRoles.

- Second, we prepare the ground for the introduction of the context-free Grammar that encapsulates these SetRoles.

The SetRoles are presented in four groups, each contains primitive SetRoles extracted from common knowledge and experience.

[1] The Partition SetRole is a collection of SetNames which are related to one another. Each partition is unique and represents the distinction between several groups of relationships.

[2] The Dominant SetRole is attached to SetNames in order to specify the uniqueness of the SetName and its SetMembers in a given Partition. Furthermore, it implies the uniqueness of the relationship that exists between a SetName and its SetMembers to other non-dominant SetNames and their SetMembers in the same Partition. The Dominant SetRole is further decomposed into more primitive SetRoles:

- The Identifier which is used to number and/or name sequences of unique relationships presented to a SetMember.

- The Identity which is used to represent the "one" in a "one-to-many" items relationship.

- The Hierarchy which is used to represent a hierarchical structure of several SetMembers.

- The Generalization which is used to represent the "parent-child" relationship between SetMembers.

- The Aggregation which is used to represent "part-of" relationships between SetMembers.

[3] Following the group of Dominant SetRoles, we identify the group of Descriptive SetRoles. A Descriptive SetRole describes complex relationships between and within sets. These SetRoles are essential for describing patterns that develop during the specification of relationships. The relationships that exists between sets and can be identified as patterns may be described as: Qualifiers, Associations, Flows, Guards, Sequences, and Values.

- A Qualifier SetRole exists in a "one - to - one" relationship, between a dominant and a non-dominant identified SetMember.

- An Association SetRole exists in a "many - to - many" relationship, and it is the "many" part of a "one - to - many" relationship.

- A Flow SetRole may be assigned to the SetMembers of a SetName if they describe the input and output data of a system.

- A Guard SetRole is used to specify the validity of statements and/or expressions which are members of a SetName.

- A Sequence SetRole is used to order the SetMembers of a SetName.

- A Value SetRole is used when the SetRoles themselves are either string, integer or Boolean values.

[4] Finally there exists a certain group of SetRoles that may be used to describe systems in terms of states and transitions. These are the Transitive SetRoles. They are classified into Sequential and Concurrent SetRoles.

- A Sequential SetRole is assigned to a SetName if the contents of that SetName describe a sequential state - transition system.
- A Concurrent SetRole is assigned to a SetName if the contents of that SetName describe a concurrent state - transition system.

These groups of SetRoles are presented in figure 2.2. The list is by no means complete. Any new addition of SetRoles can be made in the overall grouping of Partition, Descriptive, Dominant and Transitive SetRoles.

Roles	*Partition*	*Dominant*	*Descriptive*	*Transitive*
SubRoles	Partition	Identity Identifier Hierarchy Aggregation Generalization	Qualifier Association Flow Guard Sequence Value	Sequential Concurrent

Figure 2.6. Summary of the InfoMap SetRoles

2.2.3 Levels of Abstraction

According to the opinions of knowledge engineers [17], knowledge in general is not organized around syntax, but rather around relationships among sets of objects. Furthermore, the context of these sets of objects and their relationships, should be as specific as the actual methodology used to describe them. Part of each methodology are levels of abstraction [17]. In order to represent any body of knowledge using sets and relationships, first we need to specify these levels of abstraction.

		Set Level	
		Between	**Within**
Relationship Level	**General**	*InfoSchema*	*General Partition*
	Detailed	*Detailed Partition*	*InfoMap*

Figure 2.7. Summary of the InfoMap Relationship Levels

Figure 2.7 shows the relationship levels among sets of objects. According to this scheme, we identify two levels of abstractions: general and detailed. Furthermore we identify two levels of relationships: between and within sets. Each pair combination of these four levels results in a relationship type:

- At the general level there exists a SetMemberRole relationship that specifies the association of SetMembers within their SetNames. This is called InfoSchema abstraction.

- At the detailed level between SetNames, we attribute a Partition SetRole in order to separate different views of SetMemberRole relationships. This is called Detailed Partition.

- At the general level within SetNames, we attribute a Partition SetRole in order to separate different views of SetRole relationships. This is called General Partition.

- At the detailed level there exists a SetMemberRole relationship that specifies the association of SetMembers within their SetNames. This is called InfoMap abstraction.

In our approach we analyze the above two levels of abstraction and the two levels of relationships to produce a generic model. This model may be applied to produce two views and two abstractions, which are the combinations of four relationship levels. The resulting two levels that are of interest to us are the general level or InfoSchema, and the detailed level or InfoMap [18].

2.3 Producing Representations

In this section we first present the definition of the context-free grammar. Then, we derive context-free grammars for the general and detailed levels of abstraction. Two processes are described that assist in the application of the context-free grammars. The first selects SetNames and attaches to them SetRoles at the general level, and the second relates the SetMembers of each SetName by attaching to them SetMemberRoles. Two algorithms are also presented. The first converts the results of the applied general level production rules into the tabular structure of an InfoSchema. The second algorithm performs the same for the production rules of the detailed level and produces the InfoMap tabular structure. Examples are also presented which trace the processes and algorithms, producing actual results. The process of deriving the tabular format from the initial concept we wish to model is shown in figure 2.8.

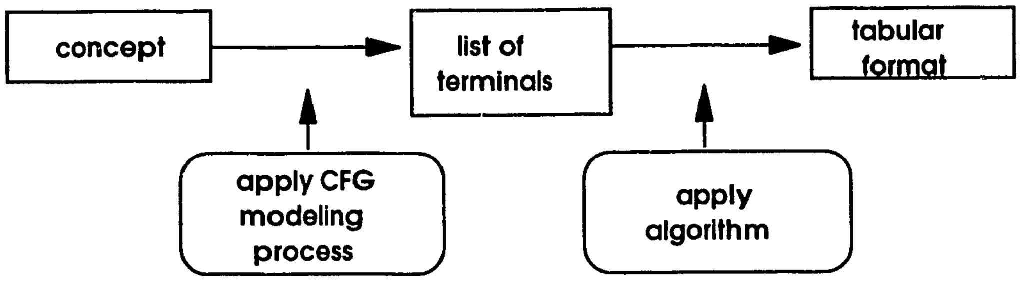


Figure 2.8. Modeling Process for Producing InfoMap Representations

2.3.1 Context-Free Grammar for the Methodology

A very powerful tool is presented to us: the idea that any concept can be presented in simple production rules based on an alphabet and its component symbols. When these rules are applied appropriately they derive a specific model. The same idea underlies the use of language generators and context-free grammars.

A **context-free grammar** G is a quadruple (V, Σ, R, S), where:

V is an alphabet

Σ (the set of **terminals**) is a subset of V,

R (the set of **rules**) is a finite subset of $(V \times V^*)$, and

S (the **start symbol**) is an element of $V - \Sigma$

The members of $V - \Sigma$ are called **non-terminals**. For any $A \in V - \Sigma$ and $u \in V^*$, we write $A \xrightarrow[G]{} u$ whenever $(A, u) \in R$. For any strings $u,v \in V^*$, we write $u \underset{G}{\Rightarrow} v$ if and only if there are strings $x, y, v' \in V^*$ and $A \in V - \Sigma$ such that $u = xAy$, $v = x\,v\,'y$, and $A \xrightarrow[G]{} v'$. Finally, $L(G)$, the **language generated** by G, is $\{w \in * : S \overset{*}{\underset{G}{\Rightarrow}} w\}$; we also say that G generates each string in $L(G)$ [16].

According to the definition of the context-free grammar we define the following:

V (alphabet): PartitionSg, PartitionSd,
Sg, SetRole, SetName, SetRoleDominant,
SetRoleDescriptive, SetRoleTransitive,
SetRoleUserDefined, SetRoleIdentifier,
SetRoleIdentity, SetRoleHierarchy,
SetRoleGeneralization, SetRoleAggregation,
SetRoleQualifier, SetRoleAssociation,
SetRoleFlow, SetRoleGuard, SetRoleSequence,
SetRoleValue, SetRoleSequential,
SetRoleConcurrent, **K, O,H, I, P, M, F, G, S, V, L, C,**
UpperCaseLetter, Sd, SetMemberRole, SetMember,
SetMemberRoleDominant, SetMemberRoleDescriptive,
SetMemberRoleTransitive, SetMemberRoleUserDefined,
SetMemberRoleIdentifier, SetMemberRoleIdentity,
SetMemberRoleHierarchy,
SetMemberRoleGeneralization,
SetMemberRoleAggregation, SetMemberRoleQualifier,
SetMemberRoleAssociation, SetMemberRoleGuard,
SetMemberRoleSequence, SetMemberRoleSequence,
SetMemberRoleValue, SetMemberRoleConcurrent,
SetMemberRoleSequential, **id, o, h, 1..n, p, c, x, a, k, v, u,**
o, t, F, T, f, s, d, a, l, e, literal, {, }

Σ (terminal):	The bold-faced characters that appear in the above alphabet.
V — Σ (non-terminal):	The unbolded characters that appear in the above alphabet.

Sg (Start Symbol for the general level grammar): PartitionSg
Sd (Start Symbol for the detailed level grammar): PartitionSd
R (the set of rules): is presented in the following sections.

In the next sections the context-free grammar production rules for both the general and detailed levels are presented. Alphabet(V) contents in brackets ({ }) may be repeated zero or more times. Alphabet(V) contents in square brackets ([ ]) may be presented zero or one time. Alphabet(V) contents in no brackets may be presented only once. Finally Alphabet(V) contents in bold, or in double quotes (" "), or in both, represent terminal symbols.

2.3.2 Context-Free Grammar for the General Level of Abstraction

At the general level of abstraction we describe the SetNames and their SetRoles. In that respect, we define the following production rules for the general level of abstraction. The first category of rules describes the main syntactical patterns, the sequences of SetRoles and SetNames for each Partition at the general level of abstraction (figure 2.9).

At this level SetRoles may be further decomposed into other categories. A SetRoleDominant SetRole may take the form of a unique Identifier or an Identity; it may represent a Hierarchical structure, a Generalization and/or an Aggregation structure. A SetRoleDescriptive SetRole may be broken down into a Qualifier (a one-to-one) relationship, an Association relationship (abstracted from any specific concept), a Flow relationship (object flow between entities), a Guard (conditions), a Sequence (set members represented in a sequence) or Value (a number, character, Boolean value). Finally a SetRoleTransitive SetRole may be replaced by its Sequential or Concurrent Control Flow SetRole. The SetRoleUserDefined SetRole is left for the user and/or the developer of new concepts to define. These production rules are shown in figure 2.10.

1	PartitionSg	→	{Sg}
2	Sg	→	{SetRole SetName}
3	SetRole	→	{SetRoleDominant | SetRoleDescriptive | SetRoleTransitive | SetRoleUserDefined}
4	SetName	→	"{"literal"}"

Figure 2.9. First Level of Context-Free Grammar Production Rules for the General Level of Abstraction

5	SetRoleDominant	→	SetRoleIdentifier |
			SetRoleIdentity |
			SetRoleHierarchy |
			SetRoleGeneralization |
			SetRoleAggregation
6	SetRoleDescriptive	→	SetRoleQualifier |
			SetRoleAssociation |
			SetRoleFlow |
			SetRoleGuard |
			SetRoleSequence |
			SetRoleValue
7	SetRoleTransitive	→	SetRoleSequential | SetRoleConcurrent

Figure 2.10. Second Level of Context-Free Grammar Production Rules for the General Level of Abstraction

8	SetRoleIdentifier	→	**K**
9	SetRoleIdentity	→	**O**
10	SetRoleHierarchy	→	**H**
11	SetRoleGeneralization	→	**I**
12	SetRoleAggregation	→	**P**
13	SetRoleQualifier	→	**X**
14	SetRoleAssociation	→	**M**
15	SetRoleFlow	→	**F**
16	SetRoleGuard	→	**G**
17	SetRoleSequence	→	**S**
18	SetRoleValue	→	**V**
19	SetRoleSequential	→	**L**
20	SetRoleConcurrent	→	**C**
21	SetRoleUserDefined	→	UpperCaseLetter

Figure 2.11. Third Level of Context-Free Grammar Production Rules for the General Level of Abstraction

We could be using words in order to represent relationships, but its is more convenient to replace them with upper-case letters. The list of production rules shown in figure 2.11 is used for this purpose. Also, we may expand this list in order to define new upper-case letters if new production rules are added in the previous list. The "UserDefined" production rule serves that purpose.

These production rules assist the modeling and developing of concepts based on the several categories and sub-categories of SetRoles we have identified so far at the general level of abstraction. A process for using this context-free grammar is presented along with an example of its application. This process can be partitioned into two main sub-processes. The first forms SetNames, and the second attaches SetRoles to the SetNames. This process is shown in figure 2.12.

[1] Identify the SetNames of the problem: [*s1*...*sn*].
[2] Apply 3rd production rule to select SetRoles [*r1*...*rn*] for [*s1*...*sn*]. One of [*r1*...*rn*] should be a dominant role.
[3] Apply 5th, 6th, 7th production rules to replace [*r1*...*rn*] with [*r1'*...*rn'*].
[4] Rewrite the right-hand side of the production rule " sg → {SetRole SetName} " with the SetNames [*s1*...*sn*] and the SetRoles [*r1'*...*rn'*].

Figure 2.12. Process for Applying the Production Rules at the General Level of Abstraction

We can apply the process shown in figure 2.12 to form the production rules at the general level of abstraction. Suppose we are presented with a list of student names, a list of their identification numbers and a list of their phone numbers. Identification numbers are assumed to be unique, students may have more than one phone number, and two students may share the same phone number(s). The relationship of student names, identification numbers and phone numbers at the general level of abstraction can be represented using the context-free grammar as follows:

```
PartitionSg  →   Sg                                  →
                 {SetRole SetName}                   →
                 SetRoleDominant {ID#}
                 SetRoleDescriptive {STUDENT}
                 SetRoleDescriptive {PHONE#}         →
                 SetRoleIdentity {ID#}
```

SetRoleQualifier {STUDENT}
SetRoleAssociation {PHONE#} →
O {ID#} X {STUDENT} M {PHONE#}

The application of the production rules of the context-free grammar at the general level of abstraction resulted in the following list of terminal symbols: **O {ID#} X {STUDENT} M {PHONE#}**. The following algorithm takes this list of terminal symbols as input, and converts it into a tabular representation (InfoSchema). We assume that each time a token is read the empty space following the token is automatically eliminated. Therefore the next token to be read contains no empty spaces. The algorithm is presented in figure 2.13.

```
convert_general_level(i)                                  [step]
        let columns "c1" "c2"                             [1]
        let "i" = 1                                       [2]
        repeat                                            [3]
                read token                                [4]
                temp1:=token                              [5]
                read token                                [6]
                temp2:=token                              [7]
                place temp1 in c1, row."i"                [8]
                place temp2 in c2, row. "i"               [9]
                let "i" := "i" + 1                        [10]
        until end of input j                              [11]
```

Figure 2.13. Algorithm for Producing the Tabular Representation at the General Level of Abstraction

The algorithm reads two tokens in a sequence from the input stream of tokens. The first is assumed to be the SetRole, and the second the SetName. It places the SetRole in the first column and the SetName in the second column. In each iteration it increments the row number by one so that each pair of SetRole, SetName appears in a different row. We can trace the above algorithm and create the tabular representation for the general level of abstraction. This trace is presented in figure 2.14. The input list is: **O {ID#} X {STUDENT} M {PHONE#}**.

C1	C2

steps [1] - [2]

C1	C2
O	{ID#}

C1	C2
O	{ID#}
X	{STUDENT}

C1	C2
O	{ID#}
X	{STUDENT}
M	{PHONE}

steps [4] - [10]

Figure 2.14. Example Application of the Algorithm in Figure 2.13

The end result after tracing the algorithm is a tabular representation of a Partition of SetNames and SetRoles at the general level of abstraction. It describes the relationship between three SetNames. The relationship between these SetNames can be read as: each SetMember of the SetName {ID#} corresponds to one SetMember of the SetName {STUDENT}, and it is associated with SetMembers from the SetName {PHONE#}. This interpretation is based on the description of the valid SetRoles in section 2.2.2.

2.3.3 Context-Free Grammar for the Detailed Level of Abstraction

The detailed level of abstraction may be viewed as an extension of the general level of abstraction. The difference is that at the detailed level of abstraction the SetMembers, along with the SetMemberRoles, are revealed. Therefore, in order to abstract the general level of abstraction from the detailed level of abstraction, all we need to do is to hide the two attributes that do not contribute to the general level of abstraction: SetMemberRole and SetMember. Also, in order to expand the general level of abstraction and derive the detailed level of abstraction, we only need to add these two attributes back to the general level of abstraction.

The production rules that follow are broken down into categories. The first category describes the main syntactical patterns that relate the four attributes described in section 2.2.1. These syntactical patterns are a collection of Partitions at the detailed level of abstraction. This is shown in figure 2.15.

22	PartitionSd	→	{Sd}
23	Sd	→	{{SetMemberRole SetMember}}
24	SetMember	→	**{literal}**

Figure 2.15. First Level of Context-Free Grammar Production Rules for the Detailed Level of Abstraction

The production rules in figure 2.15 specify the cardinality of the model. These syntactical patterns are extended in figure 2.16 in order to further analyze the SetMemberRole attribute.

25	SetMemberRole	→	{SetMemberRoleDominant	|
			SetMemberRoleDescriptive	|
			SetMemberRoleTransitive	|
			SetMemberRoleUserDefined}	

Figure 2.16. Second Level of Context-Free Grammar Production Rules for the Detailed Level of Abstraction

The categories in figure 2.16 can be further decomposed. A similarity exists between this decomposition and the one presented at the general level in section 2.3.2. However in this case, lower-case letters correspond to each SetMemberRole. Figure 2.17 shows these categories and the lower case letters.

26 SetMemberDominant	→	SetMemberRoleIdentifier | SetMemberRoleIdentity | SetMemberRoleHierarchy | SetMemberRoleGeneralization | SetMemberRoleAggregation
27 SetMemberDescriptive	→	SetMemberRoleQualifier | SetMemberRoleAssociation | SetMemberRoleFlow | SetMemberRoleGuard | SetMemberRoleSequence | SetMemberRoleValue
28 SetMemberTransitive	→	SetMemberRoleConcurrent | SetMemberRoleSequential
29 SetMemberRoleIdentifier	→	**id**
30 SetMemberRoleIdentity	→	**o**
31 SetMemberRoleHierarchy	→	**h** | **[1..n]**
32 SetMemberRoleGeneralization	→	**p** | **c**
33 SetMemberRoleAggregation	→	**w** | **c** | **v** | **h** | **m**
34 SetMemberRoleQualifier	→	**x**
35 SetMemberRoleAssociation	→	**k** | **v**
36 SetMemberRoleFlow	→	**u** | **o**
37 SetMemberRoleGuard	→	**t** | **f** | **T** | **F**
38 SetMemberRoleSequence	→	**1..n**
39 SetMemberRoleValue	→	RowIdentifier
40 SetMemberRoleSequential	→	**s** | **d** | **a** | **l** | **e**
41 SetMemberRoleConcurrent	→	**c**

Figure 2.17. Third Level of Context-Free Grammar Production Rules for the Detailed Level of Abstraction

The context-free grammars for the general and detailed levels of abstraction can be used in order to model concepts. Next, an application of the detailed level of abstraction of production rules is presented. A process similar to the one in figure 2.12 is also presented. It shows the forming of production rules at the detailed level of abstraction. Furthermore, an algorithm that converts the results of the application of these production rules into a tabular structure is presented.

[1] List the SetMembers [*sm1...smn*] of the SetNames [*s1...sn*].
[2] Apply production rule 25 to select generic SetMemberRoles [*smr1...smrn*] for the SetMembers [*sm1...smn*] of the SetNames [*s1...sn*].
[3] Apply production rules 26 to 27 and replace [*smr1...smrn*] with *[smr1'...smrn']* .
[4] Apply production rules 29 to 41 and replace *[smr1'...smrn']* with [*smr1''...smrn''*].
[5] Rewrite production rule Sd → {{SetMemberRole SetMember}} with [*smr1''...smrn''*] for SetMemberRole and [*sm1...smn*] for SetMembers.

Figure 2.18. Process for Applying the Production Rules at the Detailed Level of Abstraction

Assuming the general level of abstraction production rules have been applied, all that remains to be done is the application of the production rules at the detailed level of abstraction. In order to form rules at the detailed level of abstraction, we can apply the process that is shown in figure 2.18.

An initial description of the problem was presented in section 2.3.2. In this section the SetMembers are added along with their SetMemberRoles. The SetMembers and their SetRoles are informally described as follows: John has ID# 123 and can be reached at phone# 555-1234, George has ID# 234 and can be reached at phone# 555-2345 and 555-3456, Kelly has ID# 345 and can be reached at phone# 555-7890, Kathy has ID# 456 and can be reached at phone# 555-7890 and 555-3214. These SetMembers and SetRoles may be represented by applying the context-free grammar at the detailed level of abstraction as follows:

Sd → *{{SMR SM}}* →
SMRDominant SM SMRDescriptive SM SMRDescriptive SM
SMRDominant SM SMRDescriptive SM SMRDescriptive SM
SMRDominant SM SMRDescriptive SM SMRDescriptive SM
SMRDominant SM SMRDescriptive SM SMRDescriptive SM
→
SMRIdentifier SM SMRQualifier SM SMRAssociation SM
SMRIdentifier SM SMRQualifier SM SMRAssociation SM
SMRIdentifier SM SMRQualifier SM SMRAssociation SM
SMRIdentifier SM SMRQualifier SM SMRAssociation SM

$\rightarrow$

o 123 x John v 555-1234
o 234 x George v 555-2345 v 555-3456
o 345 x Kelly v 555-7890
o 456 x Kathy v 555-3456 v 555-3214

In the above application of the production rules the prefix SM is a SetMember, and the prefix SMR is a SetMemberRole.

In order to produce a tabular structure based on the derived list of tokens at the detailed level of abstraction, the tabular structure that was derived at the general level of abstraction is used. Since at the general level of abstraction the SetNames and their SetRoles are already identified and placed in the appropriate positions, they can be used as a base in order to expand the rows and columns. This expansion depends on the input stream of tokens that were produced by the application of the production rules at the general level of abstraction. The algorithm is presented in figure 2.19.

```
convert_detail_level(j)                                              Step
"a" := current SetName                                               [1]
read sequence of pairs [{SetMemberRole}, {SetMember}]                [2]
        until end of input
for each pair of [{SetMemberRole}, {SetMember}]                      [3]
        clone C1 ;clone: insert new column on the right side of C1   [4]
        if the SetMember exists                                      [5]
                place SetMemberRole in cloned C1 under "a",          [5.1]
                row = row in which SetMember exists
        else                                                         [6]
                insert row under "a", place SetMemberRole in new row [6.1]
                update "a" to the next SetName                       [6.2]
        let cloned C1 = current C1                                   [7]
```

Figure 2.19. Algorithm for producing the Tabular Representation at the Detailed Level of Abstraction

The output of the algorithm at the general level of abstraction is a tabular structure. Using this tabular structure, the algorithm in figure 2.19 and the input stream of terminal symbols produced with the detailed level context-free grammar, we show the results in figure 2.20.

In the 1st iteration the first sequence of {SetMemberRole}, {SetMember} is processed (step: 4, sequence: "o 123 x John v 555-1234"). Since its SetMembers do not exist under the given SetNames (i.e.: ID#, STUDENT, PHONE#) they are inserted under their respected SetNames. Also, the SetRoles that are present in this sequence are also placed under their respected SetRoles (steps 8 - 9).

In the 4th iteration the sequence of "o 234 x George v 555-2345 v 555-3456" is processed. The SetMember "555-3456" exists under the SetName {PHONE#}. Therefore steps 6 - 7 are executed. This clause guarantees the uniqueness of the SetMembers under each SetName.

2.4 Summary, Deliverables

The methodology described in this chapter is based on sets and relationships among sets. Specific roles are attached to sets in order to relate them to other sets and interrelate their set members. Furthermore, these roles are categorized according to their role types. These role types may be used to describe dominant, descriptive and transitive set role types.

We have shown how we can use a context-free grammar to describe the general and detailed levels of abstraction. Two tabular structure formats were derived from two context-free grammars. This is accomplished by:

- Following processes for the application of context-free grammars and the derivation of terminal symbols.

- Applying algorithms to convert the list of terminal symbols into a tabular structure that preserves the uniqueness of the set members of a set.

The conclusions based on this prototype attempt to describe the InfoMap methodology are:

- It is possible to construct compilers for both the general and detailed formats by using the context-free grammars that were defined for both levels of abstraction [30].

- It is possible to use compilers to produce the same specifications of a system in some format other than tabular.

- The sets and relationships that underline this approach can be modeled using a very simple context-free grammar.

- Practice shows that the tabular format derived from the context-free grammar is easily updated and analyzed [18].

- Simple operations with limited functionality may be used within the context of a database program to generate high level language code.

C1			C2
O	O		{ID#}
o			123
X	X		{STUDENT}
x			John
M	M		{PHONE #}
v			555-1234

1st iteration
(steps 1, 2, 3, 4, 6, 6.1, 6.2, 7)

C1				C2
O	O	O		{ID#}
o				123
	o			234
X	X	X		{STUDENT}
x				John
	x			George
M	M	M		{PHONE #}
v				555-1234
	v			555-2345
	v			555-3456

2nd iteration
(steps 1, 2, 3, 4, 6, 6.1, 6.2, 7)

C1					C2
O	O	O	O		{ID#}
o					123
	o				234
		o			345
X	X	X	X		{STUDENT}
x					John
	x				George
		x			Kelly
M	M	M	M		{PHONE #}
v					555-1234
	v				555-2345
	v				555-3456
		v			555-7890

3rd iteration
(steps 1, 2, 3, 4, 5, 5.1, 7)

C1						C2
O	O	O	O	O		{ID#}
o						123
	o					234
		o				345
			o			456
X	X	X	X	X		{STUDENT}
x						John
	x					George
		x				Kelly
			x			Kathy
M	M	M	M	M		{PHONE #}
v						555-1234
	v					555-2345
	v		v			555-3456
		v				555-7890
			v			555-3214

4th iteration
(steps 1, 2, 3, 4, 5, 5.1, 7)

Figure 2.20. Example Application of the Algorithm in Figure 2.19

CHAPTER 3: THE DESIGN FRAMEWORK

3.1 Introduction, Motivation

One of the reasons that the object model is becoming more popular is that design reuse is becoming more and more important [19]. A study suggests that 60 to 85 percent of the total cost of software is due to maintenance [20]. Frameworks for design and implementation are suggested as a way to reduce costs. To emphasize this point, Kent Beck [21] suggests that programmers who can go a step further and convert their procedural solutions to a particular problem into a generic library are rare and valuable. The same can be said for software designers. However, software designers need more than intuition and experience to perform their jobs. Design frameworks provide the edge that software designers need to excel in their efforts.

A design framework is a set of prefabricated concept design building blocks that designers can use, extend, and customize for specific computing and/or general problem solutions [20]. With design frameworks, designers do not need to start from scratch each time they design a concept. Frameworks are built from a collection of objects, just like the SetRoles we described in chapter two. Therefore the design that is applied onto these SetRoles can easily be reused. Designing and adapting design frameworks in order to solve particular problem domains is a task that helps developers to provide solutions for problem domains and better maintain these solutions. If a framework is created in such a way that the new pieces will fall into place perfectly, the minimum impact of any change will reduce costs dramatically [22].

Our objective is to provide a design framework for the InfoMap methodology for three main reasons.

- First, we have realized that the InfoMap methodology needs such a reusable framework, in order to ease the development of any new concept that can assist any design application domain.

- Second, we have realized that by providing the design for such a framework, we will better understand ourselves, and give others a better understanding of our model.

- Third, as we will show in the next chapter, the design of our framework will give us the opportunity to use it in order to model design patterns that exist in software [1].

In the previous chapter we described the basics of the InfoMap approach: sets, relationships among sets and abstraction levels. We also described how a context-free grammar may be defined, used to model concepts and convert these into in a tabular structured representation. In this chapter we discuss specific modeling issues involved in the representation of the InfoMap model. These issues are described in the context of a design framework, which may be used as a design tool for the modeling of design concepts. Formal specifications of this framework are generated by the Rational Inc. CASE tool [15], designed to support the methodology introduced by Grady Booch [23].

3.2 Framework Structure, Domain and Classification

The design of a framework differs from the developing process of a standalone high-level architectural design for any application. The success of our framework solves problems that on the surface are quite different from the problem that justified its creation. Early on, during the design of tools and processes for the manipulation of our representation, we realized that the application development with the InfoMap methodology is not only costly to build but impossible to maintain (*i.e.*: InfoFarm [18]). Nevertheless, the problem lies in solving expertise that must be captured so that it is independent from both the original entity design and the future, possibly unknown, solutions that it will provide. In this section we briefly discuss other frameworks, the class of problems that they address, and our design framework's classification.

There are many successful framework designs. Most of them deal with implementation issues. Among the most popular ones is the MacApp framework for Macintosh applications [24]. An abstract MacApp application consists of one or more windows, or one or more documents. Furthermore, each of them consists of other sub-classes that provide a complete and concrete Macintosh application development environment. In addition to the MacApp, other successful frameworks include the Lisa Toolkit [25], which is used to develop applications for the Lisa desktop environment. However the most successful of all, according to several experts' opinions, is the Smalltalk Model/View/Controller (MVC), a framework for constructing Smalltalk-80 user interfaces [26].

All these provide ways of reusing code and designs which are resistant to more conventional reuse attempts. Application-independent components may be rather easily reused, depending on whether the application they tackle belongs in the same domain as the framework that was developed to solve a particular problem. In other words, the user of the end product, the framework, must consider whether the application he or she needs to develop is a subset of the application framework. Therefore, the user must think first about classifying his or her design into one of the following categories: Framework domains and framework structures.

The problem domain can be chosen among application functions, domain functions or support functions [27]:

Application frameworks include expertise applicable to a wide range of programs. These frameworks are characterized by functionality that can be applied across application domains, therefore making them more general. Current commercial graphical user interfaces (GUI) are examples of such frameworks.

Domain frameworks include expertise in a particular problem domain. These frameworks are characterized by functionality for a particular problem domain. Examples of these domain frameworks are control systems for manufacturing or securities trading.

Support frameworks provide system services such as file access, distributed computing or device drivers. These frameworks are used in order to implement modifications and/or additions on existing systems, such as new file systems or device drivers.

Frameworks can also be classified according to their structure [27]:

Automatic frameworks are described as manager-driven, with a single controlling mechanism that triggers most of their actions. As soon as a call is made to the controlling function, the appropriate calls are made to other functions in order to perform a specific task. An application framework such as a graphical user interface generally uses a manager to receive input events from the user, and distribute them to the other objects in the framework.

Usage frameworks provide yet another structural classification. How we use classes of objects, derive new classes of objects, instantiate the objects or combine classes of objects depends on how the framework is designed for the specific application domain usage. This usage is further decomposed into data-driven and architecture-driven usage. Data-driven usage relies primarily on object composition for customization. The users customize the behavior of the objects by combining different objects. On the other hand, architecture-driven usage relies on inheritance for customization. The users of the framework customize its behavior by deriving new classes from the framework and overriding member functions.

The InfoMap framework may be used to model the expertise in a particular problem domain. The problem domain that this framework addresses is the modeling of design concepts in a specific, non-ambiguous fashion. Therefore, the domain of this framework can be classified as an application design domain framework. For example, the design of a process or a hierarchical structure addresses two different problem domains that are used for different purposes. A process is used to specify the steps followed in the solution of a problem, while a hierarchy is used to specify the top-down or *vice versa* arrangement of a structure [28]. Therefore, the domain upon which either the process or the hierarchy depends, is critical on the application that it addresses.

The structure of the InfoMap design framework is based on how the framework is used. Therefore, its structure is classified as a usage structure. How SetRoles and SetMemberRoles are used depends on how the framework is designed for the specific concept we wish to model. For example, we use different SetRoles to model the design of a process than to model the design of a hierarchy. This usage relies primarily on inheritance for customization.

3.3 The Design Framework

The product of the framework design process is a list of class definitions. Each class is composed of a list of operations that interact with the objects in the class in order to produce results. According to R. Johnson [20], we are faced with two major tasks. First we need to specify the list of class definitions. In order to do this we need to follow some specific rules. Therefore we need to further define our framework by following this set of rules. Two set of rules are presented, for classes and for frameworks. The following rules were applied in the design of the InfoMap framework:

Classes:

- Class hierarchies should be deep and narrow.
- The top of the class hierarchy should be abstract.
- Subclasses should be specializations.

Frameworks:

- Split large classes.
- Separate methods that do not inter-communicate.

In the following sections we describe the InfoMap design framework by defining its classes. These classes and their interaction are the result of the application of certain heuristics on the context-free grammar described in chapter two. The operations contained in each class are also described. We examine the application of the above rules for classes and frameworks. Finally we produce formal specifications for the classes and operations using the Rational Inc. CASE tool.

3.3.1 Assumptions

The tabular representation described in chapter two may be viewed as a collection of arrays. Each array is reserved for an attribute described in chapter two. As shown in figure 3.1.a, ***array [1]*** is reserved for SetMemberRoles, ***array [2]*** is reserved for SetRoles, ***array [3]*** is reserved for SetMemberNames and ***array [4]*** is reserved for the SetNames. In case more than one SetName appears in the modeling of any partition, these four arrays are repeated as shown in figure 3.1.b. Furthermore, all the arrays shown in figure 3.1 share a common characteristic: they may be divided into rows and columns. The following records describe the position of SetName, SetRole, SetMember, SetMemberRole in the arrays shown in figure 3.1:

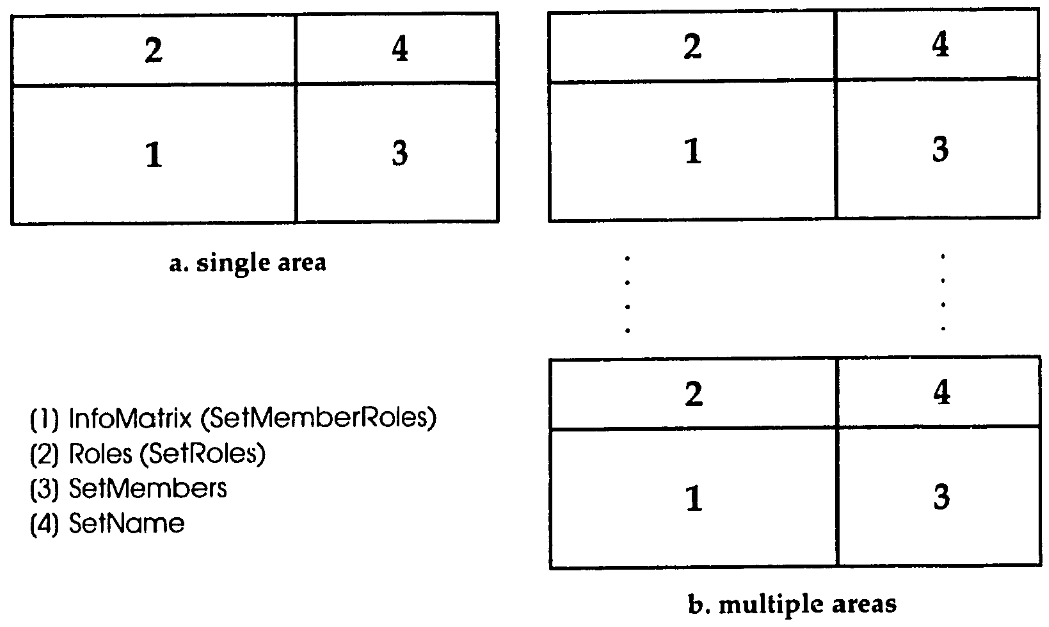


Figure 3.1. InfoMap Tabular Structure Arrays

- a triple "[array].row.column" represents a specific array's coordinates in terms of its row and column.

- a tuple "[array].row" represents a specific array's coordinates in terms of its row.

- a tuple "[array].column" represents a specific array's coordinates in terms of its column.

3.3.2 Creation of Classes

We may apply certain heuristics to the context-free grammar described in chapter two, and derive a class hierarchy. The heuristics for each production rule are the following:

[1] For the left-hand side non-terminal of each production rule an abstract class is created. Example: A $\rightarrow$ B, A is created as an abstract class.

[2] The non-terminals that appear on the right-hand side become the children classes of the abstract class which appears on the left-hand side. Example: A $\rightarrow$ B C, B and C become children classes of class A.

[3] The terminals become instances of the non-terminal (class) that appears on the left-hand side of a production rule. Example: C $\rightarrow$ **t f**, **t** and **f** are instances of class C.

[4] When a non-terminal appears in "{}", it is aggregated with a multiplicity of association *"many"* (zero or more). Example: A $\rightarrow$ {B}, zero or more "B", where "B" can be a class or an instance.

[5] When a non-terminal appears in "[]", it is aggregated with a multiplicity of association *"optional"* (zero or one). Example: A $\rightarrow$ [B], zero or one "B", where "B" can be a class or an instance.

[6] When a non-terminal appears in no brackets, it is aggregated with a multiplicity of association *"Exactly one"*. Example: A $\rightarrow$ B, only one "B", where "B" can be a class or an instance.

[7] When there exists an "or" (|) between two non-terminals, they are both in an inheritance relationship with the non-terminal that appears on

the left-hand side of the production rule. Example: A → B | C, classes "B" and "C" inherit from class "A".

[8] When no "or" (|) exists between two non-terminals, then an "ordered" relationship exists between the two. Example:
A → B C, instances of class "C" follow instances of class "B" in order "B" "C".

Therefore the context-free grammar described in chapter two can be converted into a class hierarchy. This is shown in figures 3.2 to 3.8. The diagrammatic notation is based on the Object Model Technique [29].

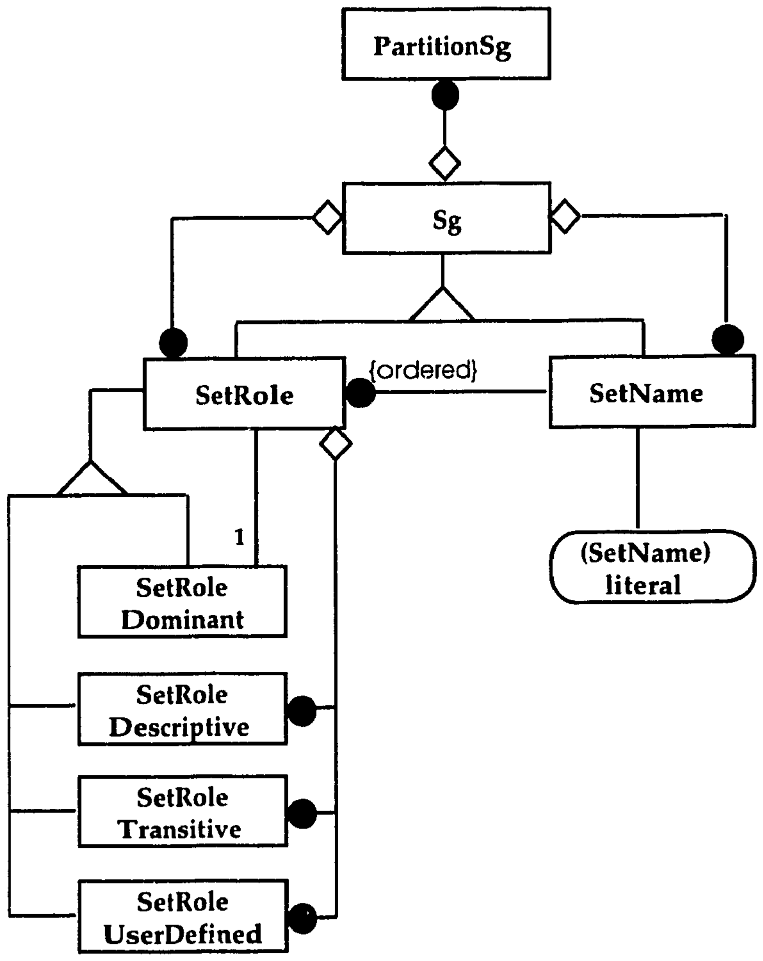


Figure 3.2. InfoMap General Level Framework Partition Corresponding to Production Rules 1, 2, 3 and 4 of figure 2.9

Figure 3.2 shows the class inheritance and multiplicity of associations between the classes derived from production rules 1, 2, 3, and 4 at the general level of abstraction. Heuristics 1, 2, 3, 4, 6, 7, and 8 where applied to transform the production rules of the context-free grammar at the general level of abstraction in the design that appears in figures 3.2 to 3.5. The class hierarchy described in figure 3.2 is deep and narrow. The top of the class (PartitionSg, SetRole) is an abstract class. Therefore the top of the hierarchy has no instances but its descendants do. The subclasses (SetRoleDominant, SetRoleDescriptive, SetRoleTransitive, SetRoleUserDefined) inherit the SetRole class are specializations. Their specializations and instances are shown in figures 3.3 - 3.5. The SetRoleUserDefined class is left to the developer of new SetRoles to define.

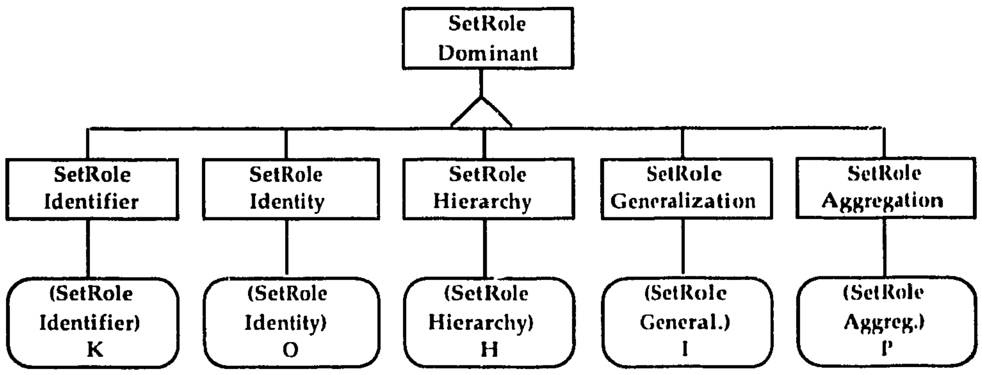


Figure 3.3. InfoMap General Level Framework Partition Corresponding to Production Rule 5 of figure 2.10 and Rules 8 to 12 of figure 2.11

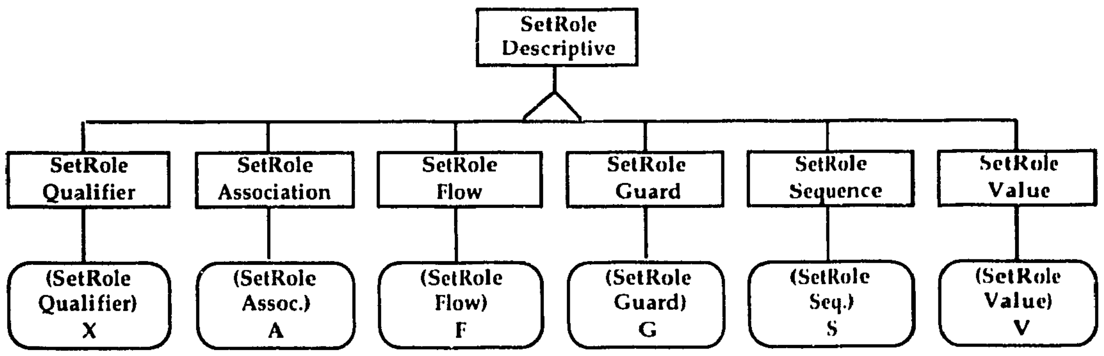


Figure 3.4. InfoMap General Level Framework Partition Corresponding to Production Rule 6 of figure 2.10 and Rules 13 to 18 of figure 2.11

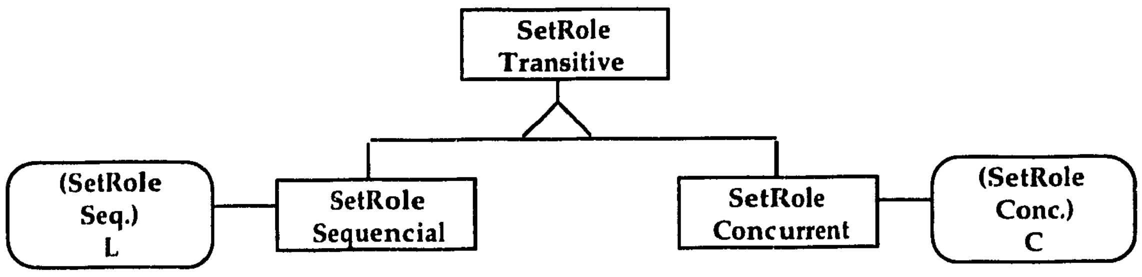


Figure 3.5. InfoMap General Level Framework Partition Corresponding to Production Rule 7 of figure 2.10 and Rules 20 and 21 of figure 2.11

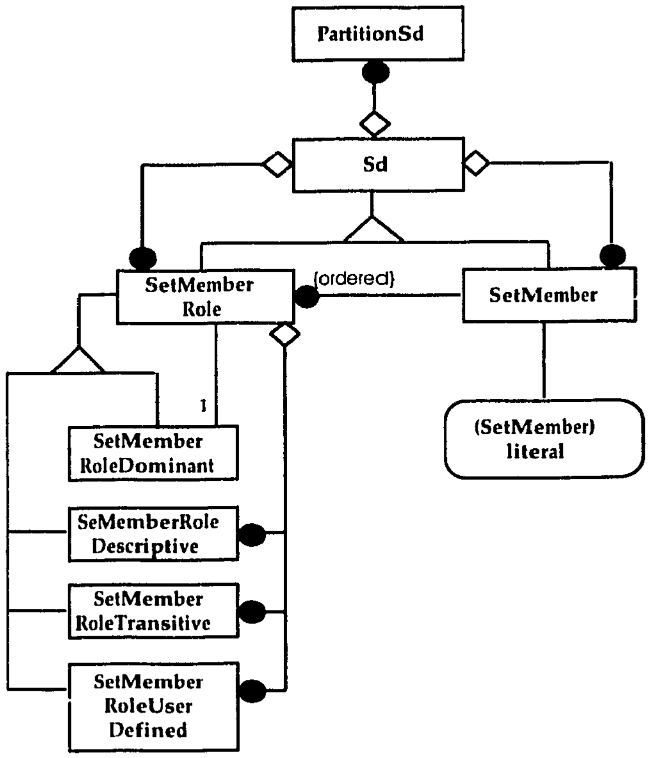


Figure 3.6. InfoMap Detailed Level Framework Partition Corresponding to Production Rules 22 to 24 of figure 2.15 and Rule 25 of figure 2.16

A structure similar to the one that appears in figures 3.3 to 3.5 exists at the detailed level of abstraction. The class hierarchy described in figure 3.6 is narrow, and the top of the class hierarchy (PartitionSd, SetMemberRole) is abstract. Therefore the top of the hierarchy has no instances but its descendants do, just like the case at the general level of abstraction. Heuristics 1, 2, 3, 4, 5, 6, 7 and 8 were applied to produce the class hierarchies from the context-free grammar at the detailed level of abstraction in the design that appears in figures 3.6 to 3.8.

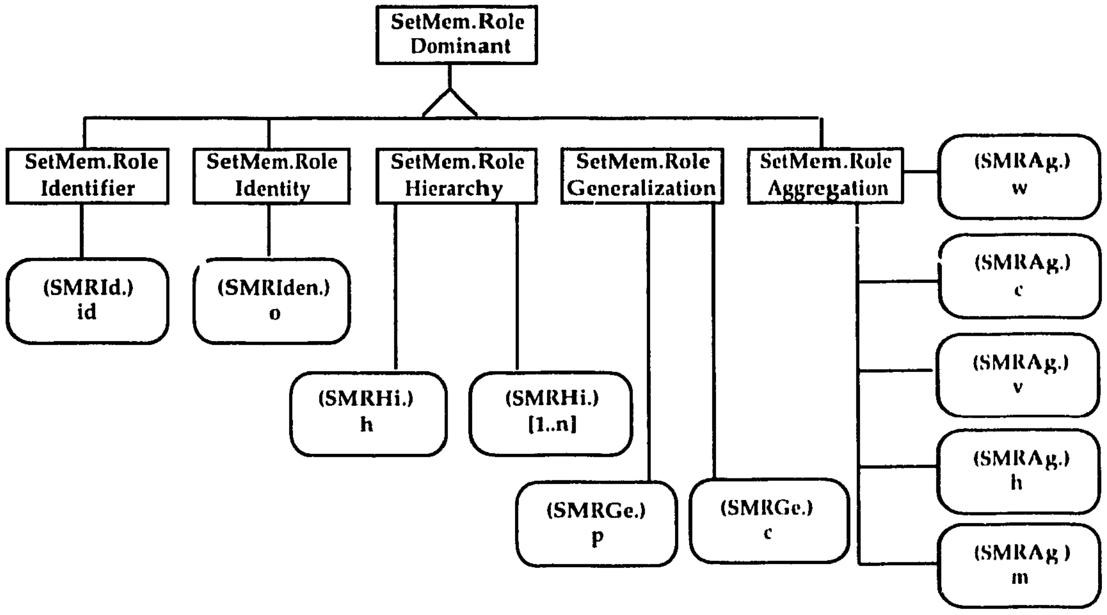


Figure 3.7. InfoMap Detailed Level Framework Partition Corresponding to Production Rules 26 and 29 to 33 of figure 2.17

The subclasses at the detailed level (SetMemberRoleDominant, SetMemberRoleDescriptive, SetMemberRoleTransitive, SetMemberRoleUserDefined) that inherit the SetMemberRole class are also specializations. Their specializations and instances are shown in figures 3.6 - 3.8. The SetMemberRoleUserDefined class at the detailed level is also left to the developer of new SetMemberRoles to define.

The approach followed to transform the context-free grammar into a framework is described in the literature as a design pattern [1]. In the next chapter we describe what design patterns are, and how they can be used and modeled using the InfoMap methodology. The description of a similar approach is presented next.

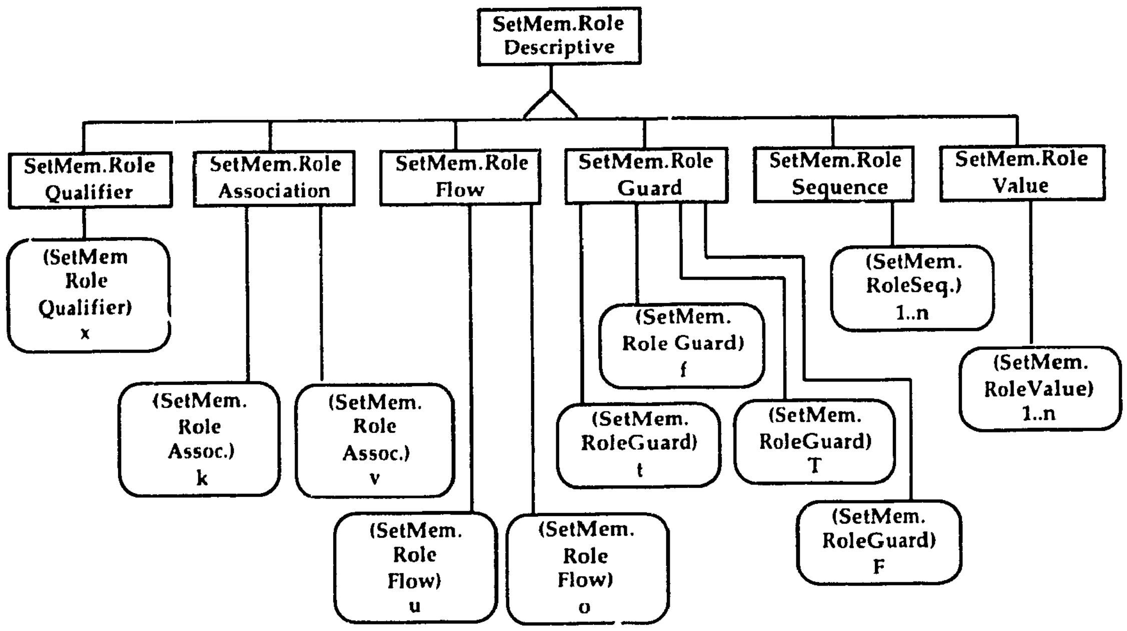


Figure 3.8. **InfoMap Detailed Level Framework Partition Corresponding to Production Rules 27 and 34 to 39 of figure 2.17**

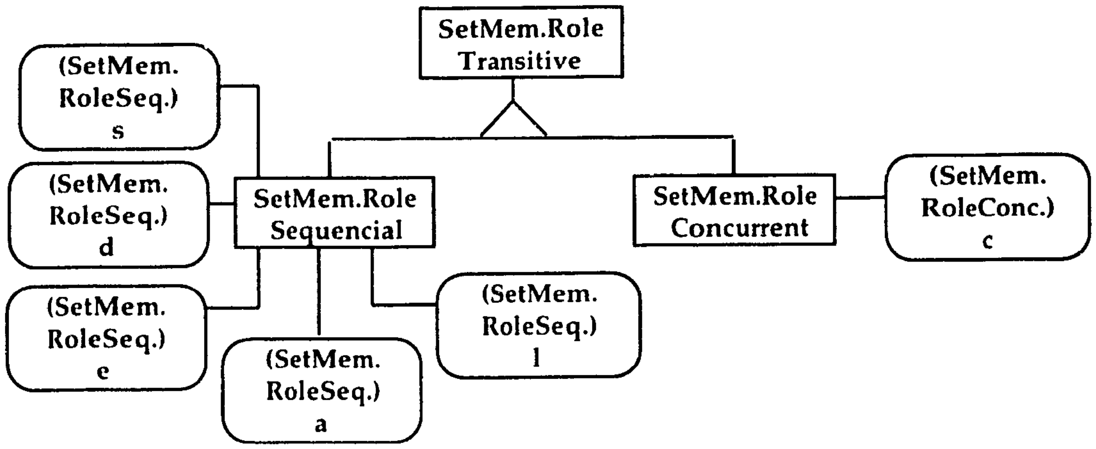


Figure 3.9. **InfoMap Detailed Level Framework Partition Corresponding to Production Rules 28, 40, and 41 of figure 2.17**

The problem is to define a representation of a grammar, along with an interpreter that uses the representation. The solution is to simply transform the left-hand side of each regular expression into a class, and the right-hand side as instances of that class. This, however, does not take into consideration the multiplicity of associations that exists in any design composed of classes and instances of these classes. In addition to that, as the authors of this design pattern agree, the context-free grammar should be very simple. This problem has been successfully applied in SPECTalk to interpret descriptions of input file formats, and in the QOCA constraint-solving toolkit which uses this design pattern to evaluate constraints [1].

3.3.3 Framework Operations

After describing the heuristics that transform the context-free grammar described in chapter two, we applied them and produced the framework design shown in figures 3.2-3.8. These figures describe the classes defined in the framework. Furthermore, they describe the class inheritance and the multiplicity of association that exists between classes. In this section we examine which operations can be included in the classes that constitute the framework. For each class identified in figures 3.2 to 3.8, there exists a set of operations. Each operation is defined in terms of what exists and what does not exist in arrays 1, 2, 3 and 4 described in section 3.3.1. Therefore the main purpose of the four arrays is to further define the syntactical patterns that may be present in the tabular structured format of the InfoMap representation model.

The general level operations are described for the SetRoles and the SetNames. Following the name of each operation a brief description is given.

PartitionSg	operation: **exists** PartitionSg. (defined as the universal quantifier operation for an existing partition at the general level)
Sg	operation: **exists** ITEM or SetRole. (defined as the universal quantifier operation for an existing ITEM and/or SetRole in a given array [1], [2], [3], [4])
SetRole	operation: **exists** upper-case letter. (An upper case letter exists for a SetRole.)

SetName	operation: **exists** literal in array [4].row.column. (A literal exists for a SetName in array reserved for SetNames.)
SetRoleDominant	operation: **exists** generic Dominant upper-case letter in array [2]. (A generic Dominant upper-case letter exists in array [2]: Roles.)
SetRoleDescriptive	operation: **exists** generic Descriptive upper case letter in array [2]. (A generic Descriptive upper-case letter exists in array [2]: Roles.)
SetRoleTransitive	operation: **exists** generic Transitive upper case letter in array [2]. (A generic Transitive upper-case letter exists in array [2]: Roles.)
SetRoleIdentity	operation: **exists** "O" in array [2]. (SetRole "O" exists in array [2].)
SetRoleIdentifier	operation: **exists** "K" in array [2]. (SetRole "K" exists in array [2].)
SetRoleHierarchy	operation: **exists** "H" in array [2]. (SetRole "H" exists in array [2].)
SetRoleAggregation	operation: **exists** "P" in array [2]. (SetRole "P" exists in array [2].)
SetRoleGeneralization	operation: **exists** "I" in array [2]. (SetRole "I" exists in array [2].)
SetRoleQualtifier	operation: **exists** "X" in array [2]. (SetRole "X" exists in array [2].)

SetRoleAssociation	operation: **exists** "M" in array [2]. (SetRole "M" exists in array [2].)
SetRoleFlow	operation: **exists** "F" in array [2]. (SetRole "F" exists in array [2].)
SetRoleGuard	operation: **exists** "G" in array [2]. (SetRole "G" exists in array [2].)
SetRoleSequence	operation: **exists** "S" in array [2]. (SetRole "S" exists in array [2].)
SetRoleValue	operation: **exists** "V" in array [2]. (SetRole "V" exists in array [2].)
SetRoleSequential	operation: **exists** "L" in array [2]. (SetRole "L" exists in array [2].)
SetRoleConcurrent	operation: **exists** "C" in array [2]. (SetRole "C" exists in array [2].)

Whereas at the general level operations are described for the SetRoles and the SetNames, at the detailed level operations are described for SetMemberRoles and SetMemberNames. Following the name of each operation a brief description is given.

PartitionSd	operation: **exists** PartitionSd. (defined as the universal quantifier operation for an existing partition at the detailed level)
Sd	operation: **exists** SetMemberRole / ITEM. (defined as the universal quantifier operation for an existing ITEM and/or SetMemberRole in a given array [1], [2], [3], [4])

SetMemberRole	operation: **exists** lower-case letter. (A lower-case letter exists for a SetMemberRole.)
SetMemberName	operation: **exists** literal in array [3].row.column. (A literal exists for a SetMemberName in the array reserved for SetMemberNames.)
SetMemberRoleDominant	operation: **exists** generic lower-case letter in array [1]. (A generic Dominant lower-case letter exists in array [1]: Roles.)
SetMemberRoleDescriptive	operation: **exists** generic lower-case letter in array [1]. (A generic Descriptive lower case letter exists in array [1]: Roles.)
SetMemberRoleTransitive	operation: **exists** generic lower-case letter in array [1]. (A generic Transitive lower-case letter exists in array [1]: Roles.)
SetMemberRoleIdentity	operation: **exists** unique "o" in array[1].column. (SetMemberRole "o" is dominant in its column)
SetMemberRoleIdentifier	operation: **exists** number in array [1].column. (A numeric value is dominant in its column.)
SetRoleHierarchy	operation: **exists** unique "h" in array [1].column. operation: **exists** number in array [1].column. (The letter "h" represents the root of a hierarchy in a column, and the numeric value of its children.)
SetRoleAggregation	operation: **exists** unique "w" in array [1].column. operation: **exists** "c" in array [1].column. operation: **exists** "v" in array [1].column. operation: **exists** "h" in array [1].column.

operation: **exists** "m" in array [1].column.
(The letter "w" represents the "whole" of an aggregation relationship, the letter "c" the "part-of", the letter "v" the visibility, the letter "h" the non-visibility and the letter "m" anything else.)

SetRoleGeneralization operation: **exists** unique "p" in array [1].column.
operation: **exists** "c" in array [1].column.
(The letter "p" represents the "parent" in a generalization, and the letter "c" its children.)

SetRoleQualifier operation: **exists** unique "x" in array [1].column.
(The letter "x" exists in [1].column as a qualifier.)

SetRoleAssociation operation: **exists** "v" in array [1].column.
(The letter "v" exists in [1].column as an association)

SetRoleFlow operation: **exists** "u" in array [1].column
operation: **exists** "o" in array [1].column
(The letter "u" represents user input in array [1].column, and the letter "o" represents user output in array [1].column)

SetRoleGuard operation: **exists** "t" in array [1].row.column **xor**
ITEM = <true> in array [3].row.
operation: **exists** "f" in array [1].row.column **xor**
ITEM = <false> in array [3].row.
operation: **exists** "T" in array [1].row.column **xor**
ITEM = <complementary of t> in array [3].row
operation: **exists** "F" in array [1].row.column **xor**
ITEM = <complementary of f> in array [3].row.
(The letters in array [1].row.column are evaluated with an xor operator against the SetMemberNames in

array[3].row, which are defined as ITEMS. In this case
ITEMS represent expressions.)

SetRoleSequence operation: **exists** number in array [1].column.
(A numeric value is present in array [1].column.)

SetRoleValue operation: **exists** number/string in array [1].column
(A number and/or string is present in array [1].column.)

SetRoleSequential operation: **exists** unique "s" in array [1].column.
operation: **exists** "d" in array [1].column.
operation: **exists** unique "l" in array [1].column.
operation: **exists** unique "a" in array [1].column.
operation: **exists** unique "e" in array [1].column.
(The letter "s" represents a source state, the letter "d" a destination state, the letter "a" an assertion state, and the letter "e" an exemption state.)

SetRoleConcurrent operation: **exists** "c" in array [1].column.
(The letter "c" represents concurrent states in array [1].column.)

3.4 Class and Framework Rules Applied

According to R. Johnson [20], several rules of thumb can help the development of frameworks. These rules were applied in the InfoMap framework design. On the other hand, these rules were also automatically implied by the heuristics of transforming a context-free grammar into a framework. In this section we review the rules and their application. The review is broken down into class - and framework - related rules.

For the framework-related rules, the class hierarchy of the described framework is deep and narrow. Even though there are not too many classes in the InfoMap framework, the depth of the hierarchy is as deep as the context-free grammar left-hand side terminal and non-terminal symbols. Therefore the depth of the hierarchy depends on the structure of the production rules described in chapter two.

Inheritance for generalization, or design sharing, usually indicates the need for new subclasses. The inheritance structure of the described framework needs no new subclasses, as long as no new SetRoles and SetMemberRoles are added in the context-free grammar. Therefore, the top of the class hierarchy will always be abstract, since any new production rule will have to deal with terminal symbols. Therefore, only new concrete classes will be created if the context-free grammar is expanded to include more SetRoles and SetMemberRoles.

At the specialization level the elements of subclasses can be viewed as elements of their superclass. The design of the InfoMap class hierarchy indicates that the subclasses do not refine any of the inherited operations (*i.e.*: operation "exists"), but only use them with different parameters.

For the class-related rules, large classes (*i.e.*: SetRole) were broken down into smaller classes (*i.e.*: SetRoleDominant, SetRoleDescriptive *etc.*). A class represents an abstraction. Therefore, a large class with too many operations is suspicious. The InfoMap framework classes were small in terms of operations included in each class. This is the result of the transformation of the context-free grammar described in chapter two into the class hierarchy of the framework. It is obvious that when re-write rules are designed, the non-terminals and terminals of the alphabet are broken down to the most elementary detail. Consequently, large classes are broken down into elementary subclasses. For example, a SetRole class was broken down into SetRoleDominant, SetRolDescriptive, SetRoleTransitive and SetRoleUserDefined.

Several operations that do not communicate were separated into different classes. For example, operations related to SetMemberRoleHierarchy and to SetMemberRoleGeneralization were separated into different classes. Once again, this is a direct result of the transformation of the context-free grammar into the framework, and the separation of operations that do not communicate. It is obvious that different non-terminals may be re-written in different ways.

Therefore different operations were included in different classes, separating the communication of operations.

3.5 Rational Inc. CASE Tool Specifications

The use of CASE (Computer Aided Software Engineering) tools has improved the quality of software design [15], because new methodologies for software design are accompanied by CASE tools. Therefore the process of software design has become less painful and more organized. In this section we use such a combination of a methodology and a CASE tool in order to produce specifications for our design framework. The Booch method [23] recommends the use of four models -- logical, physical, static and dynamic -- to capture the in-process products of object-oriented analysis and design. Using the Booch notation, the Rational Inc. CASE tool enables the creation and refinement of these four models within the overall model representing any problem domain. A model contains diagrams and specifications which provide the means for a formal specification of a system.

The InfoMap framework described in this section is specified using the Rational Inc. CASE tool. Rational's graphical user interface allows the creation, viewing, and modification of the components in a model. The InfoMap framework components needed for the generation of the specifications were the class diagram, the operations diagram and the state - transition diagram for each operation. Figures 3.10 and 3.11 show the class diagrams of the InfoMap methodology at the general and the detailed level of abstraction produced by the CASE tool. Figure 3.12 shows a sample of the generated specifications produced by the CASE tool.

The complete specifications for the InfoMap model produced by this CASE tool are shown in appendix A. In the second appendix we present the same specifications modeled using the InfoMap methodology.

The use of the Rational Inc. CASE tool substantially reduced the time between conceptual and actual design of the InfoMap framework. In addition to that, it facilitated a controlled interactive development. Several components were replaced during the design without any loss of the overall structure of the framework. Furthermore, the specifications produced by the CASE tool where uniformly structured. This uniform structure promoted consistency and minimal typing. On the other hand, the same framework specifications presented using the InfoMap methodology, revealed the following:

The CASE tool generated specifications fail to inform the readers whether a class is defined as abstract or concrete. This information was later added in the InfoMap model of the CASE tool generated output (appendix B. figures B.3 and B.8).

The information contained in the text, generated by the CASE tool, fails to identify the relationships that exist between the several notions in the framework design. For example, there is no formal way of describing the fact that an argument may be only of one certain type. Using the InfoMap methodology the relationship "one-to-many" between SetName {TYPE} and {ARGUMENT}, clearly identifies this description (appendix B. figures B.7 and B.12)

Finally, the overall presentation of the CASE tool generated output does not show a clear-cut view of the complexity of the specifications. On the other hand the InfoMap model of these specifications shows the magnitude of this complexity in the "cardinality" column. The "cardinality" notion of the InfoMap models is described in chapter four, figure 4.5.

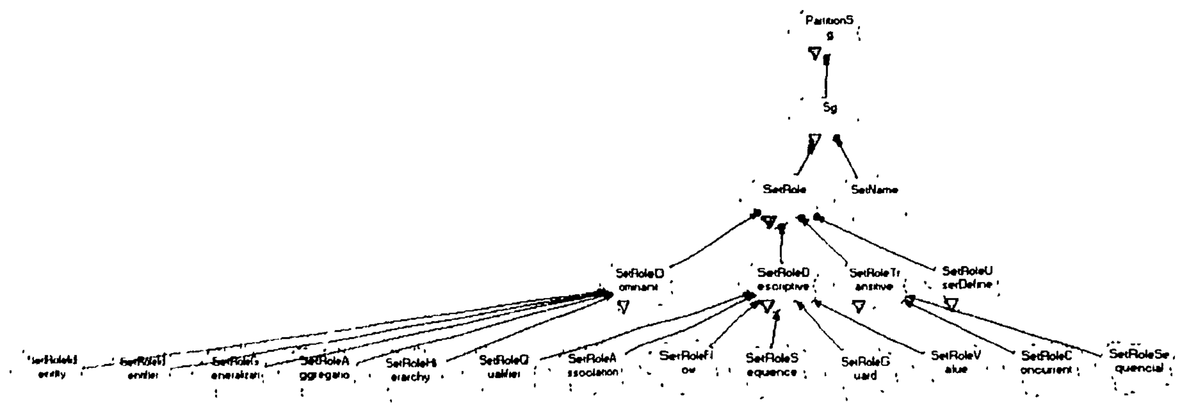


Figure 3.10. Class View Diagram for the General Level of Abstraction produced by the Rational Inc. CASE Tool

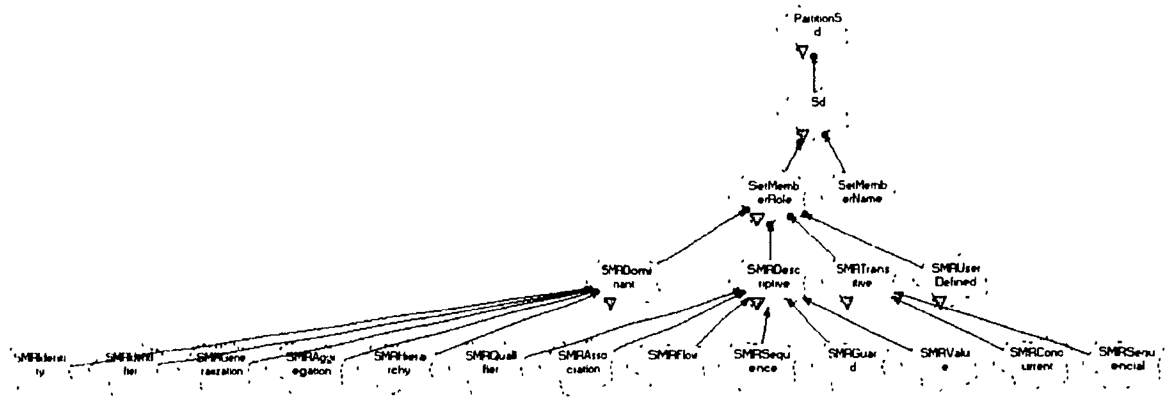


Figure 3.11. Class View Diagram for the Detailed Level of Abstraction produced by the Rational Inc. CASE Tool

Class name: Sd
Documentation:
Sd -> {[SetMemberRole SetMember]}
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: PartitionSd
Public InterfaceHas-A
Relationships:

SetMemberName
SetMemberRole
Operations: exists
State machine:Yes
Concurrency: Sequential
Persistence: Transient
Operation name:
exists
Public member of: Sd
Arguments expression ITEM

statement ITEM
LowerCaseLetter
SetMemberRole
Documentation:
operation exists is applied to SetMember / Role /expressions / statements in arrays [1] to [4]
Concurrency: Sequential

Figure 3.12. Sample of the Specifications Produced by the Rational Inc. CASE Tool

3.6 Summary, Deliverables

Framework design for the designing of concepts is not a panacea, just as object-oriented design is not either [1]. In this chapter we first introduced the concept of frameworks. Other frameworks and classifications were also discussed. The InfoMap framework was classified, and its design was shown from the point of view of a class hierarchy and the operations included in the classes of the hierarchy. The described framework was the result of a direct and straight forward translation of the context-free grammar described in the previous chapter. This translation was based on several heuristic rules, according to three observations:

- The multiplicity of associations that exist in a context-free grammar.
- The transformation of non-terminal symbols into abstract classes.
- The transformation of terminal symbols into instances of concrete classes.

The resulting framework follows certain rules for finding classes and designing frameworks. These rules (by R. Johnson) were automatically applied by the conversion process from the context-free grammar to the framework design. This enables us to conclude that any concept described in the form of a context-free grammar may be easily converted into a framework consisting of classes. A related approach was presented specifying a design pattern that, if followed, may solve the problem of defining the representation of a language and the construction of its interpreter. The overall InfoMap framework was described using the Rational Inc. CASE tool. The generated output is shown in appendix A. The CASE tool generated output was modeled using the InfoMap methodology. This model is presented in appendix B.

CHAPTER 4: DESIGN PATTERN MODELING

4.1 Introduction, Motivation

Christopher Alexander, an architect by profession, is credited for his contributions in the area of design patterns in the architectural community. He really did not have in mind software engineering when he proposed design patterns as a method for resolving design conflicts in the architectural world. Nevertheless, his ideas have been applied more in the software community than in the architectural community [31]. It is quite possible that his ideas did not gain any ground in the architectural community because of his relentless pursuit of standardizing design patterns in an art like architecture, which according to his colleagues is free of standardizations. His work in "Patterns" [32], "Notes" [33], and "Timeless" [34] converges to the same idea that the software community has been looking into from the early days of software re-usability research [35]: a way of resolving conflicts that exist as a result of existing relationships in designs. Alexander's design patterns mainly consist of three parts:

- A context that describes when a pattern is applicable; this is expressed in a set of prerequisites or preconditions that must be evaluated as true.
- The problem that the pattern resolves, explained in terms of the conflicting forces that are present in the context.
- A layout of the physical relationships that provide a solution to the problem, and the process for eliminating the conflicting forces.

The main theme of this new approach is the resolution of conflicts in designs which can provide a tranquil survival for any design. According to Alexander, this may be accomplished by the introduction of rules, processes, and design patterns [34].

Alexander says, "Each pattern describes a problem which occurs over and over again in our environment" [32]. The solution to the problem can be used a million times over, without doing the same thing twice. The same idea may be applied in software design. Design patterns describe problems which occur over and over again in software design. The solution to a problem in software design may be applied over and over again without doing the same thing twice.

This chapter is not about architecture. The main objective is to use the InfoMap methodology and framework, described in the previous chapters, in order to model design patterns that exist in object-oriented architectures. Before presenting design patterns using the InfoMap approach, we discuss issues that relate building and, software design, and definitions related to design patterns. Observations about design patterns and the InfoMap methodology used to model them are also presented.

4.2 Design Patterns: Definitions, Classification, Origin

Building design and software design do not really share any principles. However, ideas found in building design may also be found in software design [36]. These are:

- Software entities, just like buildings, engage in greater dynamic interaction. In software design, messages are sent from one component to another, and they must be understood. In building design, blocks of concrete depend on one another in terms of physical and esthetic interaction.

- Sometimes describing software is the same as constructing it. In software design several specification languages assist programmers in prototyping and producing specifications in an executable form. In building design it is often the case that the end product is described and simulated in every detail.

- Much of a software design is hidden from its user. Transparency in software design does not provide any clues to the user about mathematical and other formulas used. In building design the blueprints are not available to the users.

- Software in general has scant physical constraints. In software design laws of nature do not play any role; therefore we may experiment more freely. In building design this is definitely not the case.

- Some software requirements are allegedly less explicit than building requirements. In software design we may have ill/well-defined problems. In building design we have only well-defined problems [7, 18].

Perhaps it is true that in every other engineering discipline the end product consists of some tangible material, whereas in software there is no material substance of the end product. It is our view that the problem lies in the representation of the design of a product with no actual material substance. Often it has been said, "if only software engineering could be more like X...", where X is any design-intensive profession with a longer history than software [36]. Another favorite aphorism is, "if software engineers were designing buildings, we'd definitely be homeless right now". Design patterns have been proposed in order to deal with these handicaps.

In order to provide harmony in building design, several issues regarding design and engineering have to be resolved [36]:

- Incapacity to balance individual, group, societal, and ecological needs.

- Lack of purpose, order and human scale.

- Aesthetic and functional failure to adapt to local and physical social environments.

- Development of material and standardized components that are ill-suited for use in any specific application.

- Creation of artifacts that people do not like.

Design patterns may be a way of providing harmony in software design as well. Our contribution to this proposal is the representation of design patterns using the InfoMap/InfoSchema methodology. By doing this we provide an *n*-dimensional environment for the modeling of design patterns, and ways of uncovering problems with these design patterns.

This relatively new way of designing software is based on the observation that in order to give a solution, the designer must zero in on the most promising question. In other words, he or she must focus on the question that will give the most promising answer. Therefore, that question may provide some hints about the way of packaging and developing reusable software components at the design level. In that respect, design patterns may be viewed as a set of forces and relationships among them. These forces are described as a set of conflicting constraints acting on any solution to a given problem. The objective is a solution that will analyze the set of constraints and resolve the design conflicts.

In the area of object-oriented design and/or programming, design patterns may be applied to solve several problems. These problems and how design patterns assist in their solution are [1] :

- In order to find appropriate objects, design patterns assist the identification of abstractions and the objects that can capture these abstractions.

- Design patterns assist us in deciding what is an object, by representing the whole system as objects and/or decomposing systems into smaller objects.

- Object interface specification is also an area where design patterns may be of any assistance, by identifying elements and kinds of data that can be sent across an interface.

Ralph Johnson, Eric Gamma and others have recently proposed that Alexander's work may in fact provide a new way of thinking about software design issues. They focus mainly on the Object-Oriented technology. They are the main contributors designing frameworks and methodologies using Smalltalk and Eiffel as their implementation languages. However, we should not confuse design patterns with either methodologies or frameworks. Dictionary definitions for all three of them follow [28]:

Methodology: The science of method, a treatise or dissertation on method, systematic classification, the study of direction and implications of empirical research or of the suitability of the techniques employed in it.

Framework: A structure composed of parts framed together, one designed for enclosing or supporting anything, a frame or skeleton.

Pattern: A regular or logical form order, or arrangement of parts (behavior pattern), model design from which copies can be made, but also a random combination of shapes or colors.

Dictionary definitions may very well prove our point. Nevertheless, in adapting to the software design reality some additional points should be made if common sense is to be used. First, design patterns dictate which conclusions to make, based on what decisions, and they furnish a justification for the validity of the decisions. Second, frameworks are a way of building what you want, and specifying its layout. Finally, methodologies tell us how to write down our conclusions. In addition to that, while making the distinction between frameworks and design patterns we should point out that the latter allow only the reuse of abstract micro-architectures, without implementation. Therefore they occupy the design level of the software engineering spectrum. A language for design patterns may function much the same way. Of course, design patterns may help in designing frameworks, since a framework consists of reusable design patterns [20].

A way of organizing design patterns is essential. It has been suggested that design patterns should be classified according to their purpose and scope [1, 17]. The purpose of a design pattern is what it does. The purpose of a design pattern may be to create, structure, or describe behavior. The scope of a design pattern applies to either objects or classes. Creational design patterns concern the process of object and/or class creation. Structural design patterns deal with the composition of classes and/or objects. Behavior design patterns characterize the ways in which classes or objects interact and distribute responsibility.

Figure 4.1 [1, 17] shows the classification scheme and several design patterns that fall under each category.

			Purpose	
		Creational	**Structural**	**Behavioral**
Scope	class	Factory method	Adapter	Interpreter
				Template method
	object	Abstract Factory	Adapter	Chain of responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Figure 4.1. Design Pattern Classification Scheme

4.3 The Parts of a Design Pattern

Design patterns may be viewed as information templates, combinations of textual and diagrammatic descriptions. Erich Gamma *et.al.* [1, 17] have suggested the following attributes for each template describing a design pattern. Each part is presented with an example:

Name, classification: A simple name that can capture the overall concept that the design pattern is describing. It is an acceptable practice to include also alternative names, so that the reader may have more room to identify with the situation. The purpose of the classification is to categorize design patterns so that it will be easier to refer to families of related design patterns, learn them, and find new ones.

example: *INTERPRETER, Class Behavioral*

Intent: Several questions should be answered. These include questions as to what the design pattern does, what design issue it addresses, and the rational behind it.

example: *Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.*

Also Known As: Other well known names for the pattern.

example: *No other name for the pattern exists.*

Motivation: Since design patterns express a set of forces, their relationships and a solution to the problem, a brief statement of the purpose of a given design pattern is necessary. In addition to that, an example is needed to make things more clear to the reader.

example: *The interpreter pattern describes how to define a grammar for simple languages, represent sentences in a language, and interpret these sentences. Suppose the following grammar defines the regular expressions:*

expression ::= literal | alteration | sequence ...

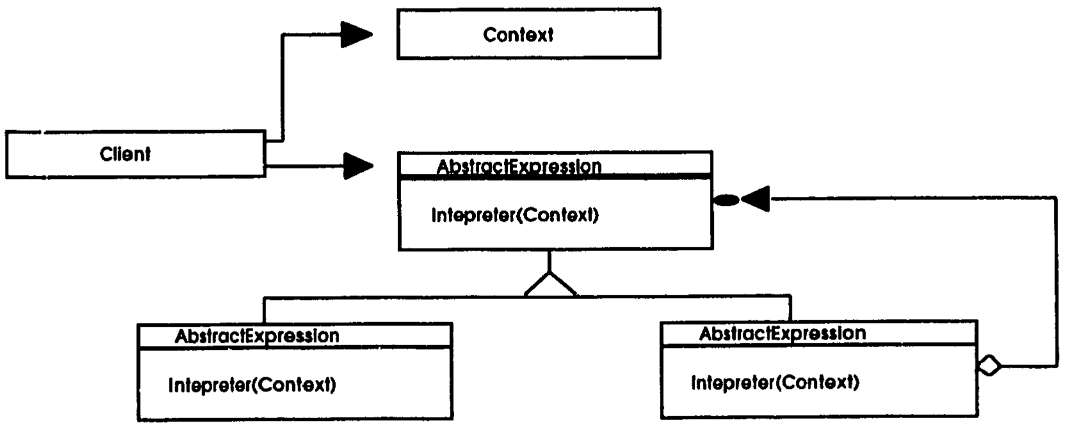
The Interpreter pattern uses a class to represent each grammar rule.

Applicability: In a poor design a design pattern may solve the problem. Therefore identifying the specific situation in which a design pattern is applied is essential to its usage.

example: *Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees.*

Structure: Usually the Object Model Technique diagrammatic representation is used to represent the design pattern's classes and instances.

example:



Participants: The classes and objects participating in the design pattern and their responsibilities, are needed in order to place each design pattern in a specific context.

example: *abstract expression, terminal expression, non-terminal expression, context, client.*

Collaborators: How the participants collaborate to carry out their responsibilities.

example: *The client builds the abstract syntax tree, each non-terminal expression node defines the interpreter, and the interpreter operation of each node uses the context to store and access the state of the interpreter.*

Consequences: When each design pattern is applied, several trade-offs exist. These trade-offs are weighed against each design pattern's objectives.

example: Easy to change and extend the grammar, grammar implementation is easy, complex grammars are hard, adding new ways to interpret expressions is easy.

Implementation: Any language-specific issue and any programming technique that is of any concern when applying a design pattern.

example: *Create abstract syntax tree, define interpreter operation, sharing terminal symbols with the Flyweight pattern.*

Sample Code: Smalltalk and/or C++ code fragments that illustrate the implementation of the design pattern.

example: (smalltalk)

```
match: InputState
        | final state |
        finalState :=  alternative1 match:
        InputState.
        finalState addAll:    (alternative2 match:
                InputState).
        ^finalState
```

Known Users: Design patterns can be found in several existing systems. Any design pattern that qualifies for inclusion in a library of design patterns should exist in at least two different domains.

example: *SpecTalk, QQCA.*

Related Patterns: Other design patterns may be related to the one currently described. Differences and similarities should be stated in this section.

example: *Composite, Flyweight, Iterator, Visitor.*

The whole point of having this template for design patterns is to provide the reader with a better view of each design pattern. In addition to that, new design patterns may be identified and presented in this template.

4.4 Modeling of Design Patterns

Design patterns provide a common design vocabulary. Therefore a common way of representing and communicating design patterns is necessary for specification purposes. In this section we present the InfoMap modeling of design patterns as a way of communicating and representing them. We accomplish that by following the example presented in the previous section: the Interpreter design pattern. Figure 4.2 shows the InfoSchema (general level) modeling of the Interpreter design pattern. Each column represents a unique partition. These partitions are listed in the SetName {PARTITION}. For each abstraction at the general level displayed in figure 4.2, an abstraction at the detailed level is presented in figures 4.3 to 4.5.

In row 5 of figure 4.2, the name and the classification of the design pattern appears. The Interpreter design pattern is classified as a behavioral design pattern.

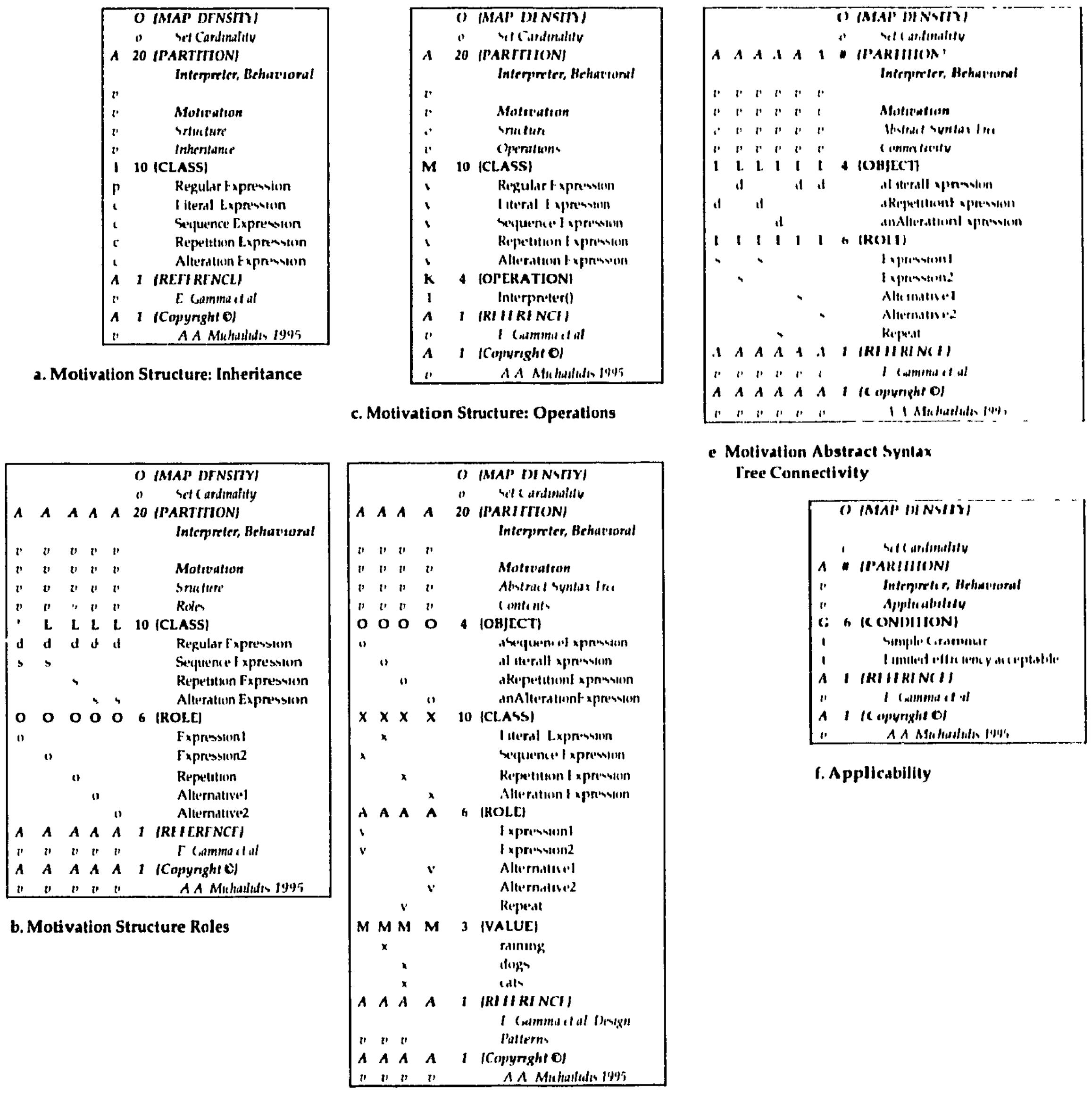
The motivation behind the Interpreter design pattern appears in columns 2 to 13 of figure 4.2. It is sub-partitioned according to the problem that the design pattern attempts to solve. The first sub-partition (columns 2 to 8) describes the structure of the problem, and the second (columns 9 to 13) describes the Abstract Syntax Tree that is part of the Interpreter design pattern. Each of the columns is described as follows:

Column 2: Inheritance relationship of the classes. The SetName participating in this partition is {CLASS} and the SetRole attached to it is "I" for "Inheritance". The detailed level of abstraction is shown in figure 4.3.a.

Column 3: Roles attached to classes. The SetNames participating in this partition are {CLASS} and {ROLE}. The SetRole attached to {CLASS} is "L" for "Sequential state-transition", and the SetRole attached to {ROLE} is "O" for "Identity". Therefore this partition describes how each role is attached to each class. The detailed level of abstraction is shown in figure 4.3.b.

	2	3	8	9	13	19	20	21	24	25	27	28	29	30	31	35	36	37
2																*O*	***{MAP_DENSITY}***	
3																*o*		*Set Cardinality*
4	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***A***	***20***	***{PARTITION}***	
5	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			***Interpreter, Behavioral***
6	*v*	*v*	*v*	*v*	*v*													***Motivation***
7	*v*	*v*	*v*															*Sructure*
8	*v*																	*Inheritance*
9		*v*																*Roles*
10			*v*															*Operations*
11				*v*	*v*													*Abstract Syntax Tree*
12				*v*														*Contents*
13					*v*													*Connectivity*
14						*v*												***Applicability***
15							*v*	*v*										***Structure***
16							*v*											*Inheritance*
17								*v*										*Sequence*
18									*v*									***Participants***
19										*v*								***Collaborations***
20											*v*							***Concequences***
21												*v*						***Implementation***
22													*v*					***Code (C++ / Smalltalk)***
23														*v*				***Known Users***
24															*v*			***Related Patterns***
25						**G**					**G**		**G**			**6**	**{CONDITION}**	
32				**O**	**L**								**F**		**X**	**4**	**{OBJECT}**	
38	**I**	**L**	**M**	**X**			**I**	**L**	**A**	**A**			**F**			**10**	**{CLASS}**	
49										**S**		**S**	**S**			**7**	**{ACTION}**	
57			**K**				**X**								**S**	**4**	**{OPERATION}**	
62		**O**		**A**	**L**											**6**	**{ROLE}**	
69				**M**												**3**	**{VALUE}**	
73													**L**			**?**	**{STATE}**	
75													**K**			**?**	**{TRANSITION}**	
77														**A**		**2**	**{PACKAGE / SYSTEM}**	
80															**O**	**4**	**{DESIGN PATTERN}**	
85	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*1*	*{REFERENCE}*	
86	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*E. Gamma et al. 1994*
87	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*1*	*{Copyright ©}*	
88	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*A.A. Michailidis 1995*

Figure 4.2. General Abstraction Model of the Interpreter Design Pattern



```
O {MAP DENSITY}
o     Set Cardinality
A 20 {PARTITION}
      Interpreter, Behavioral
v
v     Motivation
v     Structure
v     Inheritance
L 10 {CLASS}
p     Regular Expression
c     Literal Expression
c     Sequence Expression
c     Repetition Expression
c     Alteration Expression
A 1  {REFERENCE}
v     E Gamma et al
A 1  {Copyright ©}
v     A A Michailidis 1995
```

a. Motivation Structure: Inheritance

```
            O {MAP DENSITY}
            o     Set Cardinality
A A A A A 20 {PARTITION}
                  Interpreter, Behavioral
v v v v v
v v v v v         Motivation
v v v v v         Structure
v v v v v         Roles
' L L L L 10 {CLASS}
d d d d d         Regular Expression
s s               Sequence Expression
    s             Repetition Expression
      s s         Alteration Expression
O O O O O  6 {ROLE}
o                 Expression1
  o               Expression2
    o             Repetition
      o           Alternative1
        o         Alternative2
A A A A A  1 {REFERENCE}
v v v v v         E Gamma et al
A A A A A  1 {Copyright ©}
v v v v v         A A Michailidis 1995
```

b. Motivation Structure Roles

```
   O {MAP DENSITY}
   o     Set Cardinality
A 20 {PARTITION}
         Interpreter, Behavioral
v
v        Motivation
v        Structure
v        Operations
M 10 {CLASS}
v        Regular Expression
v        Literal Expression
v        Sequence Expression
v        Repetition Expression
v        Alteration Expression
K  4 {OPERATION}
1        Interpreter()
A  1 {REFERENCE}
v        E Gamma et al
A  1 {Copyright ©}
v        A A Michailidis 1995
```

c. Motivation Structure: Operations

```
           O {MAP DENSITY}
           o     Set Cardinality
A A A A  20 {PARTITION}
                 Interpreter, Behavioral
v v v v
v v v v          Motivation
v v v v          Abstract Syntax Tree
v v v v          Contents
O O O O   4 {OBJECT}
o                aSequenceExpression
  o              aLiteralExpression
    o            aRepetitionExpression
      o          anAlterationExpression
X X X X  10 {CLASS}
  x              Literal Expression
x                Sequence Expression
    x            Repetition Expression
      x          Alteration Expression
A A A A   6 {ROLE}
v                Expression1
v                Expression2
      v          Alternative1
      v          Alternative2
    v            Repeat
M M M M   3 {VALUE}
  x              raining
    x            dogs
    x            cats
A A A A   1 {REFERENCE}
                 E Gamma et al Design
v v v            Patterns
A A A A   1 {Copyright ©}
v v v v          A A Michailidis 1995
```

d. Motivation Abstract Syntax Tree Contents

```
              O {MAP DENSITY}
              o     Set Cardinality
A A A A A A  # {PARTITION}
                    Interpreter, Behavioral
v v v v v v
v v v v v c         Motivation
c v v v v v         Abstract Syntax Tree
v v v v v c         Connectivity
L L L L L L  4 {OBJECT}
  d     d d         aLiteralExpression
d   d               aRepetitionExpression
      d             anAlterationExpression
L L L L L L  6 {ROLE}
s   s               Expression1
  s                 Expression2
        s           Alternative1
          s         Alternative2
      s             Repeat
A A A A A A  1 {REFERENCE}
v v v v v c         E Gamma et al
A A A A A A  1 {Copyright ©}
v v v v v v         A A Michailidis 1995
```

e. Motivation Abstract Syntax Tree Connectivity

```
   O {MAP DENSITY}
   c     Set Cardinality
A  # {PARTITION}
v        Interpreter, Behavioral
v        Applicability
G  6 {CONDITION}
c        Simple Grammar
c        Limited efficiency acceptable
A  1 {REFERENCE}
v        E Gamma et al
A  1 {Copyright ©}
v        A A Michailidis 1995
```

f. Applicability

Figure 4.3. Detailed Abstraction Model corresponding to figure 4.2. Part [1]

	O	***{MAP_DENSITY}***	
	o		*Set Cardinality*
A	*20*	***{PARTITION}***	
v			***Interpreter, Behavioral***
v			***Structure***
v			*Inheritance*
I	**10**	**{CLASS}**	
p			AbstractExpression
c			TerminalExpression
c			NonTerminalExpression
X	**4**	**{OPERATION}**	
x			Interpreter()
A	*1*	***{REFERENCE}***	
v			*E Gamma et al Design Patterns*
A	*1*	***{Copyright ©}***	
v			*A A Michailidis 1995*

a. Structure, Inheritance

			O	***{MAP_DENSITY}***	
			o		*Set Cardinality*
A	*A*	*A*	*20*	***{PARTITION}***	
v	*v*	*v*			***Interpreter, Behavioral***
v	*v*	*v*			***Structure***
v	*v*	*v*			*Sequence*
L	**L**	**L**	**10**	**{CLASS}**	
		d			AbstractExpression
	d				TerminalExpression
		s			NonTerminalExpression
d					Context
s	s				Client
A	*A*	*A*	*1*	***{REFERENCE}***	
v	*v*	*v*			*E Gamma et al*
A	*A*	*A*	*1*	***{Copyright ©}***	
v	*v*	*v*			*A A Michailidis 1995*

b. Structure, Sequence

	O	***{MAP_DENSITY}***	
	o		*Set Cardinality*
A	*20*	***{PARTITION}***	
v			***Interpreter, Behavioral***
v			***Participants***
A	**10**	**{CLASS}**	
v			AbstractExpression
v			TerminalExpression
v			NonTerminalExpression
v			Context
v			Client
A	*1*	***{REFERENCE}***	
v			*E Gamma et al Design Patterns*
A	*1*	***{Copyright ©}***	
v			*A A Michailidis 1995*

c. Participants

		O	***{MAP_DENSITY}***	
		o		*Set Cardinality*
A	*A*	*20*	***{PARTITION}***	
v	*v*			***Interpreter, Behavioral***
v	*v*			***Collaborations***
A	**A**	**10**	**{CLASS}**	
	v			NonTerminalExpression
v				Client
S	**S**	**7**	**{ACTION}**	
1				Built A.S.T.
2				Initialize Interpreter
	1			Define Interpreter
	2			Define base case for recursion
A	*A*	*1*	***{REFERENCE}***	
v	*v*			*E Gamma et al*
A	*A*	*1*	***{Copyright ©}***	
v	*v*			*A.A Michailidis 1995*

d. Collaborators

	O	***{MAP_DENSITY}***	
	o		*Set Cardinality*
A	*20*	***{PARTITION}***	
v			***Interpreter, Behavioral***
v			***Concequences***
G	**6**	**{CONDITION}**	
t			Change Extend Grammar
t			Implementing Grammar
t			New ways to interpret expressions
f			Hard grammar maintain
A	*1*	***{REFERENCE}***	
v			*E Gamma et al*
A	*1*	***{Copyright ©}***	
v			*A A Michailidis 1995*

e. Concequences

	O	***{MAP_DENSITY}***	
	o		*Set Cardinality*
A	*20*	***{PARTITION}***	
v			***Interpreter, Behavioral***
v			***Implementation***
S	**7**	**{ACTION}**	
1			Create Abstract Syntax Tree
2			Define Interpreter Operation
3			Share terminals
A	*1*	***{REFERENCE}***	
v			*E Gamma et al*
A	*1*	***{Copyright ©}***	
v			*A A. Michailidis 1995*

f. Implementation

Figure 4.4. Detailed Abstraction Model corresponding to figure 4.2. Part [2]

	O	*{MAP_DENSITY}*
	o	*Set Cardinality*
A	*20*	*{PARTITION}*
v		***Interpreter, Behavioral***
v		***Code (C++ / Smalltalk)***
G	6	{CONDITION}
F	4	{OBJECT}
F	10	{CLASS}
S	7	{ACTION}
L	?	{STATE}
K	?	{TRANSITION}
A	*1*	*{REFERENCE}*
v		*E. Gamma et al*
A	*1*	*{Copyright ©}*
v		*A.A. Michailidis 1995*

a. Code (C++/Smalltalk)

	O	*{MAP_DENSITY}*
	o	*Set Cardinality*
A	*#*	*{PARTITION}*
v		***Interpreter, Behavioral***
v		***Known Users***
A	2	{PACKAGE / SYSTEM}
v		SPECTalk
v		QOCA
A	*1*	*{REFERENCE}*
v		*E. Gamma et al*
A	*1*	*{Copyright ©}*
v		*A.A. Michailidis 1995*

b. Known Users

				O	*{MAP DENSITY}*
				o	*Set Cardinality*
A	*A*	*A*	*A*	*20*	*{PARTITION}*
v	*v*	*v*	*v*		***Interpreter, Behavioral***
v	*v*	*v*	*v*		***Related Patterns***
X	**X**	**X**	**X**	**4**	**{OBJECT}**
x	x	x			Abstract Syntax Tree
S	**S**	**S**	**S**	**4**	**{OPERATION}**
	1				Share
		1			Traverse
			1		Maintain Behavior
O	**O**	**O**	**O**	**4**	**{DESIGN PATTERN}**
o					Composite
	o				Flyweight
		o			Iterator
			o		Visitor
A	*A*	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*	*v*		*E. Gamma et al*
A	*A*	*A*	*A*	*1*	*{Copyright ©}*
v	*v*	*v*	*v*		*A.A. Michailidis 1995*

c. Related Patterns

O	***{MAP_DENSITY}***
o	*Set Cardinality*
20	***{PARTITION}***
6	**{CONDITION}**
4	**{OBJECT}**
10	**{CLASS}**
7	**{ACTION}**
4	**{OPERATION}**
6	**{ROLE}**
3	**{VALUE}**
?	**{STATE}**
?	**{TRANSITION}**
2	**{PACKAGE / SYSTEM}**
4	**{DESIGN PATTERN}**
1	*{REFERENCE}*
	E. Gamma et al
1	*{Copyright ©}*
	A.A. Michailidis 1995

d. Cardinality

Figure 4.5. Detailed Abstraction Model corresponding to figure 4.2. Part [3]

Column 8: Operations contained in each class. The SetNames participating in this partition are {CLASS} and {OPERATION}. The SetName {CLASS} is assigned the SetRole "M" for "Association" and the SetName {OPERATION} is assigned the SetRole "K" for "Identifier". Therefore, each class contains a set of operations. The detailed level of abstraction is shown in figure 4.3.c.

Column 9: Contents of the Abstract Syntax Tree. The SetNames participating in this partition are {OBJECT}, {CLASS}, {ROLE} and {VALUE}. The SetName {OBJECT} is assigned the SetRole "O" for "Identity", the SetName {CLASS} is assigned the SetRole "X" for "Qualifier", the SetName {ROLE} is assigned the SetRole "A" for "Partition-association" (letter "A" is a SetRoleUserDefined), and the SetName {VALUE} is assigned the SetRole "M" for "Association". Therefore the abstract syntax tree is shown as an object in a given class. The classes are associated with SetRoles between them, and contain instances which are the nodes of the abstract syntax tree. The detailed level of abstraction is shown in figure 4.3.d.

Column 13: Connectivity of the Abstract Syntax Tree. The SetNames participating in this partition are {OBJECT} and {ROLE}. They both have SetRoles "L" for "Sequential state-transition". Therefore the structure of the abstract syntax tree is described in terms of objects and roles between these objects. The detailed level of abstraction is shown in figure 4.3.e.

The applicability of the design pattern is shown in column 19 of figure 4.2. Whether the design pattern is applicable or not is described in the SetName {CONDITION}. This SetName contains the several conditions that should be meet in order for the design pattern to be applicable. The SetRole "G" for "Guard" is attached to the SetName {CONDITION} in order to demonstrate this point. The detailed level of abstraction is shown in figure 4.3.f.

The structure of the design pattern is shown in columns 20 and 21 of figure 4.2. This structure is sub-partitioned into an inheritance and a sequence structure as follows:

Column 20 shows the inheritance of the design pattern: The SetNames participating in this partition are {CLASS} and {OPERATION}. The SetName {CLASS} is assigned the SetRole "I" for "Inheritance", and the SetName {OPERATION} is assigned the SetRole "X" for "Qualifier". This indicates that operations are contained in classes. The detailed level of abstraction is shown in figure 4.4.a.

Column 21 shows the sequence of the classes in the structure of the design pattern. The SetName {CLASS} is assigned the SetRole "L" for "Sequential-state-transition" to demonstrate this sequence; the detailed level of abstraction is shown in figure 4.4.b.

The participant classes of the design pattern are shown in column 24 of figure 4.2. The SetName {CLASS} is assigned the SetRole "A" for "Partition-Association". Therefore this sub-partition states the classes that participate in the design pattern. The detailed level of abstraction is shown in figure 4.4.c.

The collaborator's structure of the design pattern is shown in column 25 of figure 4.2. This is presented in terms of the classes that participate in the design pattern and their actions. The SetNames participating in this partition are {CLASS} and {ACTION}. The SetName {CLASS} is assigned the SetRole "A" for "Partition-Association" and the SetName {ACTION} is assigned the SetRole "S" for "Sequence". Therefore the collaboration between classes in the design pattern is shown in terms of actions that each class is performing. The detailed level of abstraction is shown in figure 4.4.d.

The consequences of the design pattern in terms of its conditions are shown in column 27 of figure 4.2. The SetName {CONDITION} is participating in the design pattern, and it is assumed the SetRole "G" for "Guard". Therefore the application of the design pattern results in a series of conditions that are true or false after the design pattern has been applied. The detailed level of abstraction is shown in figure 4.4.e.

Implementation-related issues of the design pattern are shown in column 28 of figure 4.2, in terms of sequences of actions, with the SetRole "S" for "Sequence" attached to the SetName {ACTION}. Therefore each implementation-related issue is listed in a sequential fashion in the SetName {ACTION}. The detailed level of abstraction is shown in figure 4.4.f.

Code modeling of the implementation of the design pattern is shown in column 29 of figure 4.2. This column is further explained in chapter 6, where we present a system that traces control flow graphs. This model is described by the SetNames {CONDITION} with SetRole "G" for "Guard", {OBJECT} and {CLASS} with SetRole "F" for "Flow", {ACTION} with SetRole "S" for "Sequence", {STATE} with SetRole "L" for "Sequential-state-transition", and {TRANSITION} with SetRole "K" for "Identifier". The detailed level of abstraction is shown in figure 4.5.a.

The known users of this design pattern are shown in column 30 of figure 4.2. These known users are listed in the SetName {PACKAGE/SYSTEM} with the SetRole "A" for "Partition-Association". The detailed level of abstraction is shown in figure 4.5.b.

Column 31 of figure 4.2 shows the design patterns that are related to the one currently described. The SetName {OBJECT} is assigned the SetRole "X" for "Qualifier", and the SetName {OPERATION} is assigned the SetRole "S" for "Sequence". Therefore these related design patterns are described in terms of associated objects and sequences of operations that are used from other design patterns. The detailed level of abstraction is shown in figure 4.5.c.

At this point we introduce another attribute of the InfoMap approach: the cardinality of each SetName. The number of SetMembers of each SetName appears in column 35 of figure 4.2, on the left side of each SetName. The detailed level of abstraction is shown in figure 4.5.d.

4.5 Observations

The point of the exercise presented in section 4.4 of this chapter is to demonstrate several aspects of the InfoMap representation methodology. Abstractions at the general level (InfoSchemata) and design patterns share common properties:

encapsulation, generality, equilibrium, abstraction, openness, and composibility. These properties are demonstrated using examples. In addition to these common properties, we show how we have used the InfoMap methodology to arrive at design patterns through the modeling of several existing methodologies.

4.5.1 InfoMap - Design Patterns Common Properties

At the general level of abstraction (InfoSchemata) the InfoMap representation methodology presents the characteristic of composibility. Several schemata can be interrelated in a "whole-part" relationship. Consider the example given in figure 4.6.a. It relates two partitions. This relationship representation can be composed into one partition, as shown in figure 4.6.b. In figure 4.6.a, SetName {Set1} and SetName {Set2} are related in a one - to - one order. SetName {Set2} and SetName {Set3} are also related in a one - to - one order. Therefore they can be composed in such a way so that SetName {Set1} can be related to SetName {Set2} and SetName {Set3} in a one - to - one - to - one relationship, as shown in figure 4.6.b. Therefore composibility of partitions at the general level of abstraction is possible, and enables the synthesis of partitions that share common SetNames.

A	**A**	**{PARTITION}**
v		Partition 1
	v	Partition 2
	O	**{SET1}**
O	**X**	**{SET2}**
X		**{SET3}**

a. Two Partitions

A	**{PARTITION}**
v	Pn = P1 compose P2
O	**{SET1}**
X	**{SET2}**
X	**{SET3}**

b. Composed Partitions

Figure 4.6. InfoMap Composibility

A design pattern can be specialized. It can be extended to small details, and applied to solve different problems. For example, the Interpreter design pattern presented in section 4.4 can be applied to different languages, and therefore to different grammars. This characteristic is called openness. The same way that design patterns can be instantiated based on the details of a specific problem, abstractions at the general level (InfoSchemata) of the InfoMap can be instantiated. This is shown in figure 4.7. Part [a] shows a partition at the general level of abstraction. Part [b] shows an instantiation of that partition, and part [c] shows a different instantiation of the same partition. These two instantiations depend on problem-specific details. The first instantiation at the general level of abstraction (figure 4.7.b) describes a one - to - one relationship between identification numbers and names, while the second instantiation (figure 4.7.c) describes a one - to - one relationship between professors and course sections. Therefore, several problem constraints such as SetName's instantiation provide the basis for a design pattern's openness.

A	{PARTITION}
v	Generic
O	{SET1}
X	{SET2}

a. Generic Partition

AAA	{PARTITION}
v v v	Instantiation 1
OOO	{ID#}
o	123
o	234
o	345
XXX	{NAME}
x	John
x	Kelly
x	Jerry

b. Instantiation 1

AAA	{PARTITION}
v v v	Instantiation 2
OOO	{PROFESSOR}
o	John
o	Kelly
o	Jerry
XXX	{SECTION}
x	A
x	B
x	C

c. Instantiation 2

Figure 4.7. InfoMap Openness

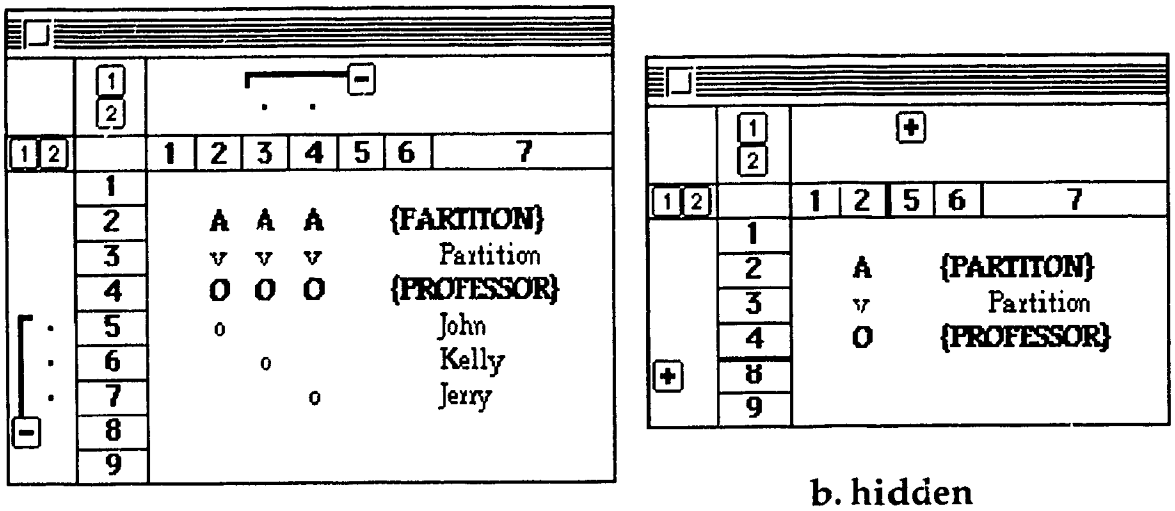
Design patterns and the InfoMap representation methodology share another common characteristic: the capability of a design pattern, as well as abstractions at the general level (InfoSchemata), to be abstract. It can be said that this abstraction is the opposite of the openness characteristic presented in the previous paragraph. Figures 4.7.b and 4.7.c share the same SetRoles. Therefore they can be abstracted at the general level and presented in figure 4.7.a. The

InfoMap representation methodology provides this generality within the stated context of a problem.

Design patterns provide a step by step approach in solving a problem. This approach provides an equilibrium for each step taken in solving a particular problem. The same applies in the InfoMap representation methodology. The equilibrium that this approach provides is the solution space where the problem is stated. The template of information contained in the general level of abstraction (*i.e.* : SetRoles and SetNames) provides the establishment of objective equilibria [34]. In figure 4.7.a there exists a very well defined and established equilibrium between two sets related in a one - to - one relationship. Therefore each partition at the general level of abstraction (InfoSchema) establishes an equilibrium through the SetRoles attached to each SetName.

The design pattern approach solves a series of problems. This characteristic of generativity is also present in the InfoMap approach. Several problems have been represented using this approach by experts or novice users [40, 41]. Applications of specific abstractions at the general level (InfoSchemata) have been used to model several problems. One of them is presented in figure 4.7.a. It is a template of information that is used just like a recipe. It is used to design processes performing different things, therefore using different ingredients. In the next chapter we describe this template in further detail.

Design patterns provide the characteristic of encapsulation. The representation of objects and their implementation can be designed in such a way so that they can be hidden from the outside world. Tools have been developed to perform encapsulation. Figure 4.8 shows this encapsulation characteristic within the InfoFarm [42] environment. In part [b] details of the SetName {SET1} are hidden, and in part [a] details of the same SetName are visible. Within the InfoFarm environment we are able to switch from a hidden to a visible state of information, and therefore provide the concept of encapsulation.



a. visible

b. hidden

Figure 4.8. InfoMap Encapsulation

4.5.2 Arriving at Design Patterns

It is often the case that a simple word, such as "pattern", may be used in different knowledge domains to capture different notions. The definition of the word "pattern" was given in section 4.2 of this chapter. However, when this word is mentioned in a specific knowledge domain it describes different notions. For example, C. Alexander used this word to describe design patterns that exist in the architecture of buildings. Leonardo da Vinci used the same word to describe design patterns that exist in paintings and engineering concepts [58]. J.F. Sowa used the word "pattern" to describe structures that exist in system's architecture [3].

We can claim that our arrival at design patterns came through the InfoMap modeling of known systems and methodologies. In [50] the various documents that are used during the software engineering life cycle are described as design patterns. Furthermore in [10] a library of design patterns is given. It contains the models of design patterns used to model hierarchies, aggregations, generalizations, data flows, process models and relational database models. Therefore it is clear to us that the idea behind design patterns as it is applied to system development is not new. Whether it is object-oriented development or any other methodology, design patterns have been researched and published since the late 1960's when the idea of repository-type products was first argued [54].

What is needed is a system that can automate this repository-type product. The InfoMap methodology provides an environment for the development of such a repository system. Even though we are far from the automation of such a system, such a reality may be applied into any body of knowledge that design patterns exist, and can be modeled using the InfoMap methodology. We do not claim that there could be a complete automated repository for every design pattern that exists in this world, but we agree with the following statement given in [3]:

> *"An automated repository system for models makes it clear that architecture is no longer mere intellectual entertainment. It will become an imperative for any enterprise that needs to be a serious player in the information age."*

4.6 Summary, Deliverables

Design patterns provide us with a systematic approach to problem solving. This systematic approach is the result of the observations made by experienced problem solvers who had to deal with the same design problem in the past. Therefore the solutions applied over and over again have been standardized. Christopher Alexander's ideas of design patterns were applied in building architecture [37]. In software design the idea of design patterns has been applied in object-oriented design and programming. Design patterns were applied in order to standardize several templates for problem solving. In this chapter we have discussed and delivered the following:

- Design patterns from the point of view of building design.
- Common characteristics of design patterns in building design and software design.

- Design pattern definitions, and how they differ from frameworks and methodologies.

- Design pattern classification and parts.

- The use of the InfoMap representation methodology to represent the parts of design patterns.

- Application of design pattern characteristics in the InfoMap representation methodology, and common properties of both.

- InfoMap representation of design patterns in an automated repository-type system.

CHAPTER 5: InfoMap IN InfoMap

5.1 Introduction, Motivation

Keeping in mind that research [17] has shown that knowledge is not organized around syntax, but in larger conceptual structures such as data structures and algorithms, plans of action should exist for every case that arises and requires the inclusion of these structures at the design level. Therefore we need to establish a way of describing these conceptual structures and their inclusion at the design level. Also, at the design level, we should be able to accomplish reusability by abstracting up to a point any implementation-related issue. This endeavor can be achieved as long as the following is present in any approach:

- A common vocabulary for the efficient and effective communication of designs and design documents [10].

The above point may vary from one schematic description to another in terms of effectiveness to accomplish its objective, or in terms of the actual representation methodology [9]. In addition it may add overhead, due to sometimes complex CASE tools [38].

In this chapter we use this point to demonstrate and test the InfoMap representation methodology. The testing is performed on the vocabulary of the InfoMap representation methodology, by presenting it in the InfoMap format. The template of information used to achieve this is the framework presented in chapter three, modeled at the general (infoSchema) and detailed level (InfoMap) of abstraction. Furthermore we identify the several by-products of the InfoMap representation methodology, categorized according to their function.

5.2 InfoMap Presented in its Own Terms

Several methodologies exist for the modeling and design of concepts in software engineering. Some of these methodologies are accompanied by CASE tools [39, 15]. In addition to that, several methodologies attempt to use their own notation to describe themselves [39]. The common vocabulary of the InfoMap representation methodology can be effectively used to communicate designs and documents. This can be done by using the methodology itself. In the following section two aspects demonstrate our point:

- The basics of the methodology in its own terms (from chapter two)
- The syntax of the methodology in its own terms (from chapter three)

The basics of the approach described in this thesis were stated in chapter two. These basics formulated the tabular structure of the representation methodology. Figure 5.1 shows the SetRoles, SetMemberRoles, SetNames and SetMemberNames of the InfoMap representation methodology in terms of itself. SetName {PARTITION} in row #2 contains the SetMemberName InfoSyntax (row #3), which indicates that the concept described is the syntax of the methodology. Furthermore, the generic roles of the methodology are listed in rows #4, #5, #6. They describe the dominant, descriptive and transitive roles. SetName {SET ROLE} in row #7 contains the SetMemberNames for all the SetRoles used by the InfoMap representation methodology to describe concepts at the general level of abstraction (InfoSchema). SetName {SET MEMBER ROLE} in row #22 contains the SetMemberNames for all the SetMemberRoles used by the InfoMap representation methodology to describe concepts at the detailed level of abstraction (InfoMap). The two SetNames ({SET ROLE} and {SET MEMBER ROLE}) are related using an inheritance structure. Therefore, each SetMember of the SetName {SET ROLE} is a parent to several SetMembers of the SetName {SET MEMBER ROLE}.

In chapter three we presented a design framework for the InfoMap representation methodology, by describing the classes of SetRoles and their instances, which are the SetMemberRoles. These classes are presented in figure 5.1 and are listed in SetName {SET ROLE}. The instances of the classes are listed in the same figure in SetName {SET MEMBER ROLE}. The relationship of inheritance explained in the previous paragraph corresponds to the inheritance relationship between the specified classes of chapter three. Therefore, it can be shown that the design framework can also be represented using the InfoMap representation methodology. Furthermore, the operations performed on the

SetRoles and SetMemberRoles that were described in chapter four can also be presented in the InfoMap format. This is shown in figures 5.2 and 5.3.

Figure 5.2 shows the operations at the general level of abstraction (InfoSchema). SetNames {SET ROLE} and {OPERATION} are related in a one - to - many relationship. This can be seen by the SetRole "O - Identity" attached to SetName {SET ROLE}, and SetRole "M - Association" attached to SetName {OPERATION}. An example of this one - to - many association between SetRoles and operations can be seen in column #4 of figure 5.3. A DominantSetRole that exists in ***array [2]***, identified by the upper case letter "K", is an "Identifier".

Figure 5.3 shows the operations at the detailed level of abstraction. The SetMembers represent the operations at the detailed level of abstraction. For example, a unique DominantSetMemberRole that exists in ***array [3]***, identified by the lower case letter "o", is an "Identity".

The InfoSyntax and the operations represented in the tabular format of the InfoMap representation methodology show that it is possible to use the basics of the methodology to represent itself. Furthermore it is possible to add new SetRoles, new SetMemberRoles, and new operations in each of the three basic categories of SetRoles (Dominant, Descriptive, Transitive), just by inserting new columns and/or rows in SetNames {SET ROLE} {SET MEMBER ROLE} and {OPERATIONS}. In this section we have shown how it is possible to use the InfoMap representation methodology itself to model two aspects of itself: first the basics covered, in chapter two, and second the framework, covered in chapter three. Two conclusions are derived from this attempt to use the InfoMap representation itself to represent itself. First, the methodology can be applied to itself. Second, the framework can be expanded using the methodology itself to describe this expansion. The SetRole "U - User Defined" (figure 5.1, row 8, column 3) can be used to make this possible.

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	4	*{PARTITION}*	
3	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*InfoSyntax*
4		*v*	*v*	*v*	*v*	*v*											*Dominant Roles*
5							*v*	*v*	*v*	*v*	*v*	*v*					*Descriptive Roles*
6													*v*	*v*			*Transitive Roles*
7	I	I	I	I	I	I	I	I	I	I	I	I	I	I	14	{SET ROLE}	
8	p																U - User Defined
9		p															K - Identifier
10			p														O - Identity
11				p													H - Hierarchy
12					p												I - Generalization
13						p											P - Aggregation
14							p										X - Qualifier
15								p									M - Association
16									p								F - Flow
17										p							G - Guard or Goal
18											p						S - Sequence
19												p					V - Value or Instance
20													p				L - Sequencial State Transition
21														p			C - Concurrent State Transition
22	I	I	I	I	I	I	I	I	I	I	I	I	I	I	28	{SET MEMBER ROLE}	
23	c																letter, symbol
24		c															id - unique identifier
25			c														o - column marker
26				c													h - root
27				c							c						1 n - part marker
28					c												p - parent
29					c												c - child
30						c											w - whole
31						c											c - part
32						c											v - visible part
33						c											h - hidden part
34						c											m - many parts
35							c										x - qualifier marker
36								c									v - row marker
37								c									k - key attribute
38									c								u - used, input
39									c								o - produced, output
40										c							t - true
41										c							f -false
42										c							T - implied true
43										c							F - implied false
44												c					instance, value, string
45													c				s - source
46													c				d - destination
47													c				l - loop
48													c				a - assertion
49													c				e - exemption
50														c			c - concurrent
51	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	1	*{COPYRIGHT © }*	
52	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*WM Jaworski 1988-1994*

Figure 5.1. InfoMap Framework Presented in the InfoMap Methodology

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	4	*{PARTITION}*	
3	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*Operations General Level (InfoSchema)*
4		*v*	*v*	*v*	*v*	*v*											*Dominant Roles*
5							*v*	*v*	*v*	*v*	*v*	*v*					*Descriptive Roles*
6													*v*	*v*			*Transitive Roles*
7	O	O	O	O	O	O	O	O	O	O	O	O	O	O	14	{SET ROLE}	
8	o																U - User Defined
9		o															K - Identifier
10			o														O - Identity
11				o													H - Hierarchy
12					o												I - Generalization
13						o											P - Aggregation
14							o										X - Qualifier
15								o									M - Association
16									o								F - Flow
17										o							G - Guard or Goal
18											o						S - Sequence
19												o					V - Value or Instance
20													o				L - Sequencial State Transition
21														o			C - Concurrent State Transition
22	M	M	M	M	M	M	M	M	M	M	M	M	M	M	20	{OPERATION}	
23	v	v	v	v	v	v	v	v	v	v	v	v	v	v			Exists in Area [1], [2], [3], [4]
24	v	v	v	v	v	v	v	v	v	v	v	v	v	v			SetRole
25	v	v	v	v	v	v	v	v	v	v	v	v	v	v			UpperCase Letter
26	v	v	v	v	v	v											UpperCase Generic Dominant Letter in Area [2]
27							v	v	v	v	v	v					UpperCase Generic Descriptive Letter in Area [2]
28													v	v			UpperCase Generic Transitive Letter in Area [2]
29	v																U in Area [2]
30		v															K in Area [2]
31			v														O in Area [2]
32				v													H in Area [2]
33					v												I in Area [2]
34						v											P in Area [2]
35							v										X in Area [2]
36								v									M in Area [2]
37									v								F in Area [2]
38										v							G in Area [?]
39											v						S in Area [2]
40												v					V in Area [2]
41													v				L in Area [2]
42														v			C in Area [2]
43	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	2	*{COPYRIGHT ©}*	
44	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*A A Michailidis 1995*
45	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*WM Jaworski 1988-1994*

Figure 5.2. Operations of the Framework at the General Level of Abstraction Presented in the InfoMap Methodology

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	4	*{PARTITION}*	
3	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*Operations Detailed Level (InfoMap)*
4		*v*	*v*	*v*	*v*	*v*											*Dominant Roles*
5							*v*	*v*	*v*	*v*	*v*	*v*					*Descriptive Roles*
6													*v*	*v*			*Transitive Roles*
7	O	O	O	O	O	O	O	O	O	O	O	O	O	O	14	{SET MEMBER ROLE}	
8	o																u - User Defined
9		o															id - Identifier
10			o														o - Identity
11				o													h, 1..n - Hierarchy
12					o												p, c - Generalization
13						o											w, p, v, h, m - Aggregation
14							o										x - Qualifier
15								o									v - Association
16									o								u, o - Flow
17										o							t, f, T, F - Guard or Goal
18											o						1..n - Sequence
19												o					string, number - Value or Instance
20													o				s, d, e, a, l - Sequencial State Transition
21														o			c - Concurrent State Transition
22	M	M	M	M	M	M	M	M	M	M	M	M	M	M	28	{OPERATION}	
23	v	v	v	v	v	v	v	v	v	v	v	v	v	v			Exists in Area [1], [2], [3], [4]
24	v	v	v	v	v	v	v	v	v	v	v	v	v	v			SetMemberRole
25	v	v	v	v	v	v	v	v	v	v	v	v	v	v			LowerCase Letter
26	v	v	v	v	v	v											LowerCase Generic Dominant Letter in Area [1]
27							v	v	v	v	v	v					LowerCase Generic Descriptive Letter in Area [1]
28													v	v			LowerCase Generic Transitive Letter in Area [1]
29	v																Letter, symbol in Area [1]
30		v															id in Area [1]
31			v														Unique o in Area [1]
32				v													Unique h in Area [1]
33					v												p, c in Area [1]
34						v											w, c, v, h, m in Area [1]
35							v										x in Area [1]
36								v									v in Area [1]
37									v								u, o in Area [1]
38										v							t in Area [1].row.column xor ITEM = <true> in Area [3].row
39										v							f in Area [1].row.column xor ITEM = <false> in Area [3].row
40										v							T in Area [1].row.column xor ITEM = <complim. t> in Area [3].row
41										v							F in Area [1].row.column xor ITEM = <complim f> in Area [3].row
42				v													1..n in Area [1]
43											v	v					Number in Area [1]
44												v					String in Area [1]
45													v				Unique s in area [1].column
46													v				d in area [1].column
47													v				l in area [1] column
48													v				a in area [1].column
49													v				e in area [1].column
50														v			c in area [1].column
51	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	2	*{COPYRIGHT ©}*	
52	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*A A Michailidis 1995*
53	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*			*WM Jaworski 1988-1994*

Figure 5.3. Operations of the Framework at the Detailed Level of Abstraction Presented in the InfoMap Methodology

5.3 InfoMap by-Products

In the previous section of this chapter we described the InfoMap representation methodology in terms of itself. In this section we identify the several parts of this representation methodology and their contents. The major parts of this technology are the InfoSyntax (described in 5.2), the InfoSchemata (abstractions at the general level, described in chapter two) and the InfoFactory. These are shown in figure 5.4.

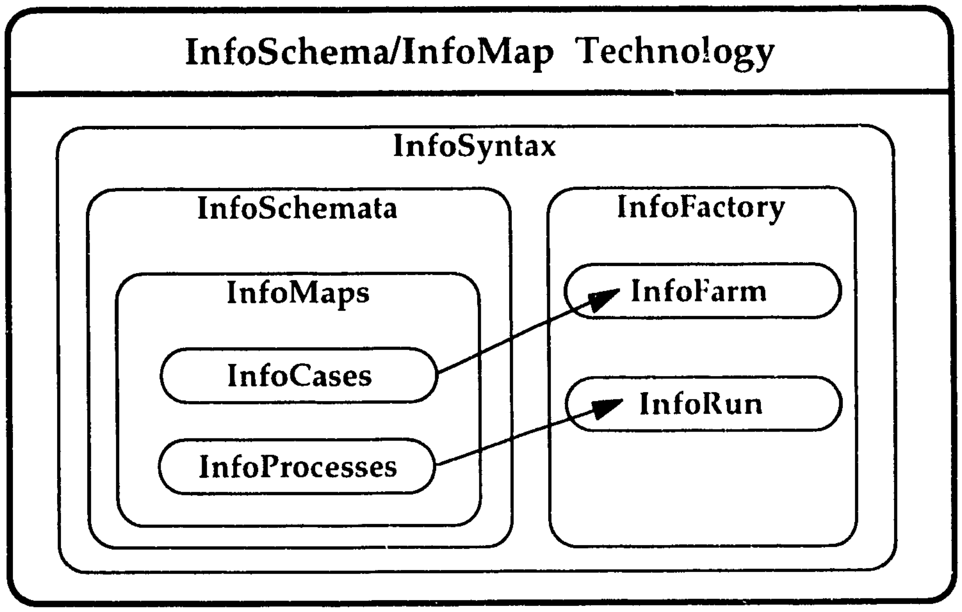


Figure 5.4. Parts of the InfoMap Technology

The InfoSyntax contains the specific SetRoles and SetMemberRoles used to model concepts in the InfoMap representation methodology. These concepts can be represented at the InfoSchema level (abstractions at the general level) and the InfoMap level (abstractions at the detailed level). Furthermore, these concepts can be classified as InfoCases and InfoProcesses. InfoCases are templates of information presented in the InfoMap representation methodology. The design pattern "Interpreter" presented in chapter four is an example of an InfoCase. It is a body of knowledge modeled using the InfoMap representation methodology. On the other hand, InfoProcesses describe processes that can be applied to solve particular problems. In chapter six we presented the InfoMap for InfoProcesses.

The InfoSyntax is also used by the InfoFactory. The InfoFactory is a collection of tools that assist developers of models in the InfoMap representation methodology. Two tools are currently available. The first, InfoFarm, is used to built InfoCases by altering the Ms-Excel environment in such a way that InfoMap-related aspects can be demonstrated [42]. These aspects range from easy insertion of rows and columns in an InfoCase to libraries of abstractions at the general (InfoSchema) and/or detailed (InfoMap) level of abstraction. The second tool (described in chapter six) is used to trace processes developed to solve a particular problem. This problem is stated in terms of states and transitions between states. Both tools are developed for teaching [40, 41] and are also commercially available.

5.4 Summary, Deliverables

This chapter started with the statement that knowledge is not organized around syntax, but in larger conceptual structures such as data structures and algorithms. The InfoMap representation methodology provides us with an environment for the representation of knowledge that is captured in data structures and algorithms. Furthermore the use of the InfoMap representation methodology can be applied to itself, demonstrated in figures 5.2, 5.3 and 5.4. This shows that the methodology can be represented in terms defined by itself. Several parts of the InfoMap representation methodology were presented. These were categorized into InfoMaps at the general, and InfoSchemata at the detailed, levels of abstraction. These are used to represent general knowledge (InfoCases, InfoProcesses), and may be manipulated by tools contained in the InfoFactory.

CHAPTER 6: TRACING OF CONTROL FLOW GRAPHS

6.1 Introduction, Motivation

Spreadsheet software packages can be used to accommodate the InfoMap methodology. These spreadsheet packages do not provide several functions essential to the specific needs of the users of this methodology. Tools have been developed to alter a spreadsheet's behavior. These tools belong to the InfoFactory environment described in chapter five. Several InfoCases (recovered bodies of knowledge), modeled using the InfoMap methodology were developed for student projects and commercial applications [40, 41, 42]. In addition, processes described in control flow graphs were modeled using the InfoMap methodology [43]. The processes and their models are presented in this chapter. A system for tracing and/or executing such processes is also presented as part of the InfoFactory environment. Through examples we show how the InfoMap model of a control flow graph can be used to trace the source code of design patterns.

6.2 Representing Control Flow Graphs

In chapter two we described the basics of the InfoMap methodology. In chapter three we described a framework approach to represent the InfoMap methodology. The tabular template that was produced, and the operations described in these chapters, can be specialized to deal with specific problems. One of these problems is the representation of control flow graphs.

A control flow graph is a collection of nodes and arcs that connect nodes. It describes the flow of execution of a given process. It contains states, transitions, conditions and actions. A control flow node is a state. A control flow arc is a transition. Onto these arcs we attach conditions and actions. Several arcs can be attached to a given node. Therefore a given state can be connected to many other states through several transitions. More formally this is stated in [44] and shown in figure 6.1, were circles represent states and arcs represent transitions.

"Each transition links states; each transition is implemented by a sequence of actions guarded by conditions."

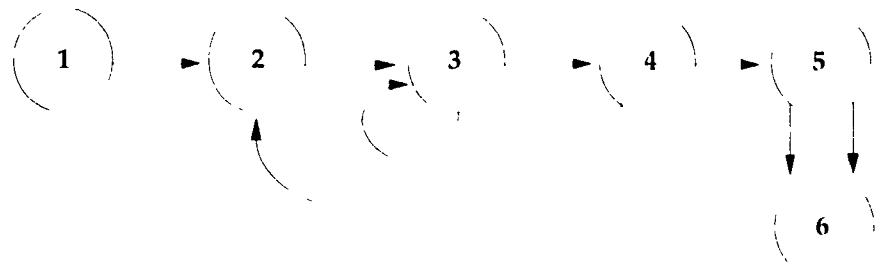


Figure 6.1. Diagrammatic Representation of a Typical Control Flow Graph

6.2.1 General Level of Abstraction

The graph in figure 6.1 can be used to represent processes. On the other hand, these processes can be represented using the InfoMap representation methodology. Therefore it is possible to represent program segments or even complete programs in this format. The standard SetRoles and SetNames (at the general level or InfoSchema) that are used to represent processes in the InfoMap representation methodology are shown in figure 6.2.

L	{STATE}
O	{TRANSITION}
G	{PreCONDITION}
S	{ACTION}
G	{PostCONDITION}

Figure 6.2. InfoMap Representation of a Control Flow Graph at the General Level of Abstraction

In figure 6.2 the SetName {STATE} is allocated the SetRole "Sequential state-transition" (letter "L"); the SetName {TRANSITION} is allocated the SetRole "Identity" (letter "O"); the SetName {PreCONDITION} is allocated the SetRole "Guard" (letter "G"); the SetName {ACTION} is allocated the SetRole "Sequence" (letter "S"); the SetName {PostCONDITION} is allocated the SetRole "Guard" (letter "G"). The rational behind the use of these SetRoles in the model is the following: The dominant SetRole "Identity" represents the connection between states. These states are given the SetRole "Sequential state - transition".

In order to move from one state to another and perform the actions of SetName {ACTION} in "Sequence", the "Guard(s)" of the SetName {PreCONDITION} should be evaluated as true. After the change of states takes place the {PostCONDITION} "Guards" will evaluate whether the change of states was a successful one.

6.2.2 Detailed Level of Abstraction

The five SetNames and their corresponding SetRoles identified in section 6.2.1 (figure 6.2) can be expanded to the detailed level of abstraction (InfoMap). This can be accomplished by listing the SetMembers of each SetName, and relating them within and between the SetNames with the use of the SetMemberRoles. The SetMember's listing and their corresponding SetMemberRoles are described in this section.

A state can be identified as either a source or a destination state. A state can be both a source and a destination state. States are connected by transitions. These transitions can represent one of the cases described in figure 6.3. These cases are:

- Branch: a simple source to destination transition.
- Branch with fork: a transition that travels from source to either an exemption or assertion state.
- Loop: a transition that has the same source, and destination state.
- Loop with exit: a transition that has the same source, and destination state and can reach any other specified state through an exemption.

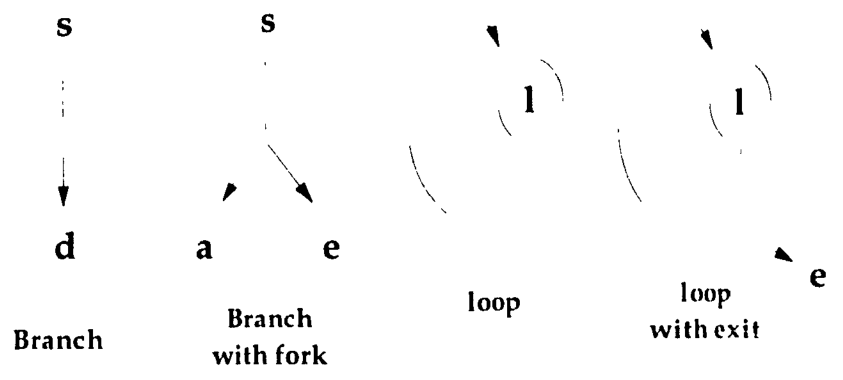


Figure 6.3. Graphical Representation of State - Transition Cases

The state - transition cases of figure 6.3 can be represented using the InfoMap methodology. Figure 6.4 shows this representation.

A	A	A	A	{PARTITION}
v				*Branch*
	v			*Branch with fork*
		v		*Loop*
			v	*Loop with Exit*
L	L	L	L	{STATE}
s	s	l	l	source
d	a			destination 1
	e		e	destination 2
O	O	O	O	{TRANSITION}
o				transition 1
	o			transition 2
		o		transition 3
			o	transition 4

Figure 6.4. InfoMap Representation of State - Transition Cases at the Detailed Level of Abstraction

In each of the columns of figure 6.4 a state - transition case is represented. A source state can reach a destination state through a simple transition ("s" to "d"); a source state can reach an exemption or an assertion state through a transition ("a", "e"); a source state can also be a destination state through a loop transition ("l"); a source state can be a destination state ("l") and reach an exemption state through a transition.

The SetRole chosen to represent a given condition is the "Guard" (letter "G") at the general level of abstraction (InfoSchema). According to the framework described in chapter 3, a SetMemberRole "Guard" can either be true, false or complementary of true or false. The following cases of control in a control flow graph are identified (figure 6.5):

- If -Then - Else: simple branch condition.
- While - Do: iteration condition that checks the condition before it reaches its destination state.
- Repeat - Until: iteration that checks the condition after it has reached its destination state.

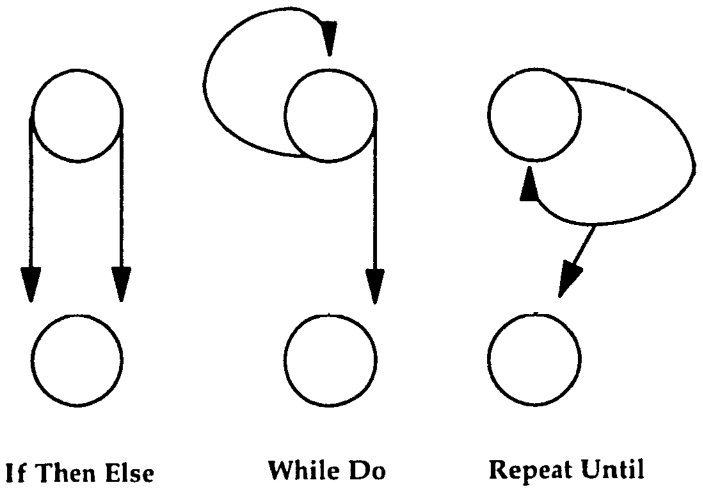


Figure 6.5. Graphical Representation of Control Flow Cases

In order to represent these cases in the InfoMap representation methodology, pre-conditions and post-conditions should be attached to transitions. This is shown in the columns of figure 6.6.

A	A	A	A	A	{PARTITION}
v	v				*If Then Else*
		v	v		*While Do*
				v	*Repeat Until*
L	L	L	L	L	{STATE}
s	s	l	s	l	source
d	d		d	e	destination
O	O	O	O	O	{TRANSITION}
o					transition 1
	o				transition 2
		o			transition 3
			o		transition 4
G	G	G	G	G	{Pre-CONDITION}
t	f	t	f		pre-condition 1
G	G	G	G	G	{Post-CONDITION}
				t	post-condition 1

Figure 6.6. InfoMap Representation of Control Flow Cases at the Detailed Level of Abstraction

The SetMemberRoles allocated to represent sequences of actions are numbers. Therefore each SetMember in the SetName {ACTION} is numbered and attached to a given transition. Figure 6.7 shows a transition from one state to another. On the source end of the transition (left) there exists a set of pre-conditions, and at the end source of the transition (right) there exists a set of post-conditions. Between these two ends there exists a set of actions. These actions can be executed either sequentially or concurrently.

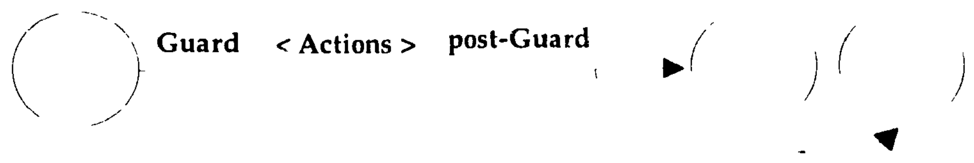


Figure 6.7. Graphical Representation of a Transition with Actions

These sequences of actions attached to each transition can be represented using the InfoMap representation methodology. Figure 6.8 shows the set of actions from one state to another.

A	**{PARTITION}**
v	*Transition with actions*
L	**{STATE}**
s	source
d	destination
O	**{TRANSITION}**
o	transition 1
G	**{Pre-CONDITION}**
t	pre-condition 1
S	**{ACTION}**
1	action 1
3	action 2
2	action 3
4	action 4
G	**{Post-CONDITION}**
t	post-condition 1

Figure 6.8. InfoMap Representation of a Transition with Actions at the Detailed Level of Abstraction

In this section the complete model of the control flow graph is shown in the InfoMap representation methodology. Figure 6.9 shows the overall model of a control flow graph that describes the binary search paradigm [46]. The several SetNames and SetMemberNames are connected with their corresponding SetRoles and SetMemberRoles, in order to provide a complete system for the representation of the binary search paradigm.

L	L	L	L	3	{STATES}
s					Start
d	l	l	s		Search
	e	e	d		End
O	O	O	O	4	{TRANSITIONS}
o					Initialization
	o				Continue "low" unless no more
		o			Continue "high" unless no more
			o		Success
G	G	G	G	2	{CONDITIONS}
	t	f	f		index(Xa,Xmed,1) < Xe
		t	f		index(Xa,Xmed,1) > Xe
S	S	S	S	11	{ACTIONS}
1					set.name("Xlow",1)
6					set.name("Xe",8)
2					set.name("Xhigh",10)
3					set.name("Xfound",false)
4					set.name("Xindexx",1)
	1				set.name("Xlow",Xmed+1)
		1			set.name("Xhigh",Xmed-1)
5	2	2			set.name("Xmed",int((Xlow+Xhigh)/2))
			1		set.name("Xindexx",Xmed)
			3		alert("Found at position: "&Xindexx)
			2		set.name("Xfound",true)
G	G	G	G	1	{Post-CONDITIONS}
	t	t			Xlow<=Xhigh

Figure 6.9. InfoMap Representation of the Binary Search Paradigm at the Detailed Level of Abstraction

6.3 InfoRun System

The control flow graph in figure 6.9 is presented in the InfoMap representation methodology. This control flow graph can be imported in a spreadsheet-based software package and processed producing actual results. A system for this purpose has been developed using the Microsoft EXCEL spreadsheet package. The design of this system is based on the idea that the InfoMap tabular structure can be seen as an array of items. These items can either be column and/or row markers (SetRoles and SetMemberRoles), or SetMembers of SetNames. SetRoles and SetMemberRoles are used for guidance purposes in order to travel through states, execute sequences of actions, and evaluate pre/post-conditions.

SetMembers can be state and/or transition names, or expressions that evaluate as true or false and are listed in the {Pre-CONDITION} and/or {Post-CONDITION} SetNames. In this section we describe the InfoRun System.

The InfoRun System is composed of three parts.

- The Interface: An altered Microsoft EXCEL spreadsheet showing the control flow graph using the InfoMap representation methodology and the push buttons that control its execution. This is shown in figure 6.10.

- The Inference Mechanism: A Microsoft EXCEL macrosheet containing the source code of the system.

- The Data File: A Microsoft EXCEL macrosheet containing the data objects that the control flow graph is using.

6.3.1 The InfoRun Interface

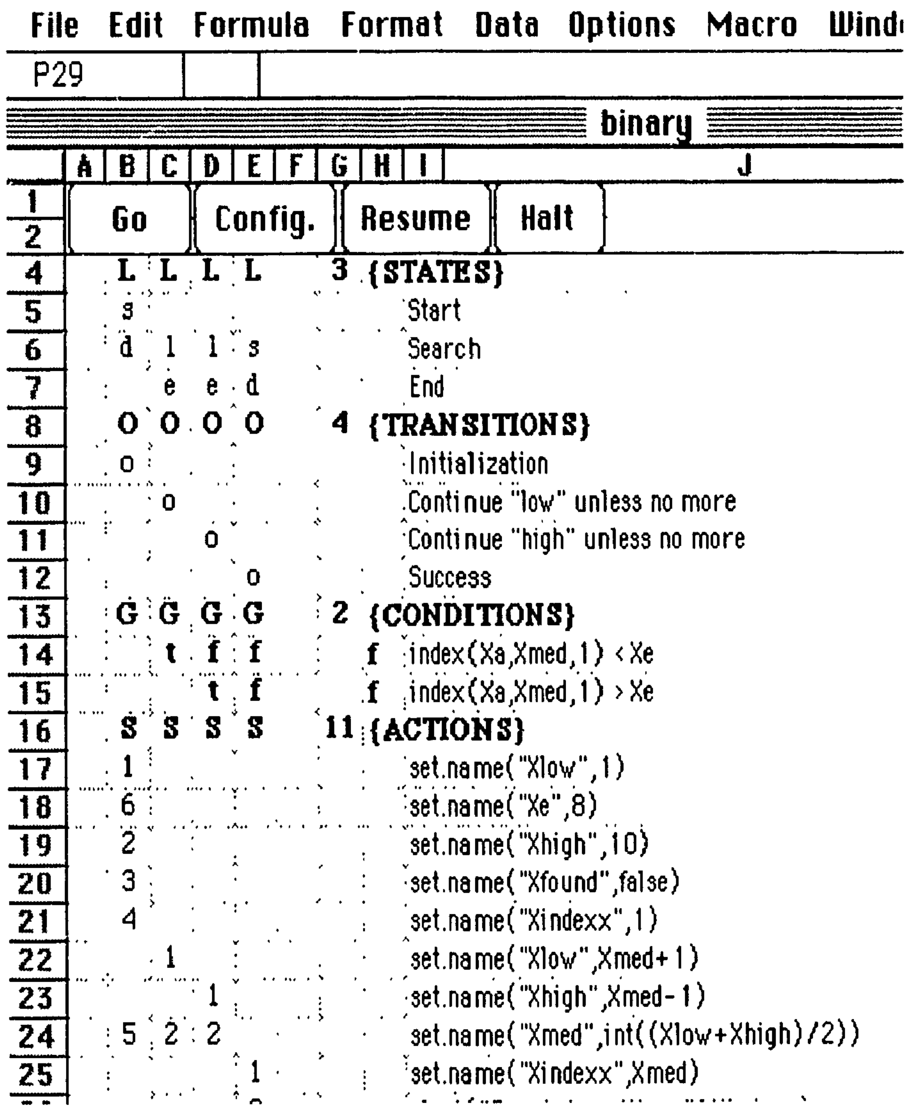
The InfoRun interface is a set of push buttons on the top part of a spreadsheet, containing a control flow graph represented using the InfoMap representation methodology. These push buttons are shown in figure 6.10, and their functions are the following:

- Go: Initializes and starts the system.

- Config: Configures the execution parameters of the system.

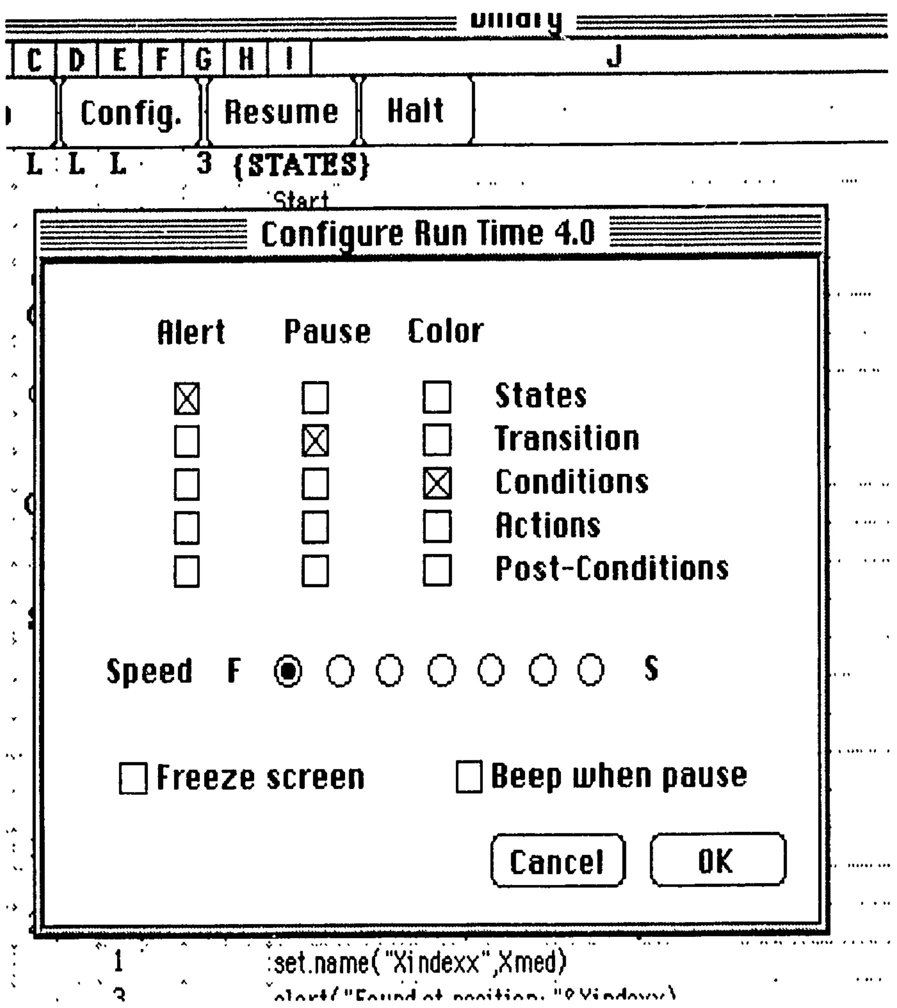
- Resume: Resumes a paused execution.

- Halt: Halts a paused execution.



	A	B	C	D	E	F	G	H	I / J
4		L	L	L	L		3	{STATES}	
5		s							Start
6		d	1	1	s				Search
7			e	e	d				End
8		O	O	O	O		4	{TRANSITIONS}	
9		o							Initialization
10			o						Continue "low" unless no more
11				o					Continue "high" unless no more
12					o				Success
13		G	G	G	G		2	{CONDITIONS}	
14			t	f	f			f	index(Xa,Xmed,1) < Xe
15				t	f			f	index(Xa,Xmed,1) > Xe
16		S	S	S	S		11	{ACTIONS}	
17		1							set.name("Xlow",1)
18		6							set.name("Xe",8)
19		2							set.name("Xhigh",10)
20		3							set.name("Xfound",false)
21		4							set.name("Xindexx",1)
22			1						set.name("Xlow",Xmed+1)
23				1					set.name("Xhigh",Xmed-1)
24		5	2	2					set.name("Xmed",int((Xlow+Xhigh)/2))
25					1				set.name("Xindexx",Xmed)

Figure 6.10. InfoRun Interface

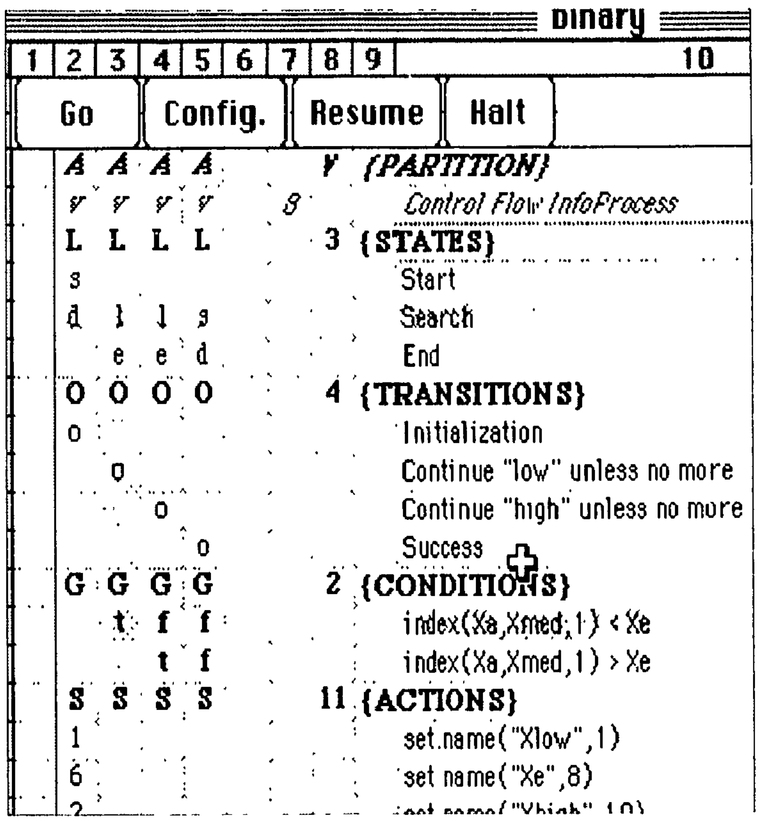


	Alert	Pause	Color
States	☒	☐	☐
Transition	☐	☒	☐
Conditions	☐	☐	☒
Actions	☐	☐	☐
Post-Conditions	☐	☐	☐

Speed F ● ○ ○ ○ ○ ○ ○ S

☐ Freeze screen ☐ Beep when pause

Figure 6.11. InfoRun Configuration Dialog Box



```
binary
 1  2  3  4  5  6  7  8  9                 10
 Go    Config.    Resume    Halt
    A  A  A  A           Y  {PARTITION}
    Y  Y  Y  Y        8        Control Flow InfoProcess
    L  L  L  L           3  {STATES}
    s                          Start
    d  l  l  s                 Search
       e  e  d                 End
    O  O  O  O           4  {TRANSITIONS}
    o                          Initialization
       o                       Continue "low" unless no more
          o                    Continue "high" unless no more
             o                 Success
    G  G  G  G           2  {CONDITIONS}
       t  f  f                 index(Xa,Xmed,1) < Xe
          t  f                 index(Xa,Xmed,1) > Xe
    S  S  S  S          11  {ACTIONS}
    1                          set.name("Xlow",1)
    6                          set name("Xe",8)
    2
```

Figure 6.12. InfoRun in Action

The configuration parameters of the system can be controled by a pop-up dialog box. This dialog box is shown in figure 6.11. It controls the following:

- The speed of the execution.
- Whether an "alert" message should be displayed when a SetMember of any SetName is processed.
- Whether a SetMember that is processed should be animated in color.
- Whether the system should pause when a SetMember is reached.
- The "beep" sound for the alert message on/off mode.
- The "freeze screen" option to switch off all of the above.

A typical layout of the screen during an execution run of a process is shown in figure 6.12.

6.3.2 The InfoRun Inference Mechanism

The inference mechanism of the InfoRun system is a Microsoft EXCEL macro [47], a program that acts on the data displayed on a spreadsheet. The inference mechanism of the InfoRun system is based on the principles of the control flow execution described in section 6.2. In order to apply these principles, we view each array of the SetMemberRoles and the SetMembers of the SetNames in the control flow graph as distinct arrays of information. These arrays are indexed according to rows and columns. Therefore it is possible to access a specific row and/or column location, and process the information that is contained in this location. These arrays are shown in figure 6.13.

L	L	L	L	{STATE}
	array			*array*
O	O	O	O	{TRANSITION}
	array			*array*
G	G	G	G	{PreCONDITION}
	array			*array*
S	S	S	S	{ACTION}
	array			*array*
G	G	G	G	{PostCONDITION}
	array			*array*

Figure 6.13. InfoRun Arrays for the Processing of Rows and Columns

Based on this array structure, the inference mechanism uses a guidance system to travel through the SetName {STATE} following the SetMemberNames of the SetName {TRANSITION}, while evaluating conditions and executing actions. In order to travel from one state to another, the system evaluates the conditions listed in SetNames {Pre-CONDITION} and {Post-CONDITION}. These two SetNames list Microsoft EXCEL expressions as their SetMembers. These can either be evaluated as true or false. SetMemberNames listed in the SetName {ACTION} are Microsoft EXCEL functions that act on the data objects contained in the "data-file" (section 6.3.3).

The arrays of the SetMemberRoles in the SetNames {PreCONDITION} and {PostCONDITION} contain true or false entries (letters "t", "f"). These entries are evaluated against their corresponding expressions in each row of the SetNames {PreCONDITION} and {PostCONDITION} with the "xor" operation. Furthermore, the array containing the SetMemberRoles of the SetName {ACTION} is used to specify the sequence of execution of the functions that are listed in the SetName {ACTION}.

The inference mechanism is based on the idea that expressions and actions can be translated into any executable programming language. Furthermore, the information provided by the SetRoles can be used to guide the control flow graph through its states. These SetRoles are contained in arrays corresponding to each SetName.

6.3.3 The Data File

The data file is a Microsoft EXCEL macro-sheet, used to hold the input data of a control flow graph, like the one that appears in figure 6.10. In addition to that, it is used to execute the actions listed in the SetName {ACTION}, and to evaluate the expressions listed in the SetNames {PreCONDITION} and {PostCONDITION}. These actions and expressions can not be executed and/or evaluated directly on the spreadsheet where the control flow graph appears. Therefore, first they must be copied onto the macro-sheet, and second executed and/or evaluated [46].

6.4 Design Pattern Source Code Modeling

In chapter four of this thesis we described design patterns, and used the InfoMap representation methodology to model them. The control flow model described in section 6.2 of this chapter can be used to model the source code of a given design pattern. The template of information describing a control flow graph is a design pattern part, described in section 4.4 of this thesis and shown in figure 4.5.a. In this section we present the C++ program source code that is part of the "Interpreter" design pattern. This program is used to evaluate Boolean expressions [1]. The source code of class BooleanExp is shown in figure 6.14, and its corresponding model in the InfoMap representation methodology is shown in figure 6.15. This model is imported and traced with the InfoRun system. The model shows how messages can be sent to the BooleanExp, class and how the corresponding code can be executed in terms of a control flow graph. This control flow graph first requests the message to be executed, and then decides whether the message is a "copy", "evaluate" or "replace" message. After it has decided, it responds by displaying the source code which responded to the message. Instead of displaying a message, the system can be upgraded in order to use the "subscribe" facility with dynamic data link [46, 47]. Therefore it is possible to pass the SetMembers of the SetName {ACTION} that contain the source code to the actual source code interpreter.

```
class BooleanExp    {
public:
BooleanExp();
virtual ~BooleanExp();
virtual bool Evaluate (Context&) = 0;
virtual BooleanExp* Replace (const char*, BooleanExp&) = 0;
virtual BooleanExp* Copy() const = 0;
                          }
```

Figure 6.14. Interpreter Design Pattern Source Code for Class BooleanExp

```
A  A  A  A      V {PARTITION}
v  v  v  v    8     BooleanExp for Interpreter
L  L  L  L      5 {STATES}
s                   start BooleanExp
d  s  s  s          selected operation
   d                execute code for Evaluate
      d             execute code for Replace
         d          execute code for Copy
O  O  O  O      4 {TRANSITIONS}
o                   select operation
   o                respond to message Evaluate
      o             respond to message Replace
         o          respond to message Copy
G  G  G  G      3 {CONDITIONS}
   t                (message = "evaluate")
      t             (message = "replace")
         t          (message = "copy")
S  S  S  S      4 {ACTIONS}
1                   set.name("message", input("respond to message (evaluate / replace / co
   1                alert("virtual bool Evaluate(Context&) = 0;")
      1             alert("virtual BooleanExp* Replace (const char*, BooleanExp&) - 0,")
         1          alert("virtual BooleanExp* Copy() const = 0;")
G  G  G  G      1 {Post-CONDITIONS}


A  A  A  A      1 {dataSHEET}
V  V  V  V          datasheet1
A  A  A  A      A {AUTHOR}
v  v  v  v      v   InfoSCHEMA Copyright © W.M. Jaworski
v  v  v  v      v   InfoMAP Copyright © A.A.Michailidis
```

Figure 6.15. InfoMap Model Corresponding to figure 6.14 Imported in the InfoRun System

6.5 Graph Tracing with the InfoRun System

It is possible to perform more than just tracing of a process with the InfoRun system. The main advantage of this system is that we can alter the SetMemberRoles of all the SetNames during execution. This alteration during execution allows us to manipulate and experiment with the process during run-time. The changes can be made by selecting to pause the execution. This can be accomplished if the system is configured to pause at states, transitions, conditions and actions. The changes and their effects can take place in the {STATE}, {TRANSITION}, {pre/post-CONDITION} and {ACTION} SetNames.

Changes in the {STATE} and/or {TRANSITION} SetNames: We can move and/or delete source and/or destination states. We can also add new states and connect them to the current model. In effect, what we can accomplish is the redirection of transitions from one state to another, or the expansion of the control flow graph. An example is shown in figures 6.16.a and 6.16.b. The original state - transition layout is displayed in part [a]. In part [b] of the same figure a new state and a new transition are added.

Changes in the {pre/post-CONDITION} SetNames: We can move and/or delete true and/or false SetMemberRoles. We can also add new conditions in both the {pre-CONDITION} and {post-CONDITION} SetNames. These changes can alter the model of a control flow graph by adding and/or removing conditions from transitions. Therefore the model can become more flexible. An example is shown in figures 6.16.c and 6.16.d. The original conditions' layout is displayed in part [c]. In part [d] of the same figure the SetMemberRole "t" was moved from the first condition to the second condition.

Changes in the {ACTION} SetName: We can add new actions by listing and enumerating them in the SetName {ACTION}. We can also disable actions by removing their SetMemberRoles, and/or re-sequence actions by re-sequencing their SetMemberRoles. An example is shown in figures 6.16.e. and 6.16.f. In part [e] the sequence of actions is: 1, 5, 2, 3, 4. In part [f] the sequence of the same actions was changed to: 1, 2, 5, 3, 4.

L	L	L	L	3 {STATES}
s				Start
d	l	l	s	Search
	c	e	d	End
O	O	O	O	4 {TRANSITIONS}
o				Initialization
	o			Continue "low" unless no more
		o		Continue "high" unless no more
			o	Success

a. Original State - Transition Layout

L	L	L	L	L	3 {STATES}
s					Start
d	$\boxed{s}$	l	l	s	**Search**
	$\boxed{d}$				New State
		c	e	d	End
O	O	O	O	O	4 {TRANSITIONS}
o					Initialization
	o				**New Transition**
		o			Continue "low" unless no more
			o		Continue "high" unless no more
				o	Success

b. Modified State - Transition Layout

G	G	G	G	2 {CONDITIONS}
		$\boxed{t}$		index(Xa,Xmed,1) < Xe
				index(Xa,Xmed,1) > Xe

c. Original Conditions Layout

G	G	G	G	2 {CONDITIONS}
				index(Xa,Xmed,1) < Xe
		$\boxed{t}$		index(Xa,Xmed,1) > Xe

d. Modified Conditions Layout

S	S	S	S	# {ACTIONS}
		1		set name("Xlow",1)
		$\boxed{5}$		set name("Xe",8)
		$\boxed{2}$		set name("Xhigh",10)
		3		set name("Xfound",false)
		4		set.name("Xindexx",1)

e. Original Actions Layout

S	S	S	S	# {ACTIONS}
		1		set name("Xlow",1)
		$\boxed{2}$		set name("Xe",8)
		$\boxed{5}$		set name("Xhigh",10)
		3		set name("Xfound",false)
		4		set name("Xindexx",1)

f. Modified Actions Layout

Figure 6.16. Modification of SetMembers and SetMemberRoles of the InfoMap Representation of Control Flow Graphs

6.6 Summary, Deliverables

The InfoMap tabular structure allows us to represent concepts using sets, roles and their relationships. Spreadsheet software packages can be used to accommodate the InfoMap tabular structure. In this chapter we have shown what constitutes a control flow graph, and how it can be modeled using the InfoMap representation methodology. In addition, with the help of the EXCEL spreadsheet program we have demonstrated the InfoRun system that executes control flow graphs. A design pattern source code, as well as the binary search paradigm, was imported and traced with the InfoRun system. Finally we have demonstrated the capabilities of the InfoRun system. These capabilities include the addition, removal, and modification of the several SetMemberRoles and

SetMembers of the model. These modifications allow us to change the layout of the control flow model at run-time, by choosing to pause the InfoRun system during its execution.

CHAPTER 7: CONCLUSION

7.1 Retrospect

It has come to the attention of most readers of new methodologies that several aspects involving the notation of these methodologies can be confusing and misleading. Furthermore, in several cases the developers of new methodologies do not think about the notation they are using for recording their research, but they mostly think about how to match their methodology against existing problems in the world. Therefore most of the time they ignore their prime objective of transferring their knowledge through understandable notations, by introducing complex structures.

In this work we have presented a notation that is the backbone of the InfoMap methodology. It is a technique for modeling concepts that can be represented with sets and relationships among sets. The application of the technique produces tabular structured environments that can be manipulated if imported into automated systems. A design framework for this methodology is presented, along with its description in both its own terms and in terms of a CASE tool. The framework that accompanies the notation focuses more on the reduction of complexities. This was demonstrated by the application of the framework on the representation of the several parts that constitute a design pattern. Furthermore, several characteristics of the InfoMap methodology involving designs were compared to the corresponding characteristics of design patterns.

The InfoRun tool was presented. It belongs to a set of tools under development for the manipulation of InfoMap tabular structured representations. The focus of this tool is the tracing and execution of control flow graphs represented using the InfoMap methodology. Furthermore this tool, was applied to specify parts of source code found in design patterns.

Overall we have reached the conclusion that a repository system for the communication of design patterns is needed. The InfoMap methodology and its notation can accommodate such a system. Design patterns and InfoMap models share several characteristics, therefore a repository system based on the InfoMap methodology and its notation can easily be used to store and manipulate any useful software design pattern. Weather a design pattern describes object-oriented compositions of classes and objects or hierarchical structures or processes, it has already be shown by several users of the InfoMap methodology and its notation that a repository system based on the InfoMap technology can provide effective and efficient means for the communication of design patterns.

7.2 Future Research

We do not claim that this methodology is flawless. There are several parts of it that still need to be analyzed. Some of these parts are:

- A non-deterministic knowledge modeling algorithm is needed in order to map a problem, probably presented in natural language form, to the context-free grammar production rules described in chapter 2.

- Further research and automation in the area of transforming context-free grammar specifications into frameworks composed of abstract and concrete classes.

- Automation of the InfoMap design framework described in chapter 3, by the use of the c++ code generator contained in the Rational Inc. CASE tool.

- The use of the InfoMap representation to establish a verification technique for design patterns.

Even though the idea of the tabular structure that characterizes the InfoMap methodology is not new, tools for manipulating its tabular structure are relatively new. The InfoRun system described in chapter 6 is a prototype for understanding the control flow processes presented using the InfoMap methodology. Therefore it is not fully automated. A fully automated version should include verification and validation options for the control flow graphs represented using the InfoMap methodology. Furthermore, the tool should not be limited to the capabilities provided by its software platform (*i.e.* Microsoft

EXCEL). This platform provides many advantages because it is based on the idea of data manipulation displayed on a spreadsheet. On the other hand, there is considerable calculation overhead that is not needed. Therefore in order to increase the speed of the InfoRun system, the internal unnecessary spreadsheet calculations should be eliminated. Consequently according to the opinion of this researcher, the InfoRun system should be built as a standalone entity.

References

[1] Gamma, E., Helm, R., Johnson, R., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.

[2] Firesmith, D., "Object-Oriented Requirement Analysis and Logical Design", Wiley, 1993.

[3] Sowa, J.F., Zachman, J.A., "Extending and Formalizing the Framework for Information Systems Architecture", *IBM Systems Journal*, **31**(3):590-616, 1992.

[4] Amble, T., "Logic Programming and Knowledge Engineering", Addison-Wesley, 1987.

[5] Sowa, J.F., "Conceptual Structures: Information Processing in Mind and Machines", Addison-Wesley, 1984.

[6] Sowa, J.F., "Towards the Expressive Power of Natural Language, Principles of Semantic Networks", Morgan Kaufmann, 1991.

[7] Winston, P., "Artificial Intelligence", Addison-Wesley, 1984.

[8] Shinghal, R., "Formal Concepts in Artificial Intelligence", Chapman and Hall, 1992.

[9] Zahniser, R.A., "Design by Walking Around", *Communications of the ACM*, **36**(10), 1993.

[10] Jaworski, W.M., Michailidis, A.A., "Recovery and Enhancements of System Patterns: InfoSchemata and InfoMaps", Proceedings of the 3rd Annual North Test Workshop, Lowell, Massachusetts, 1994.

[11] Berzins, V., Luqi, L., "Software Engineering with Abstractions", Addison-Wesley, 1991.

[12] Butler, G., Grogono, P., Shinghal, R., Tjandra, I., "A Process Algebra for Data Flow Diagrams", Department of Computer Science, Concordia University, 1994.

[13] Iglewski M., Madey, J., Parnas, L., Kelly, P., "Documentation Paradigms, A Progress Report", Telecommunications Research Institute of Ontario, CRL Report No. 270, 1993.

[14] James, A., "Natural Language Understanding",The Handbook of Artificial Intelligence, edited by Barr, A., Feigenbaum, A., Stanford, Calif. : HeirisTech Press, 4:195-238.

[15] "Rational Rose/C++ CASE Tool v2.0.14., On-line Reference Manual", Rational Inc., 1994.

[16] Lewis, H., Papadimitriou, C., "Elements of the Theory of Computation", Prentice-Hall, 1981.

[17] Gamma, E., Helm, R., Johnson, R., "Design Patterns: Abstraction and Reuse of Object-Oriented Design", ECOOP 1993 Conference Proceedings, Springer-Verlag, Lecture Notes in Computer Science, **707**, 1993.

[18] Kattou, S., "Synthetic and Reusable Products of the Software Development Process Through InfoSchemata", Masters Thesis, Department of Computer Science, Concordia University, 1992.

[19] Cheng J., "A Reusability-Based Software Development Environment", *SIGSOFT Software Engineering Notes*, **19**(2):57-62, 1994.

[20] Foote, B., Johnson, R., "Designing Reusable Classes", Journal of Object-Oriented Programming, Department of Computer Science, University of Illinois at Urbana-Champaign, June/July 1988.

[21] Beck K., Johnson R., "Patterns Generate Architectures", ECOOP 94 Conference Proceedings, Springer-Verlag, Lecture Notes in Computer Science, **821**, 1994.

[22] Nelson, C., "A Forum for Fitting the Task", *IEEE Computer*, **27**(3):104, 1994.

[23] Booch, G., "Object-Oriented Analysis and Design with Applications", Benjamin/Cummings, 1994.

[24] Schmucker, K., "Object Oriented Programming for the Macintosh", Hayden Book Company, 1986.

[25] "Lisa Toolkit v3.0", Apple Computer Inc., Cupertino, 1984.

[26] Goldberg A., "Smalltalk-80: The Interactive Programming Environment", Addison-Wesley, 1984.

[27] "Building Object-Oriented Frameworks: A Taligent White Paper", Taligent Inc. World Wide Web: http://www.taligent.com/resources-list.html.

[28] Hawkins J., "The Oxford Encyclopedic English Dictionary", Cleanroom Press, 1991.

[29] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., "Object Oriented Modeling and Design", Prentice-Hall, 1991.

[30] Aho, A., Sethi, R., Ulman, J., "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1988.

[31] Beck, K., "Patterns and Software Development", *Dr. Dobb's Journal,* **19**(2):18-22, 1994.

[32] Alexander, C., Ishikawa, S., Silverstein, M., "A Pattern Language", Oxford University Press, 1977.

[33] Alexander, C., "Notes on the Synthesis of Forum", Harvard University Press, 1964.

[34] Alexander, C., "A Timeless Way of Building", Oxford University Press, 1979.

[35] Tracs, W., "Software Reuse Myths", *ACM SIGSOFT Software Engineering Notes,* **13**(1):17-21, 1988.

[36] Lea, D., "Christopher Alexander: An Introduction for Object-Oriented Designers", *ACM SIGSOFT Software Engineering Notes,* **19**(1):39-46, 1994.

[37] Alexander, C., "The Linz Cafe", Oxford University Press, 1981.

[38] "IEF Technical Description: Methodology and Technology Overview", Texas Instruments Inc., 1992.

[39] "SOMATiC CASE Tool On-Line Reference Manual v1.20", Bezant Ltd., 1994.

[40] Bourgault, P., "InfoADM3: A System Supporting Knowledge/Information Workers", COMP457/657, Office Automation, Department of Computer Science, Concordia University, 1994.

[41] Benc, T., "InfoSchemata and InfoCases for an Object-Oriented Methodology", COMP458/658, Structure of Information Systems, Department of Computer Science, Concordia University, 1994.

[42] "infoFarm v1.0 User's Manual", GS General Strategies, Ottawa, 1991.

[43] Cummings, T., "A Knowledge Acquisition Method: Transformation of Algorithms and Programs with infoMAPs", Masters Thesis, Department of Computer Science, Concordia University, 1991.

[44] Jaworski, W.M., Cummings, T., "Program Normalization and Optimization using InfoMaps as an Inspection and Programming Processing Tool", Canadian Conference on Electrical and Computer Engineering, Québec City, September 1991.

[45] Aho, V., "Data Structures and Algorithms", Addison-Wesley, 1983.

[46] "Microsoft EXCEL v4.0 User's Manual Reference", Microsoft Corporation 1993.

[47] "Microsoft EXCEL v4.0 Function Reference Manual", Microsoft Corporation 1993.

[48] Kronic, M., "An ABL Software Environment for a Mini-Computer", Masters Thesis, Department of Computer Science, Concordia University, 198.

[49] Ficocelli, L., "Problems to Programs: A Humanistic Approach (An Introduction to ABL MEthodology)", Masters Thesis, Department of Computer Science, Concordia University, 1983.

[50] Deslauriers, B., "Inspection of Software Deliverables: An InfoMap-Based Methodology", Masters Thesis, Department of Computer Science, Concordia University, 1991.

[51] Finkelstein, K., "A Prototype of an ABL Syntax-Driven Editor Supporting Software Development", Masters Thesis, Department of Computer Science, Concordia University, 1983

[52] Eddy, D., "An Environment Conducive to Software Design: JMSS System for Modeling of Software Processes", Masters Thesis, Department of Computer Science, Concordia University, 1988

[53] Hinterberger, H., Jaworski, W. M., "Controlled Program Design by use of the ABL Programming Concept", Angewandte Informatik, Weisbaben, West Germany, July 1981, pp. 302-310.

[54] Jaworski, W.M., Zaborowski, B., "Network Models for Design of Information Systems" (Modele Sieciowe w Prejektowaniu Systemów Przetwarzania Informacji w CROPI), *Maszyny Matematyczne*, 3(3): 26-31, 1967.

[55] Jaworski, W.M., Ficocelli, L., O'Mara, K.S., "The ABL/W4 Methodology for System Modelling", *System Research*, 4(1):23-37, 1987.

[56] Jaworski, W.M., Radhakrishnan, T., "Modelling of System Development Methodologies", CompInt 1987 Proceedings, Conference on Computer Aided Technologies, November 9-12, Montreal 1987.

[57] Jaworski, W.M., "An Interactive System for the Generation of Programs from Decision Tables", Computer Aided System Simulation, Analysis and Design Project (CASSAD '70), University of Houston, Houston 1970.

[58] "Leonardo da Vinci : engineer and architect / The Montreal Museum of Fine Arts", Montreal Museum of Fine Arts, May 22 1987.

APPENDIX A. Rational Rose/C++ CASE Tool Results

Class name:

PartitionSg

Documentation:
PartitionSg -> {Sg}

Export Control.	Public
Cardinality.	n
Hierarchy:	
Superclasses:	none
Public Interface:	
Has-A Relationships:	
	Sg
Operations.	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence	Transient

Operation name.

exists

Public member of.		PartitionSg
Arguments:		
	SgPartition	Sg

Documentation:
operation exists is defined as the universal quantifier for a partition of set names and set roles

Concurrency: Sequential

Class name:

Sg

Documentation:
Sg -> {SetRole SetName}

Export Control:	Public
Cardinality.	n
Hierarchy	
Superclasses:	PartitionSg
Public Interface:	
Has-A Relationships:	
	SetName
	SetRole
Operations:	
	exists

State machine.	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	Sg	
Arguments:		
expression	ITEM	
statement	ITEM	
UpperCaseLetter		SetRole

Documentation:
operation exists is applied to SetRole / expressions / statements

Concurrency: Sequential

Class name:
SetRole

Documentation:
SetRole -> {SetRoleDominant I SetRoleDescriptive I SetRoleTransitive I SetRoleUserDefined}

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses:	Sg
Public Interface:	
Has-A Relationships:	
	SetRoleUserDefined
	SetRoleDominant
	SetRoleTransitive
	SetRoleDescriptive
Operations:	
	exists

State machine: Yes
Concurrency: Sequential
Persistence: Transient

Operation name:
exists

Public member of:	SetRole	
Arguments:		
UpperCaseLetter		SetRole
Concurrency:	Sequential	

Class name:
SetName

Documentation:
SetName -> "{"literal"}"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	Sg
Public Interface:	
Operations:	
	exists

State machine.	Yes
Concurrency:	Sequential
Persistence.	Transient

Operation name:

exists

Public member of:		SetName
Arguments:		
	literal	SetName

Documentation·

operation exists is defined for a SetName that is a literal

Concurrency: Sequential

Class name·

SetRoleDominant

Documentation:

SetRoleDominant -> SetRoleIdentifier I SetRoleIdentity I SetRoleAggregation I SetRoleGeneralization I SetRoleHierarchy

Export Control:	Public
Cardinality.	1
Hierarchy:	
Superclasses:	SetRole
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SetRoleDominant	
Arguments·		
	GenericDominantUpperCaseLetter	Dominant
Concurrency:	Sequential	

Class name:

SetRoleDescriptive

Documentation:

SetRoleDescriptive -> SetRoleQualifier I SetRoleAssociation I SetRoleFlow I SetRoleGuard I SetRoleSequence I SetRoleValue

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRole
Public Interface:	

Operations:		
	exists	
State machine:	Yes	
Concurrency:	Sequential	
Persistence:	Transient	

Operation name:

exists

Public member of:	SetRoleDescriptive	
Arguments:		
	GenericDescriptiveUpperCaseLetter	Descriptive
Concurrency:	Sequential	

Class name:

SetRoleTransitive

Documentation:
SetRoleTransitive -> SetRoleSequencial I SetRoleConcurrent

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRole
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of.	SetRoleTransitive	
Arguments:		
	GenericTransitiveUpperCaseLetter	Transitive
Concurrency:	Sequential	

Class name:

SetRoleUserDefined

Documentation:
SetRoleUserDefined -> new set roles invented by the InfoSchema/InfoMap technology users

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRole
Public Interface:	
Operations:	

	exists	

State machine	Yes
Concurrency	Sequential
Persistence.	Transient

Operation name

exists

Public member of·	SetRoleUserDefined	
Arguments		
	GenericUserDefinedUpperCaseLetter	UserDefined
Concurrency.	Sequential	

Class name

SetRoleIdentity

Documentation:
SetRoleIdentity -> "O"

Export Control·	Public
Cardinality	1
Hierarchy.	
Superclasses·	SetRoleDominant
Public Interface:	
Operations	
	exists

State machine·	Yes
Concurrency.	Sequential
Persistence.	Transient

Operation name·

exists

Public member of	SetRoleIdentity	
Arguments		
	UpperCaseLetter	"O"
Concurrency·	Sequential	

Class name.

SetRoleIdentifier

Documentation.
SetRoleIdentifier -> "K"

Export Control·	Public
Cardinality	1
Hierarchy	
Superclasses	SetRoleDominant
Public Interface.	
Operations.	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SetRoleIdentifier	
Arguments:		
	UpperCaseLetter	"K"
Concurrency:	Sequential	

Class name:

SetRoleGeneralization

Documentation:
SetRoleGeneralization -> "I"

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses:	SetRoleDominant
Public Interface:	
Operations.	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence.	Transient

Operation name·

exists

Public member of:	SetRoleGeneralization	
Arguments:		
	UpperCaseLetter	"I"
Concurrency:	Sequential	

Class name:

SetRoleAggregation

Documentation:
SetRoleAggregation -> "P"

Export Control:	Public
Cardinality:	1
Hierarchy.	
Superclasses.	SetRoleDominant
Public Interface:	
Operations:	
	exists

State machine: Yes

Concurrency:	Sequential
Persistence	Translent

Operation name.

exists

Public member of	SetRoleAggregation	
Arguments		
	UpperCaseLetter	"P"
Concurrency·	Sequential	

Class name·

SetRoleHierarchy

Documentation:

SetRoleHierarchy -> "H"

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses·	SetRoleDominant
Public Interface:	
Operations.	
	exists

State machine	Yes
Concurrency.	Sequential
Persistence:	Translent

Operation name.

exists

Public member of:	SetRoleHierarchy	
Arguments:		
	UpperCaseLetter	"H"
Concurrency:	Sequential	

Class name·

SetRoleQualifier

Documentation

SetRoleQualifier -> "X"

Export Control·	Public
Cardinality·	n
Hierarchy:	
Superclasses.	SetRoleDescriptive
Public Interface:	
Operations:	
	exists

State machine·	Yes
Concurrency:	Sequential

Persistence:	Transient	

Operation name:

exists

Public member of:	SetRoleQualifier	
Arguments:		
	UpperCaseLetter	"X"
Concurrency:	Sequential	

Class name:

SetRoleAssociation

Documentation·
SetRoleAssociation -> "M"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRoleDescriptive
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SetRoleAssociation	
Arguments:		
	UpperCaseLetter	"M"
Concurrency:	Sequential	

Class name:

SetRoleFlow

Documentation:
SetRoleFlow -> "F"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRoleDescriptive
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of·	SetRoleFlow	
Arguments.		
	UpperCaseLetter	"F"
Concurrency.	Sequential	

Class name·

SetRoleSequence

Documentation:
SetRoleSequence -> "S"

Export Control.	Public
Cardinality.	n
Hierarchy:	
Superclasses·	SetRoleDescriptive
Public Interface·	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence	Transient

Operation name.

exists

Public member of.	SetRoleSequence	
Arguments.		
	UpperCaseLetter	"S"
Concurrency.	Sequential	

Class name:

SetRoleGuard

Documentation.
SetRoleGuard -> "G"

Export Control·	Public
Cardinality:	n
Hierarchy	
Superclasses·	SetRoleDescriptive
Public Interface.	
Operations·	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name

exists

Public member of:	SetRoleGuard	
Arguments:		
	UpperCaseLetter	"G"
Concurrency:	Sequential	

Class name:

SetRoleValue

Documentation:
SetRoleValue -> "V"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRoleDescriptive
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SetRoleValue	
Arguments:		
	UpperCaseLetter	"V"
Concurrency:	Sequential	

Class name:

SetRoleConcurrent

Documentation:
SetRoleConcurrent -> "C"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetRoleTransitive
Public Interface:	
Operations:	
	exists

State machine: Yes

Concurrency.	Sequential
Persistence·	Transient

Operation name:

exists

Public member of.		SetRoleConcurrent
Arguments:		
	UpperCaseLetter	"C"
Concurrency:	Sequential	

Class name.

SetRoleSequencial

Documentation:

SetRoleSequencial -> "L"

Export Control.	Public
Cardinality·	n
Hierarchy.	
Superclasses:	SetRoleTransitive
Public Interface:	
Operations:	
	exists

State machine·	Yes
Concurrency:	Sequential
Persistence·	Transient

Operation name·

exists

Public member of:		SetRoleSequencial
Arguments·		
	UpperCaseLetter	"L"
Concurrency·	Sequential	

Class name:

PartitionSd

Documentation:
PartitionSg -> {Sd}

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	none
Public Interface:	
Has-A Relationships:	
	Sd
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:		PartitionSd
Arguments:		
	SdPartition	Sd

Documentation:
operation exists is defined as the universal quantifier for a partition of set members and set member roles

Concurrency:	Sequential

Class name:

Sd

Documentation:
Sd -> {(SetMemberRole SetMember)}

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	PartitionSd
Public Interface:	
Has-A Relationships:	
	SetMemberName
	SetMemberRole
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of		Sd	
Arguments			
	expression	ITEM	
	statement	ITEM	
	LowerCaseLetter		SetMemberRole

Documentation

operation exists is applied to SetMember / Role / expressions / statements in areas [1] to [4]

Concurrency: Sequential

Class name

SetMemberRole

Documentation:

SMRole -> (SMRDominant I SMRDescriptive I SMRTransitive I SMRUserDefined)

Export Control:	Public
Cardinality	1
Hierarchy	
Superclasses.	Sd
Public Interface	
Has-A Relationships	
	SMRUserDefined
	SMRDominant
	SMRTransitive
	SMRDescriptive
Operations:	
	exists

State machine Yes
Concurrency Sequential
Persistence Transient

Operation name

exists

Public member of	SetMemberRole	
Arguments:		
	LowerCaseLetter	SetMemberRole

Documentation:

exists upper case letter in area [1] column

Concurrency Sequential

Class name:

SetMemberName

Documentation.

SetMemberName -> "{"literal"}"

Export Control	Public
Cardinality.	n
Hierarchy:	
Superclasses	Sd

Public Interface:		
Operations:		
		exists

State machine.	Yes
Concurrency·	Sequential
Persistence·	Transient

Operation name:

exists

Public member of:		SetMemberName
Arguments:		
	literal	SetMemberName

Documentation.

operation exists is defined for a SetMember that is a literal in area [3] row column

Concurrency: Sequential

Class name:

SMRDominant

Documenta..on:

SetMemberRoleDominant -> SetMemberRoleIdentifier I SetMemberRoleIdentity I SetMemberRoleAggregation I SetMemberRoleGeneralization I SetMemberRoleHierarchy

Export Control:	Public
Cardinality.	1
Hierarchy:	
Superclasses:	SetMemberRole
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SMRDominant	
Arguments:		
	GenericDominantLowerCaseLetter	Dominant

Documentation:

exists generic dominant lower case letter in area [1].column

Concurrency· Sequential

Class name:

SMRDescriptive

Documentation:

SMRDescriptive ->SMRQualifier ISMRAssociation I
SMRFlow I SMRGuard I SMRSequence I SMRValue

Export Control·	Public
Cardinality·	n
Hierarchy:	
Superclasses:	SetMemberRole
Public Interface.	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence·	Transient

Operation name:

exists

Public member of: SMRDescriptive

Arguments.

	GenericDescriptiveLowerCaseLetter	Descriptive

Documentation·

exists generic descriptive lower case letter in area [1].column

Concurrency. Sequential

Class name:

SMRTransitive

Documentation:

SMRTransitive -> SMRSequencial I SMRConcurrent

Export Control.	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetMemberRole
Public Interface:	
Operations:	
	exists

State machine·	Yes
Concurrency:	Sequential
Persistence·	Transient

Operation name:

exists

Public member of· SMRTransitive

Arguments.

	GenericTransitiveLowerCaseLetter	Transitive

Documentation.

exists generic transitive lower case letter in area [1] column

Concurrency: Sequential

Class name:

SMRUserDefined

Documentation:

SMRUserDefined -> new set roles invented by the InfoSchema/InfoMap technology users

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SetMemberRole
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of: SMRUserDefined

Arguments:

GenericUserDefinedLowerCaseLetter	UserDefined

Documentation:

exists generic user defined letter used for new set member roles in area [1].column

Concurrency: Sequential

Class name:

SMRIdentity

Documentation:

SMRIdentity -> "o"

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses:	SMRDominant
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of: SMRIdentity

Arguments:

LowerCaseLetter	"o"

Documentation:

lower case letter "o" exists as a set member role identity in area.[1].column

Concurrency:	Sequential

Class name:

SMRIdentifier

Documentation:
SMRIdentifier -> id

Export Control.	Public
Cardinality:	1
Hierarchy.	
Superclasses:	SMRDominant
Public Interface:	
Operations.	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:		SMRIdentifier
Arguments:		
	number	id

Documentation·
id number exists in area[1].column

Concurrency:	Sequential

Class name:

SMRGeneralization

Documentation:
SMRGeneralization -> "p" | "c"

Export Control·	Public
Car.'inality·	1
Hierarchy:	
Superclasses:	SMRDominant
Public Interface:	
Operations:	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name:

exists

Public member of:	SMRGeneralization

Arguments:		
	LowerCaseLetter	"c"
	LowerCaseLetter	"p"

Documentation:
exists lower case letter "c" or "p" in area [1].column

Concurrency: Sequential

Class name:
SMRAggregation

Documentation:
SMRAggregation -> "w" I "c" I "v" I "h" I "m"

Export Control:	Public
Cardinality:	1
Hierarchy.	
Superclasses:	SMRDominant
Public Interface:	
Operations:	
	exists

State machine: Yes
Concurrency: Sequential
Persistence: Transient

Operation name:
exists

Public member of:	SMRAggregation	
Arguments:		
	LowerCaseLetter	"c"
	LowerCaseLetter	"w"
	LowerCaseLetter	"v"
	LowerCaseLetter	"h"
	LowerCaseLetter	"m"

Documentation:
exists lower case "c" I "w" I "v" I "h" I "m" in area [1].column

Concurrency: Sequential

Class name:
SMRHierarchy

Documentation:
SetRoleHierarchy -> "h" I "1..n"

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses:	SMRDominant
Public Interface:	
Operations:	
	exists

State machine: Yes

Concurrency.	Sequential
Persistence	Transient

Operation name

exists

Public member of.	SMRHierarchy	
Arguments·		
	LowerCaseLetter	"h"
	Number	id

Documentation

exists lower case letter "h" or id number in area [1].column

Concurrency	Sequential

Class name:

SMRQualifier

Documentation:

SMRQualifier -> "x"

Export Control.	Public
Cardinality	n
Hierarchy:	
Superclasses:	SMRDescriptive
Public Interface:	
Operations.	
	exists

State machine	Yes
Concurrency	Sequential
Persistence·	Transient

Operation name.

exists

Public member of	SMRQualifier	
Arguments.		
	LowerCaseLetter	"x"

Documentation

exists lower case letter "x" in area [1] column

Concurrency	Sequential

Class name:

SMRAssociation

Documentation·

SMRAssociation -> "v"

Export Control:	Public
Cardinality:	n
Hierarchy	

Superclasses:	SMRDescriptive	
Public Interface:		
Operations·		
	exists	

State machine:	Yes
Concurrency.	Sequential
Persistence:	Transient

Operation name:

exists

Public member of	SMRAssociation	
Arguments:		
	LowerCaseLetter	"v"

Documentation·

exists lower case letter in area [1].column

Concurrency: Sequential

Class name:

SMRFlow

Documentation·

SMRFlow -> "u" I "o"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SMRDescriptive
Public Interface	
Operations	
	exists

State machine:	Yes
Concurrency·	Sequential
Persistence:	Transient

Operation name:

exists

Public member of·	SMRFlow	
Arguments·		
	LowerCaseLetter	"o"
	LowerCaseLetter	"u"

Documentation.

exists lower case letter "o" or "u" in area [1].column

Concurrency: Sequential

Class name:

SMRSequence

Documentation·
SMRSequence -> id

Export Control.	Public
Cardinality:	n
Hierarchy	
Superclasses	SMRDescriptive
Public Interface·	
Operations·	
	exists

State machine.	Yes
Concurrency:	Sequential
Persistence	Transient

Operation name:

exists

Public member of.		SMRSequence
Arguments		
	number	id

Documentation
exists id number in area [1].column

Concurrency	Sequential

Class name:

SMRGuard

Documentation·
SMRGuard -> "t" | "f" | "T" | "F"

Export Control.	Public
Cardinality·	n
Hierarchy·	
Superclasses.	SMRDescriptive
Public Interface.	
Operations	
	exists

State machine·	Yes
Concurrency·	Sequential
Persistence	Transient

Operation name

exists

Public member of.	SMRGuard	
Arguments·		
	LowerCaseLetter	"t"
	LowerCaseLetter	"f"
	UpperCaseLetter	"T"
	UpperCaseLetter	"F"

Documentation:
the letters in area [1].row.column are evaluated with xor operator against the SetMemberNames in area[3].row. the SetMemberNames are ITEM expressions

Concurrency: Sequential

Class name:

SMRValue

Documentation.
SMRValue -> id

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SMRDescriptive
Public Interface:	
Operations:	
	exists

State machine: Yes
Concurrency: Sequential
Persistence: Transient

Operation name:

exists

Public member of:		SMRValue
Arguments:		
	number	id

Documentation:
exists id number in area [1].column

Concurrency: Sequential

Class name:

SMRConcurrent

Documentation:
SMRConcurrent -> "c"

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	SMRTransitive
Public Interface:	
Operations:	
	exists

State machine: Yes
Concurrency: Sequential
Persistence: Transient

Operation name:

exists

Public member of: SMRConcurrent

Arguments:		
	LowerCaseLetter	"c"

Documentation·
exists lower case letter c in area [1].column

Concurrency: Sequential

Class name:

SMRSequencial

Documentation.
SMRSequencial -> "s" I "d" I "a" I "e"

Export Control:	Public
Cardinality	n
Hierarchy·	
Superclasses:	SMRTransitive
Public Interface:	
Operations.	
	exists

State machine:	Yes
Concurrency:	Sequential
Persistence:	Transient

Operation name.

exists

Public member of:	SMRSequencial	
Arguments:		
	LowerCaseLetter	"s"
	LowerCaseLetter	"d"
	LowerCaseLetter	"a"
	LowerCaseLetter	"e"

Documentation:
exists lower case letter "a" or "e" or "s" or "d" in area [1] column

Concurrency· Sequential

APPENDIX B. InfoSchema/InfoMap Models of Appendix A

A	*A*	*A*	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*v*	*v*	*General Level of Abstraction*
v						*Class documentation*
	v					*Has-A Relationship*
		v				*Hierarchy*
			v			*Operation Documentation*
				v		*Argument Type*
					v	*Set Cardinality*
O	**P**	**H**			**21**	**{CLASS NAME}**
X					**21**	**{DOCUMENTATION: CLASS}**
V					**1**	**{CARDINALITY: CLASS}**
M					**3**	**{ATTRIBUTE/CHARACTERISTIC}**
M			**K**		**1**	**{OPERATION}**
M				**M**	**9**	**{ARGUMENTS}**
			X		**3**	**{DOCUMENTATION: OPERATION}**
			M		**1**	**{CONCURRENCY}**
				O	**21**	**{TYPE}**
M					**2**	**{CLASS TYPE}**
A	*A*	*A*	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*A*	*A*	*1*	***{Copyright ©}***
v	*v*	*v*	*v*	*v*		*A. A. Michailidis 1995*

Figure B.1. InfoSchema/InfoMap General Level of Abstraction Modeling of the Rational Inc. Generated Output Specifications (pages 110 to 120).

A	*A*	*A*	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*v*	*v*	*Detailed Level of Abstraction*
v						*Class documentation*
	v					*Has-A Relationship*
		v				*Hierarchy*
			v			*Operation Documentation*
				v		*Argument Type*
					v	*Set Cardinality*
O	**P**	**H**			**21**	**{CLASS NAME}**
X					**21**	**{DOCUMENTATION: CLASS}**
V					**1**	**{CARDINALITY: CLASS}**
M					**3**	**{ATTRIBUTE/CHARACTERISTIC}**
M			**K**		**1**	**{OPERATION}**
M				**M**	**9**	**{ARGUMENTS}**
			X		**21**	**{DOCUMENTATION: OPERATION}**
			M		**1**	**{CONCURRENCY}**
				O	**31**	**{TYPE}**
M					**2**	**{CLASS TYPE}**
A	*A*	*A*	*A*	*A*	*1*	***{REFERENCE}***
v	*v*	*v*	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*A*	*A*	*1*	***{Copyright © }***
v	*v*	*v*	*v*	*v*		*A. A. Michailidis 1995*

Figure B.2. InfoSchema/InfoMap General Level of Abstraction Modeling of the Rational Inc. Generated Output Specifications (pages 121 to 132).

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	*{PARTITION}*
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	*General Level of Abstraction*
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		*Class documentation*
																					v	*Set Cardinality*
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	21	{CLASS NAME}
o																						PartitionSg
	o																					Sg
		o																				SetRole
			o																			SetName
				o																		SetRoleDominant
					o																	SetRoleDescriptive
						o																SetRoleTransitive
							o															SetRoleUserDefined
								o														SetRoleIdentity
									o													SetRoleIdentifier
										o												SetRoleGeneralization
											o											SetRoleAggregation
												o										SetRoleHierarchy
													o									SetRoleQualifier
														o								SetRoleAssociation
															o							SetRoleFlow
																o						SetRoleGuard
																	o					SetRoleSequence
																		o				SetRoleValue
																			o			SetRoleConcurrent
																				o		SetRoleSequential
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	21	{DOCUMENTATION CLASS}
x																						PartitionSg > {Sg}
	x																					Sg > {SetRole SetName}
		x																				SetRole > {SetRoleDominant | SetRoleDescriptive |
																						SetRoleTransitive | SetRoleUserDefined}
			x																			SetName > { literal }
				x																		SetRoleDominant > SetRoleIdentifier | SetRoleIdentity |
																						SetRoleAggregation | SetRoleGeneralization | SetRoleHierarchy
					x																	SetRoleDescriptive > SetRoleQualifier | SetRoleAsscociation |
																						SetRoleFlow | SetRoleGuard | SetRoleSequence | SetRoleValue
						x																SetRoleTransitive > SetRoleSequencial | SetRoleConcurrent
							x															SetRoleUserDefined > new set roles invented by the
																						InfoSchema/InfoMap technology users
								x														SetRoleIdentity -> O'
									x													SetRoleIdentifier -> K"
										x												SetRoleGeneralization -> I
											x											SetRoleAggregation > 'P'
												x										SetRoleHierarchy > 'H'
													x									SetRoleQualifier -> 'X
														x								SetRoleAssociation > M
															x							SetRoleFlow > F
																x						SetRoleGuard > G'
																	x					SetRoleSequence > S
																		x				SetRoleValue -> V
																			x			SetRoleConcurrent -> C'
																				x		SetRoleSequential -> "I
V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	1	{CARDINALITY CLASS}
n	n	1	n	1	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n		Value
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	3	{ATTRIBUTE/CHARACTERISTIC}
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Sequential
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Transient
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Public
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	1	{OPERATION}
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Exists
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	9	{ARGUMENTS}
v																						SgPartition
	v																					expression
	v																					statement
	v	v						v	v	v	v	v	v	v	v	v	v	v	v	v		UpperCaseLetter
			v																			literal
				v																		GenericDominantUpperCaseLetter
					v																	GenericDescriptiveUpperCaseLetter
						v																GenericTransitiveUpperCaseLetter
							v															GenericUserDefinedUpperCaseLetter
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	2	{CLASS TYPE}
v	v	v	v	v	v	v	v															Abstract
								v	v	v	v	v	v	v	v	v	v	v	v	v		Concrete
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	1	*{REFERENCE}*
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		*Rational Inc Case Tool Specifications Generator*
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	1	*{Copyright ©}*
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		*A A Michailidis 1995*

Figure B.3. Class Documentation Partition Corresponding to Figure B.1.

A	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*General Level of Abstraction*
v	*v*	*v*		*Has-A Relationship*
			v	*Set Cardinality*
P	**P**	**P**	**21**	**{CLASS NAME}**
p				PartitionSg
c	p			Sg
	c	p		SetRole
	c			SetName
		c		SetRoleDominant
		c		SetRoleDescriptive
		c		SetRoleTransitive
		c		SetRoleUserDefined
A	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*1*	*{Copyright © }*
v	*v*	*v*		*A. A. Michailidis 1995*

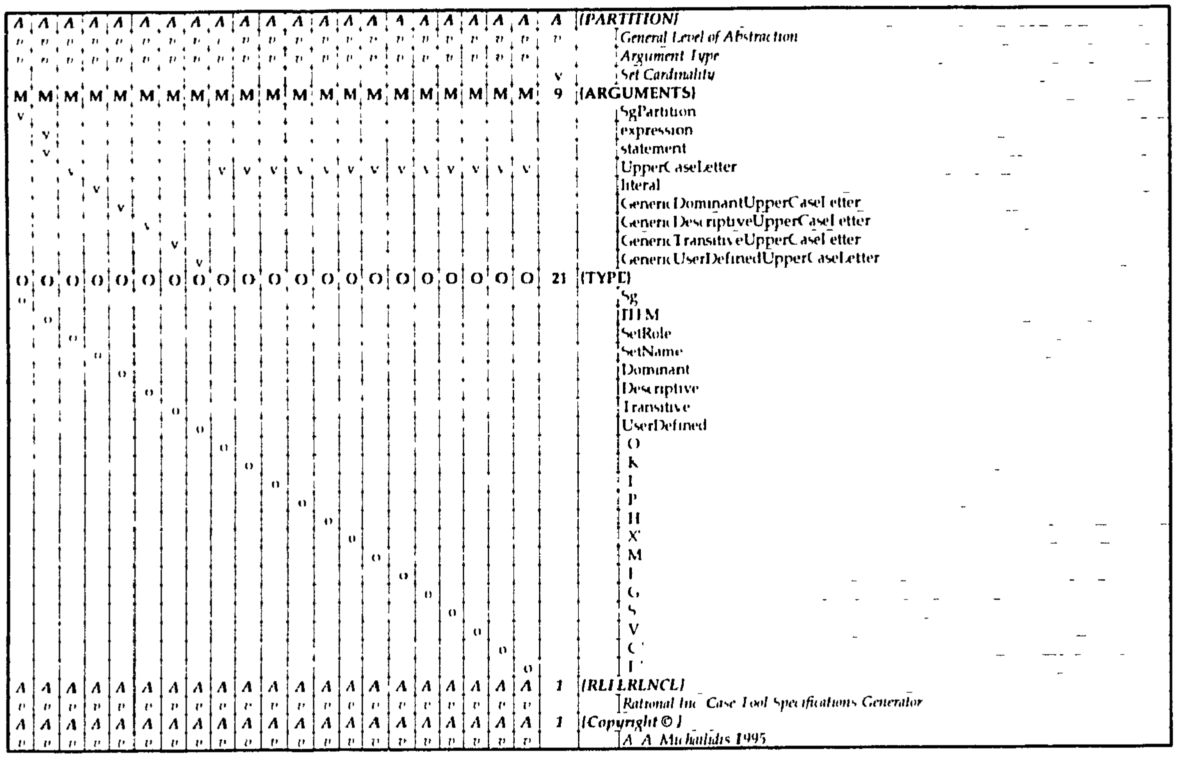
Figure B.4. Has-A Relationship Partition Corresponding to Figure B.1.

A	*A*	*A*	*A*	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*v*	*v*	*v*	*General Level of Abstraction*
v	*v*	*v*	*v*	*v*	*v*		*Hierarchy*
						v	*Set Cardinality*
H	**H**	**H**	**H**	**H**	**H**	**21**	**{CLASS NAME}**
h							PartitionSg
v	h						Sg
	v	h					SetRole
	v						SetName
		v	h				SetRoleDominant
		v		h			SetRoleDescriptive
		v			h		SetRoleTransitive
		v					SetRoleUserDefined
			v				SetRoleIdentity
			v				SetRoleIdentifier
			v				SetRoleGeneralization
			v				SetRoleAggregation
			v				SetRoleHierarchy
				v			SetRoleQualifier
				v			SetRoleAssociation
				v			SetRoleFlow
				v			SetRoleGuard
				v			SetRoleSequence
				v			SetRoleValue
					v		SetRoleConcurrent
					v		SetRoleSequential
A	*A*	*A*	*A*	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*	*v*	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*A*	*A*	*A*	*1*	*{Copyright © }*
v	*v*	*v*	*v*	*v*	*v*		*A. A. Michailidis 1995*

Figure B.5. Hierarchy Relationship Partition Corresponding to Figure B.1.

A	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*General Level of Abstraction*
v	*v*	*v*		*Operation Documentation*
			v	*Set Cardinality*
K	**K**	**K**	**1**	**{OPERATION}**
1	2	3		Exists
X	**X**	**X**	**3**	**{DOCUMENTATION: OPERATION}**
x				operation exists is defined as the universal quantifier for a partition of set names and set roles
	x			operation exists is applied to SetRole/expressions/statements
		x		operation exists is defined for a SetName that is a literal
M	**M**	**M**	**1**	**{CONCURRENCY}**
v	v	v		Sequential
A	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*1*	***{Copyright © }***
v	*v*	*v*		*A. A. Michailidis 1995*

Figure B.6. Operation Documentation Partition Corresponding to Figure B.1.



A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	*{PARTITION}*
p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	*General Level of Abstraction*
p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p		*Argument Type*
																					v	*Set Cardinality*
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	9	**{ARGUMENTS}**
v																						SgPartition
	v																					expression
	v																					statement
		v						v	v	v	v	v	v	v	v	v	v	v	v	v		UpperCaseLetter
			v																			literal
				v																		GenericDominantUpperCaseLetter
					v																	GenericDescriptiveUpperCaseLetter
						v																GenericTransitiveUpperCaseLetter
							v															GenericUserDefinedUpperCaseLetter
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	21	**{TYPE}**
o																						Sg
	o																					ITEM
		o																				SetRole
			o																			SetName
				o																		Dominant
					o																	Descriptive
						o																Transitive
							o															UserDefined
								o														O
									o													K
										o												I
											o											P
												o										H
													o									X'
														o								M
															o							I
																o						G
																	o					S
																		o				V
																			o			C'
																				o		I'
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	1	*{REFERENCE}*
p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p		*Rational Inc Case Tool Specifications Generator*
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	1	*{Copyright ©}*
p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p		*A A Michailidis 1995*

Figure B.7. Argument Type Partition Corresponding to Figure B.1.

A	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	1	*{PARTITION}*
e	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*		*Detailed Level of Abstraction*
v	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*		*Class documentation*
																					e	*Set Cardinality*
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	21	**{CLASS NAME}**
o																						PartitionSd
	o																					Sd
		o																				SetMemberRole
			o																			SetMemberName
				o																		SMRDominant
					o																	SMRDescriptive
						o																SMRTransitive
							o															SMRUserDefined
								o														SMRIdentity
									o													SMRIdentifier
										o												SMRGeneralization
											o											SMRAggregation
												o										SMRHierarchy
													o									SMRQualifier
														o								SMRAssociation
															o							SMRFlow
																o						SMRGuard
																	o					SMRSequence
																		o				SMRValue
																			o			SMRConcurrent
																				o		SMRSequential
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	21	**{DOCUMENTATION CLASS}**
x																						PartitionSd → {Sd}
	x																					Sd → {SMR SetMemberName}
		x																				SMR → {SMRDominant | SMRDescriptive | SMRTransitive | SMRUserDefined}
			x																			SetMemberName → {literal}
				x																		SMRDominant → SMRIdentifier | SMRIdentity | SMRAggregation | SMRGeneralization | SMRHierarchy
					x																	SMRDescriptive → SMRQualifier | SMRAssociation | SMRFlow | SMRGuard | SMRSequence | SMRValue
						x																SMRTransitive → SMRSequential | SMRConcurrent
							x															SMRUserDefined → new set member roles invented by the InfoSchema/InfoMap technology users
								x														SMRIdentity → o
									x													SMRIdentifier → id
										x												SMRGeneralization → p | c
											x											SMRAggregation → w | c | s | h | m
												x										SMRHierarchy → h | l | n
													x									SMRQualifier → x
														x								SMRAssociation → v
															x							SMRFlow → u | o
																x						SMRGuard → [illegible]
																	x					SMRSequence → sd
																		x				SMRValue → id
																			x			SMRConcurrent → c
																				x		SMRSequential → [illegible]
V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	1	**{CARDINALITY CLASS}**
n	n	1	n	1	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n		value
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	3	**{ATTRIBUTE/CHARACTERISTIC}**
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Sequential
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Transient
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Public
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	1	**{OPERATION}**
v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v		Exists
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	9	**{ARGUMENTS}**
v																						SdPartition
	v																					expression
	v																					statement
	v	v						v	v	v	v	v	v	v	v	v	v	v	v	v		LowerCaseLetter
			v																			literal
				v																		GenericDominantLowerCaseLetter
					v																	GenericDescriptiveLowerCaseLetter
						v																GenericTransitiveLowerCaseLetter
							v															GenericUserDefinedLowerCaseLetter
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	2	**{CLASS TYPE}**
v	v	v	v	v	v	v	v															Abstract
								v	v	v	v	v	v	v	v	v	v	v	v	v		Concrete
A	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*1*	***{REFERENCE}***
v	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*		*Rational Inc. Case Tool Specification Generator*
A	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*1*	***{Copyright ©}***
v	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*		*A. A. Michailidis 199[illegible]*

Figure B.8. Class Documentation Partition Corresponding to Figure B.2.

A	*A*	*A*	*A*	*{PARTITION}*
v	*v*	*v*	*v*	*Detailed Level of Abstraction*
v	*v*	*v*		*Has-A Relationship*
			v	*Set Cardinality*
P	**P**	**P**	**21**	**{CLASS NAME}**
p				PartitionSd
c	p			Sd
	c	p		SetMemberRole
	c			SetMemberName
		c		SMRDominant
		c		SMRDescriptive
		c		SMRTransitive
		c		SMRUserDefined
A	*A*	*A*	*1*	*{REFERENCE}*
v	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*1*	*{Copyright © }*
v	*v*	*v*		*A. A. Michailidis 1995*

Figure B.9. Has-A Relationship Partition Corresponding to Figure B.2.

A	*A*	*A*	*A*	*A*	*A*	*A*	***{PARTITION}***
v	*v*	*v*	*v*	*v*	*v*	*v*	*Detailed Level of Abstraction*
v	*v*	*v*	*v*	*v*	*v*		*Hierarchy*
						v	*Set Cardinality*
H	**H**	**H**	**H**	**H**	**H**	**21**	**{CLASS NAME}**
h							PartitionSd
v	h						Sd
	v	h					SetMemberRole
	v						SetMemberName
		v	h				SMRDominant
		v		h			SMRDescriptive
		v			h		SMRTransitive
		v					SMRUserDefined
			v				SMRIdentity
			v				SMRIdentifier
			v				SMRGeneralization
			v				SMRAggregation
			v				SMRHierarchy
				v			SMRQualifier
				v			SMRAssociation
				v			SMRFlow
				v			SMRGuard
				v			SMRSequence
				v			SMRValue
					v		SMRConcurrent
					v		SMRSequencial
A	*A*	*A*	*A*	*A*	*A*	*1*	***{REFERENCE}***
v	*v*	*v*	*v*	*v*	*v*		*Rational Inc. Case Tool Specifications Generator*
A	*A*	*A*	*A*	*A*	*A*	*1*	***{Copyright © }***
v	*v*	*v*	*v*	*v*	*v*		*A. A. Michailidis 1995*

Figure B.10. Hierarchy Relationship Partition Corresponding to Figure B.2.

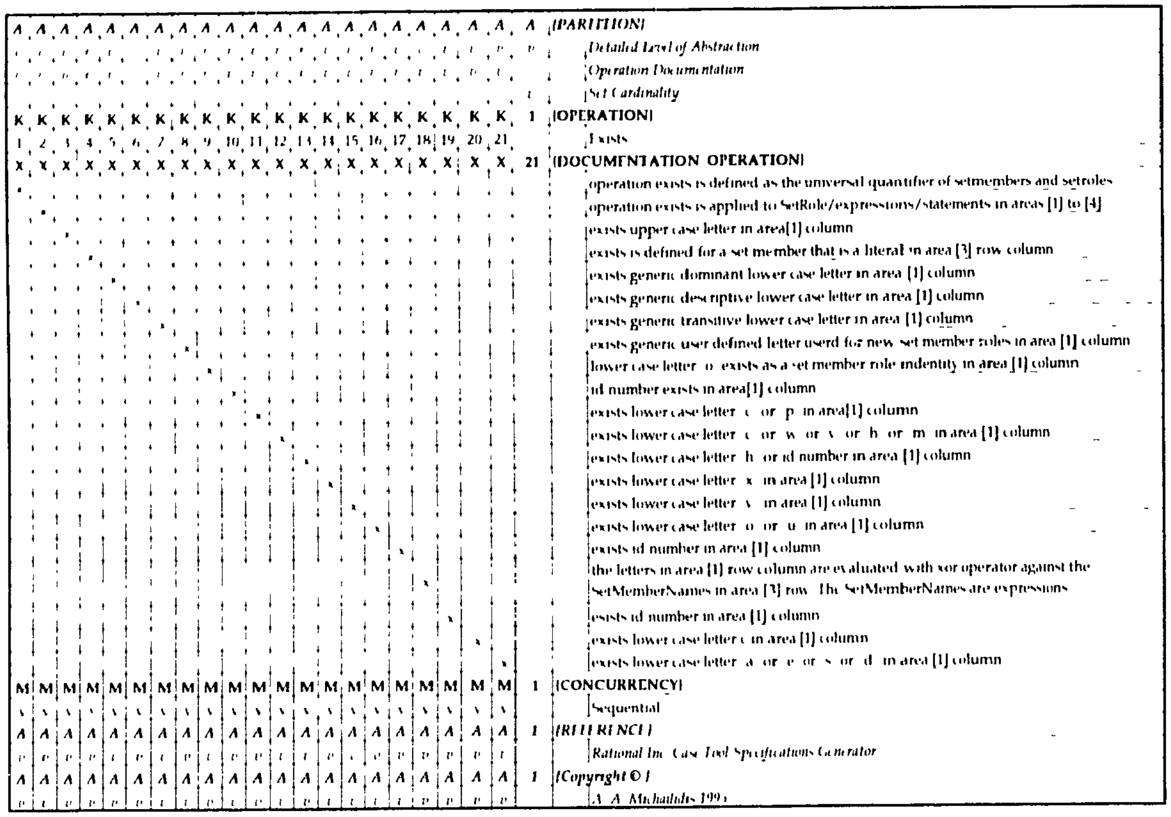

{PARTITION}
Detailed Level of Abstraction
Operation Documentation
Set Cardinality
{OPERATION}
Exists
{DOCUMENTATION OPERATION}
operation exists is defined as the universal quantifier of setmembers and setroles
operation exists is applied to SetRole/expressions/statements in areas [1] to [4]
exists upper case letter in area[1] column
exists is defined for a set member that is a literal in area [3] row column
exists generic dominant lower case letter in area [1] column
exists generic descriptive lower case letter in area [1] column
exists generic transitive lower case letter in area [1] column
exists generic user defined letter userd for new set member roles in area [1] column
lower case letter o exists as a set member role indentity in area [1] column
id number exists in area[1] column
exists lower case letter c or p in area[1] column
exists lower case letter c or w or v or h or m in area [1] column
exists lower case letter h or id number in area [1] column
exists lower case letter x in area [1] column
exists lower case letter y in area [1] column
exists lower case letter o or u in area [1] column
exists id number in area [1] column
the letters in area [1] row column are evaluated with xor operator against the SetMemberNames in area [3] row. The SetMemberNames are expressions
exists id number in area [1] column
exists lower case letter c in area [1] column
exists lower case letter a or e or s or d in area [1] column
{CONCURRENCY}
Sequential
{REFERENCE}
Rational Inc. Case Tool Specifications Generator
{Copyright ©}
A. A. Michailidis 199[illegible]
A
K
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
X
M
1
21

Figure B.11. Operation Documentation Partition Corresponding to Figure B.2.

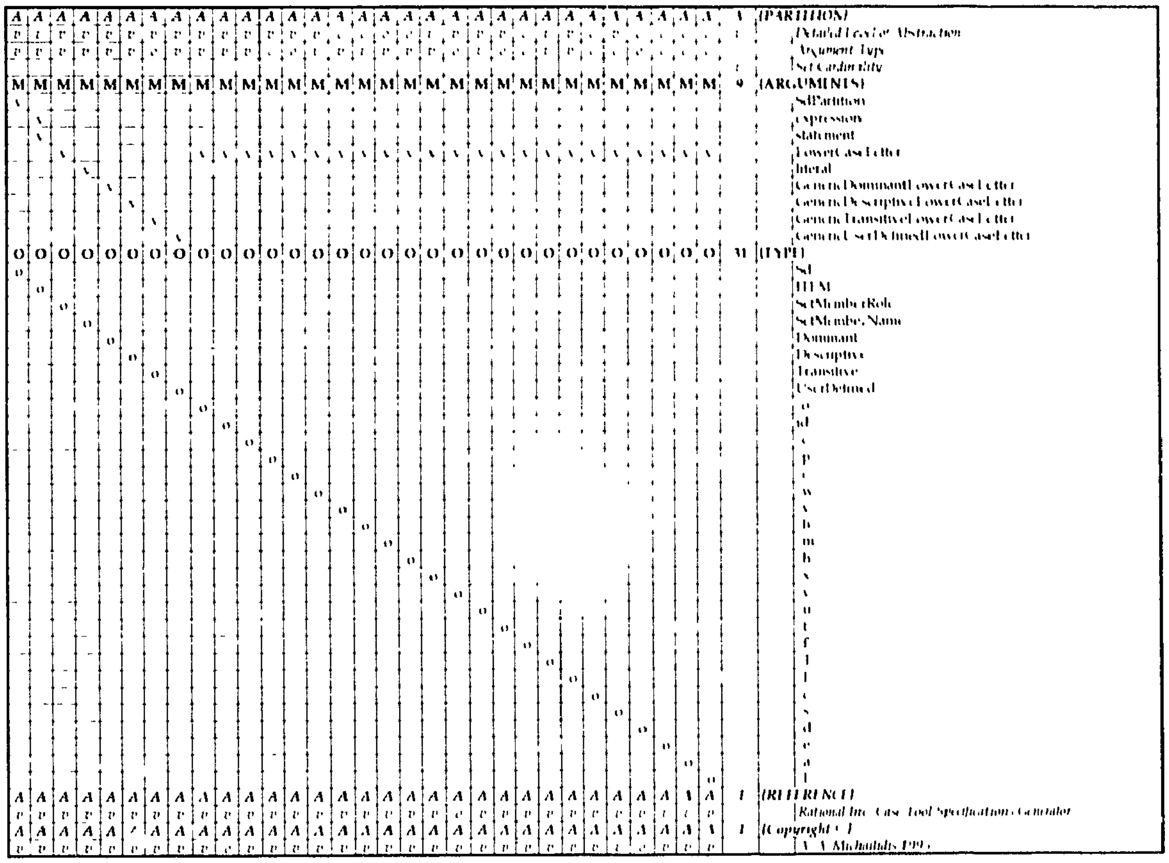


Figure B.12. Argument Type Partition Corresponding to Figure B.2.

APPENDIX C. Rational Rose/C++ CASE Tool "Exported" Results

```
(object Petal
   version          34)

(object Design "<Top Level>"
   defaults          (object defaults
          rightMargin      0.25
          leftMargin       0.25
          topMargin        0.25
          bottomMargin     0.5
          pageOverlap      0.25
          clipIconLabels   TRUE
          autoResize       FALSE
          snapToGrid       TRUE
          gridX            0
          gridY            0
          defaultFont      (object Font
             size          8
             face          "helvetica"
             bold          FALSE
             italics       FALSE
             underline     FALSE
             strike        FALSE
             color         0
             default_color TRUE))
   attributes        (list Attribute_Set
          (object Attribute
             tool          "cg"
             name          "roseId"
             value         "753117540")
          (object Attribute
             tool          "cg"
             name          "propertyId"
             value         "760817948")
          (object Attribute
             tool          "cg"
             name          "default__Project"
             value         (list Attribute_Set
                    (object Attribute
                       tool          "cg"
                       name          "FixedByValueContainer"
                       value         "")
                    (object Attribute
                       tool          "cg"
                       name          "FixedByReferenceContainer"
                       value         "")
 ..........
          (object Attribute
             tool          "cg"
             name          "default__Uses"
             value         (list Attribute_Set
                    (object Attribute
                       tool          "cg"
                       name          "ForwardReferenceOnly"
                       value         FALSE)))
```

```
        (object Attribute
            tool            "cg"
            name            "default__Subsystem"
            value           (list Attribute_Set
                (object Attribute
                    tool            "cg"
                    name            "Directory"
                    value           "AUTO GENERATE"))))
    root_category  (object Class_Category "<Top Level>"
        exportControl   "Public"
        global          TRUE
        subsystem       "<Top Level>"
        logical_models  (list unit_reference_list
            (object Class "PartitionSd"
                documentation   "PartitionSg -> {Sd}"
                fields   (list has_relationship_list
                    (object Has_Relationship
                        supplier        "Sd"))
                abstract        TRUE
                operations      (list Operations
                    (object Operation "exists"
                        documentation   "operation exists is defined as the universal quantifier for a
partition of set members and set member roles"
                        parameters      (list Parameters
                            (object Parameter "Sd"
                                type    "SdPartition"))
                        concurrency     "Sequential"
                        opExportControl "Public"
                        uid     0))
                statemachine    (object State_Machine
                    states          (list States
                        (object State "start"
                            transitions     (list transition_list
                                (object State_Transition
                                    supplier        "check for Partition")
                                (object State_Transition
                                    supplier        "ok")
                                (object State_Transition
                                    documentation "partition exists"
                                    label           "partition exists"
                                    supplier        "end"
                                    action          "return true"))
                            type            "StartState")
                        (object State "check for Partition"
                            transitions     (list transition_list
                                (object State_Transition
                                    supplier        "error")
                                (object State_Transition
                                    supplier        "OK"))
                            type            "Normal")
                        (object State "error"
                            transitions     (list transition_list
                                (object State_Transition
                                    supplier        "exit")
                (object State_Transition
                   supplier          "end"))
            type              "Normal")
         (object State "OK"
            transitions       (list transition_list
                (object State_Transition
                   supplier          "exit"))
            type              "Normal")
         (object State "exit"
            type              "EndState")
         (object State "ok"
            transitions       (list transition_list
                (object State_Transition
                   documentation "end "
                   label             "goto end"
                   supplier          "end"
                   action            "return true")
                (object State_Transition
                   supplier          "end"
                   action            "return true"))
            type              "Normal")
         (object State "end"
            type              "EndState")))
statediagram     (object State_Diagram ""
   title          ""
   zoom           100
   max_height     28350
   max_width      21600
   origin_x       0
   origin_y       0
   items          (list diagram_item_list
         (object StateView "start" @1
            location       (267, 236)
            font           (object Font
                size       12
                face       "helvetica"
                bold       FALSE
                italics    FALSE
                underline        FALSE
                strike     FALSE
                color      0
                default_color     TRUE)
            label          (object ItemLabel
                location          (267, 236)
                anchor_loc        1
                nlines            1
                max_width         480
                justify    0
                label      "start")
            size           240)
         (object StateView "end" @2
            location       (945, 223)
            font           (object Font
                size       12
```

```
                        face     "helvetica"
                        bold     FALSE
                        italics  FALSE
                        underline        FALSE
                        strike   FALSE
                        color    0
                        default_color    TRUE)
                    label            (object ItemLabel
                        location         (945, 223)
                        anchor_loc       1
                        nlines           1
                        max_width        480
                        justify  0
                        label    "end")
                    size             240))
                (object TransView "partition exists"
                    label            (object SegLabel
                        location         (606, 186)
                        anchor_loc       1
                        nlines           1
                        max_width        450
                        justify  0
                        label    "partition exists"
                        pctDist          0.5
                        height           45
                        orientation      0)
                    client           @1
                    supplier         @2
                    x_offset         FALSE))))
        (object Class "Sd"
            documentation    "Sd -> {{SetMemberRole SetMember}}"
            fields   (list has_relationship_list
                (object Has_Relationship
                    supplier             "SetMemberName")
                (object Has_Relationship
                    supplier             "SetMemberRole"))
            superclasses         (list inheritance_relationship_list
                (object Inheritance_Relationship
                    supplier             "PartitionSd"))
            abstract             TRUE
            operations           (list Operations
                (object Operation "exists"
                    documentation        "operation exists is applied to SetMember / Role / expressions
/ statements in areas [1] to [4]"
                    parameters           (list Parameters
                        (object Parameter "ITEM"
                            type     "expression")
                        (object Parameter "ITEM"
                            type     "statement")
                        (object Parameter "SetMemberRole"
                            type     "LowerCaseLetter"))
                    concurrency          "Sequential"
                    opExportControl  "Public"
                    uid      0))
```

```
statemachine       (object State_Machine
   states           (list States
      (object State "start"
         transitions       (list transition_list
               (object State_Transition
                  supplier          "check for SetRole")
               (object State_Transition
                  documentation "SetRole exists in area [2]"
                  label             "SetRole exists in area [2] "
                  supplier          "exit"
                  action            "return true")
               (object State_Transition
                  documentation "SetName exists in area [2]"
                  label             "SetName exists in area [2]"
                  supplier          "exit"
                  action            "return true")
               (object State_Transition
                  documentation "set role or set name exist in Sg"
                  label             "SetRole or SetName exist"
                  supplier          "exit"
                  action            "return true"))
         type              "StartState")
      (object State "check for SetRole"
         transitions       (list transition_list
               (object State_Transition
                  supplier          "error")
               (object State_Transition
                  supplier          "OK")
               (object State_Transition
                  supplier          "check for SetName"))
         type              "Normal")
      (object State "error"
         transitions       (list transition_list
               (object State_Transition
                  supplier          "exit"))
         type              "Normal")
      (object State "OK"
         transitions       (list transition _list
               (object State_Transition
                  supplier          "exit"))
         type              "Normal")
      (object State "exit"
         type              "EndState")
      (object State "check for SetName"
         transitions       (list transition_list
               (object State_Transition
                  supplier          "error")
               (object State_Transition
                  supplier          "OK")
               (object State_Transition
                  supplier          "error"))
         type              "Normal")))
statediagram       (object State_Diagram ""
   title            ""
```

```
zoom            100
max_height      28350
max_width       21600
origin_x        0
origin_y        0
items           (list diagram_item_list
        (object StateView "start" @3
          location          (198, 263)
          font              (object Font
                size    12
                face    "helvetica"
                bold    FALSE
                italics FALSE
                underline       FALSE
                strike  FALSE
                color   0
                default_color   TRUE)
          label             (object ItemLabel
                location        (198, 263)
                anchor_loc      1
                nlines          1
                max_width       480
                justify 0
                label   "start")
          size              240)
        (object StateView "exit" @4
          location          (1111, 246)
          font              (object Font
                size    12
                face    "helvetica"
                bold    FALSE
                italics FALSE
                underline       FALSE
                strike  FALSE
                color   0
                default_color   TRUE)
          label             (object ItemLabel
                location        (1111, 246)
                anchor_loc      1
                nlines          1
                max_width       480
                justify 0
                label   "exit")
          size              240)
        (object TransView "SetRole or SetName exist"
          label             (object SegLabel
                location        (654, 211)
                anchor_loc      1
                nlines          1
                max_width       450
                justify 0
                label   "SetRole or SetName exist"
                pctDist         0.5
                height          45
```

```
                                    orientation      0)
                              client           @3
                              supplier         @4
                              x_offset         FALSE))))
          (object Class "SetMemberRole"
               documentation
|SMRole -> {SMRDominant | SMRDescriptive | SMRTransitive | SMRUserDefined}

|

|

               fields     (list has_relationship_list
                 (object Has_Relationship
                      supplier           "SMRUserDefined")
                 (object Has_Relationship
                      supplier           "SMRDominant")
                 (object Has_Relationship
                      supplier           "SMRTransitive")
                 (object Has_Relationship
                      supplier           "SMRDescriptive"))
               superclasses       (list inheritance_relationship_list
                 (object Inheritance_Relationship
                      supplier           "Sd"))
               abstract           TRUE
               cardinality        (value Cardinality "1")
               operations         (list Operations
                 (object Operation "exists"
                      documentation     "exists upper case letter in area [1].column"
                      parameters        (list Parameters
                         (object Parameter "SetMemberRole"
                              type       "LowerCaseLetter"))
                      concurrency       "Sequential"
                      opExportControl "Public"
                      uid       0))
               statemachine       (object State_Machine
                 states           (list States
                      (object State "start"
                         transitions      (list transition_list
                              (object State_Transition
                                 supplier         "check for SetRole")
                              (object State_Transition
                                 documentation "set role exists in area [1]"
                                 label            "SetRole exists in arc [1]"
                                 supplier         "exit"
                                 action
|return true

|
                                 ))
                         type            "StartState")
                      (object State "check for SetRole"
                         transitions      (list transition_list
                              (object State_Transition
                  supplier          "error")
                (object State_Transition
                  supplier          "OK"))
            type              "Normal")
          (object State "error"
            transitions       (list transition_list
                (object State_Transition
                  supplier          "exit"))
            type              "Normal")
          (object State "OK"
            transitions       (list transition_list
                (object State_Transition
                  supplier          "exit"))
            type              "Normal")
          (object State "exit"
            type              "EndState")))
    statediagram      (object State_Diagram ""
      title           ""
      zoom            100
      max_height      28350
      max_width       21600
      origin_x        0
      origin_y        0
      items           (list diagram_item_list
          (object StateView "start" @5
            location          (206, 215)
            font              (object Font
                size      12
                face      "helvetica"
                bold      FALSE
                italics   FALSE
                underline         FALSE
                strike    FALSE
                color     0
                default_color     TRUE)
            label             (object ItemLabel
                location          (206, 215)
                anchor_loc        1
                nlines            1
                max_width         480
                justify   0
                label     "start")
            size              240)
          (object StateView "exit" @6
            location          (981, 221)
            font              (object Font
                size      12
                face      "helvetica"
                bold      FALSE
                italics   FALSE
                underline         FALSE
                strike    FALSE
                color     0
                default_color     TRUE)
```

```
                        label           (object ItemLabel
                            location            (981, 221)
                            anchor_loc          1
                            nlines              1
                            max_width           480
                            justify   0
                            label     "exit")
                        size            240)
                      (object TransView "SetRole exists in arc [1]"
                        label           (object SegLabel
                            location            (594, 174)
                            anchor_loc          1
                            nlines              1
                            max_width           450
                            justify   0
                            label     "SetRole exists in arc [1]"
                            pctDist             0.5
                            height              45
                            orientation         0)
                        client          @5
                        supplier        @6
                        x_offset        FALSE))))
          (object Class "SetMemberName"
               documentation
|SetMemberName -> "{"literal"}"

               superclasses       (list inheritance_relationship_list
                  (object Inheritance_Relationship
                     supplier             "Sd"))
               operations         (list Operations
                  (object Operation "exists"
                     documentation    "operation exists is defined for a SetMember that is a literal in
area [3].row.column"
                     parameters        (list Parameters
                        (object Parameter "SetMemberName"
                           type      "literal"))
                     concurrency       "Sequential"
                     opExportControl  "Public"
                     uid       0))
               statemachine       (object State_Machine
                  states          (list States
                     (object State "start"
                        transitions       (list transition_list
                           (object State_Transition
                              supplier          "check for literal")
                           (object State_Transition
                              documentation
|Set Name exists in area [4.row.column]

|

                              label            "SetName exists in arc [4.row.column]"
                              supplier         "exit"
                              action           "return true"))
```

```
                type            "StartState")
            (object State "check for literal"
                transitions     (list transition_list
                    (object State_Transition
                        supplier        "error")
                    (object State_Transition
                        supplier        "OK"))
                type            "Normal")
            (object State "error"
                transitions     (list transition_list
                    (object State_Transition
                        supplier        "exit"))
                type            "Normal")
            (object State "OK"
                transitions     (list transition_list
                    (object State_Transition
                        supplier        "exit"))
                type            "Normal")
            (object State "exit"
                type            "EndState")))
    statediagram    (object State_Diagram ""
        title           ""
        zoom            100
        max_height      28350
        max_width       21600
        origin_x        0
        origin_y        0
        items           (list diagram_item_list
            (object StateView "start" @7
                location        (158, 238)
                font            (object Font
                    size        12
                    face        "helvetica"
                    bold        FALSE
                    italics     FALSE
                    underline           FALSE
                    strike      FALSE
                    color       0
                    default_color       TRUE)
                label           (object ItemLabel
                    location            (158, 238)
                    anchor_loc          1
                    nlines              1
                    max_width           480
                    justify     0
                    label       "start")
                size            240)
            (object StateView "exit" @8
                location        (1001, 218)
                font            (object Font
                    size        12
                    face        "helvetica"
                    bold        FALSE
                    italics     FALSE
```

```
		underline	FALSE
		strike	FALSE
		color	0
		default_color	TRUE)
	label		(object ItemLabel
		location	(1001, 218)
		anchor_loc	1
		nlines		1
		max_width	480
		justify	0
		label	"exit")
	size		240)
(object TransView "SetName exists in arc [4.row.column]"
```