Prototyping with Objects
D. Coleman, P. Arnold, S. Bodoft, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes
Object-Oriented Development: The Fusion Method

S. Cook and J. Daniels
Designing Object Systems

B. Henderson-Sellers
A Book of Object-Oriented Knowledge

B. Henderson-Sellers and J. Edwards
Book Two Object-Oriented Knowledge: The Working Object

H. Kilov and J. Ross
Information Modelling: An Object-Oriented Approach

P. Krief
Prototyping with Objects

K. Lano and H. Haughton
Object-Oriented Specification Case Studies

J. Lindskov Knudsen, M. Lofgren, O. Lehrmann Madsen and B. Magnusson
Object-Oriented Environments: The Mjoelner Approach

M. Lorenz
Object-Oriented Software Development: A Practical Guide

M. Lorenz and J. Kidd
Object-Oriented Software Metrics

B. Meyer
An Object-Oriented Environment: Principles and Applications

B. Meyer
Eiffel: The Language

B. Meyer
Reusable Software: The Base Object-Oriented Component Libraries

B. Meyer

B. Meyer and J.-M. Nerson (eds)
Object-Oriented Applications

D. Mandrioli and B. Meyer (eds)
Advances in Object-Oriented Software Engineering

O. Nierstrasz and D. Tsichritzis
Object-Oriented Software Composition

G. Pomberger and P. Blaschek
An Object-Oriented Approach to Software Engineering

R. Rist and R. Terwilliger
Object-Oriented Programming in Eiffel

P. J. Robinson
Hierarchical Object-Oriented Design

R. Switzer
Eiffel: An Introduction

K. Walden and J.-M. Nerson
Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems

R. Wiener
Software Development Using Eiffel: There Can Be Life Other Than C++

(Various editors)
Proceedings of TOOLS Conferences, 4 to 16: Technology of Object-Oriented Languages and Systems
Prototyping with Objects

Philippe Krief

Translated by
John Plaice

Département d’Informatique
Université Laval
Ste Foy (Quebec)
Canada

Prentice Hall
London New York Toronto Sydney Tokyo Singapore
Madrid Mexico City Munich
Contents

Foreword vii
Preface x

I Prototyping and object-oriented languages 1
1 Modeling and prototyping concepts 3
1.1 Mathematical models 4
1.2 Figurative models 5
1.3 From model to prototype 7

2 Prototypes and software engineering 9
2.1 Software model 10
2.2 Software prototype 10
2.3 Rapid prototyping 15

3 Object-oriented languages 21
3.1 Objects, messages and encapsulation 21
3.2 Classes, instances and inheritance 25

4 Prototyping a calculator 32
4.1 Conceptualization 33
4.2 Testing the prototype 38
4.3 Dependency mechanism 40
4.4 Model–View–Controller (MVC) systems 43
4.5 Graphical interface for the calculator 53
4.6 Creating an infix calculator 59
5 Solving problems through prototyping  63
   5.1 Psychology and programming  63
   5.2 Problem solving  66

6 State-of-the-art programming environments  75
   6.1 The DesignNet model  75
   6.2 An interface generator  81
   6.3 The ECCAO project  86

II The Model–Point of View–Controller approach  93

7 The Model–Point of View–Controller trilogy  95
   7.1 Multiple representations  95
   7.2 Multiple interpretations  96
   7.3 Factorization of the traversal  98

8 The kernel of a Model–Point of View–Controller system  100
   8.1 Defining the model  100
   8.2 Defining the controller  115
   8.3 Defining a point of view  137

9 Properties and extensions of the MPVC environment  160
   9.1 Multiple points of view  160
   9.2 Multiple controllers  165
   9.3 Parallelism and MPVC  173
   9.4 Reflexivity and MPVC  182
   9.5 An MPVC browser  201

10 Conclusion  207
    10.1 Summary  207
    10.2 Extensions to the MPVC environment  208
    10.3 Overview  210
    10.4 To conclude  214

Bibliography  215

Index  225
Foreword

What a book! Its advanced programming examples allowed me to improve my Smalltalk programming techniques. Its presentation of object-programming theory improved my understanding of how to create, use, combine and modify objects. Finally, its suggested prototyping environment was so seductive that I am now using it myself.

The author brilliantly succeeds in this tour de force, speaking to the novice Smalltalk programmer, the object-oriented programming theoretician and the software engineering researcher. He does this through his pedagogical skills, which clearly show the reasoning necessary for developing programs, as well as by placing his work in the larger framework of cognitive psychology and artificial intelligence. All this is done in a pleasant manner where difficult concepts are made clear.

Philippe Krief makes an important contribution to a debate that is at least 20 years old. How does one program? Does one program from hindsight, starting with a plan, resulting from a formal specification, where programming is then a step-by-step simulation of a compiler for the specification? Or does one program by creating structure, a program's implementation being the result of interactive and exploratory visualizations and variations of different points of view?

Of course, the first assumption can be used, but it supposes that the programmer has an a priori perfect understanding of all of the characteristics of the future program, that this understanding can be translated into a verifiable formalism and that the specification can be translated into a programming language. Surely it is these unresolved problems themselves that prevent a compiler from being developed for these specifications? Even worse, how does one specify a program that is to explore new programming domains or new automation of knowledge? Isn't programming precisely the formalization and the modeling of a previously incompletely understood activity? How should one, for example, formally specify a program for natural language understanding or knowledge representation?

The author answers these questions by clearly choosing the second assumption. Programming is an experimental, non-linear activity of model creation and of
changing points of view. He offers us a set of software tools for this exploration. One of the fundamental arguments of this work is the refutation of linear concepts of modeling stages: a model anticipates what is to be, and it is through experiments effected with prototyping tools that it becomes clear what can be modeled, just as is done in the other experimental sciences, through the visualizing activity of the designer and the experimenter.

The author presents his designs in a convincing manner: a series of ready-to-use Smalltalk classes, which are actually used throughout the book to implement Minsky's neural nets. The latter are used to present prototype modeling's capacities to describe sequential and parallel traversal, different kinds of synchronization, and multiple-controller and multiple-points of view.

The basis for this book is the idea that computable processes are better understood if they are separated into three parts: the object manipulated by the process (the author's 'model'), the traversal of this object (the 'controller') and the actions that must be effected at each step of the traversal (the 'points of view'). It is a powerful generalization of the standard data-control dichotomy. What is new is the separation between the static aspect (the model) and the dynamic aspect (the traversal).

It should be clear that this Model–Point of View–Controller (MPVC) trilogy is derived from the Model–View–Controller (MVC) trilogy that is found in any Smalltalk graphical interface. But the MPVC partition allows further generalization; the controller becomes a natural object, which understands its model and the point of view that it must use while traversing. This traversal mode is given by an ordering graph whose nodes are rules that define the point of view's activation at a given point in the model, along with how to get to other points.

Noting that the ordering graph is itself an object that is traversed by effecting actions at each step, an MPVC description becomes appropriate. This is exactly what the author has done, by adding a 'meta'-level to his system that can itself be naturally described in MPVC. These 'meta'-levels allow the basic model to be enriched, through the use of sequential or parallel traversals, step-by-step or tracing, and of multiple-points of view. The author provides a number of examples of these 'meta'-levels, along with their fractal or self-similar character: an expression is computed, while its traversal is visualized graphically, as is the ordering graph directing the traversal.

The author has fulfilled an old dream of all programmers, by building a system that incorporates, naturally, reflexivity and treatment of meta-knowledge. Doing so allows him to reuse, without rewriting, the software components of his system in any future prototype development.

I have no doubts that reading this book will raise new, important research questions. Here are mine. For what kind of program is this decomposition adequate? Does a program simulating the cooperation between different actors conform to the MPVC trilogy? What does distributed computation mean in MPVC? To what extent can the translation of an MPVC prototype into a working program be automated? In any case, readers will find in this book both a model to follow and a
tool to use. Finally, readers will regret not having the MPVC system available in order to experiment with it and to use it in their own development work.

Harald Wertz
Professor, Université Paris VIII
Preface

Software prototyping

This book presents the concepts of software prototype and software prototyping. By making a client, a designer and a well-suited development environment interact, prototyping significantly contributes to the development of specifications and to software design.

Prototyping then becomes part of the software life cycle. Its use in software engineering dates back to the 1980s, just as object-oriented languages became popular. It is normally used when a client is unable to completely define the requirements using traditional tools such as SADT and MERISE; this is the case for most software.

The client and the designer then need to 'see', experiment, test and validate the different possible solutions to the problems, in order to gain a clear and coherent idea of what they actually want and how exactly to do it. Prototyping, also known as experimental programming, is therefore an excellent paradigm for extracting precise specifications that completely describe the problem and the proposed solutions.

Prototyping is not simply a specification of the 'proper' external use of the software; it is also used to specify, test and validate the coherence and consistency of the design and the prototyped solutions. Also, because of its high level of design, the resulting prototypes can be used as precursors to the final written system.

For prototyping to be realistic, a prototyping environment is needed for development. It should include methods, techniques, formalisms and tools to ease the act of acquiring information about the relevance and adequacy of the software's specification and design.

The design of prototyping environments combines two important themes in software engineering research; the book is therefore divided into two parts. First is the study of formalisms to represent application domains; object-oriented languages are shown to be an adequate basis for this kind of representation. Second is the
design of tools to interpret, manipulate and validate specifications expressed in
different formalisms; a design method that eases the rapid implementation of such
tools is described.

Overview

The epistemological, scientific and industrial contexts for prototyping are presented
in Part I. The concepts of model and prototype are formalized and the rôle of pro-
totyping in the software life cycle is made clear. Parallels are drawn with the
concepts and methods in object-oriented programming, through the implementa-
tion of a software prototype for a calculator in Smalltalk-80. Finally, a brief outline
of the cognitive framework for prototyping is continued with a presentation of the
state of the art for large families of prototyping environments.

Part II illustrates the design and implementation of an interactive system for pro-
totyping environment developments, using a three-pronged approach to software
development. The Model–Point of View–Controller (MPVC) approach is presented
as an object design method that simplifies the implementation of tools for inter-
preting, manipulating, observing and animation of representation models.

The mechanical aspects of the program, i.e. the data traversal instructions that
do not actually interpret, that do not carry evaluatory or semantic components, are
separated from the rest of the program. The interactive design of multiple traversal
methods is encouraged, as are multiple interpretations and multiple representations
of a given structure.

As an example, we present an MPVC system for multiple interpretations of Min-
sky’s neural nets. This implementation shows that this approach can be seen as
an experimental prototyping method, since the search mechanism is clearly sepa-
rated from the representation model and from the different possible interpreta-
tions. Hence, the writing of representation model interpreters is simplified and
systematized. The resulting MPVC environment can be seen as a design tool for
interpreters of multiple representation models and as a protototype development
environment tool.

Acknowledgements

The suggestions, criticisms and comments of many people made this book much
better. They all deserve my thanks.

I would first like to thank Harald Wertz, professor at Université Paris VIII, for
having agreed to write the foreword. His supervisory rôle during my thesis, his
numerous relevant remarks and comments, his enthusiasm for research, which I
inherited, his unlimited availability, his complete confidence and his real friendship
all played a major rôle in the writing of this book.
I would also like to thank Patrick Greussay (professor at Université Paris VIII), Jean-François Perrot (professor at Université Paris VI), Jean Bézivin (professor at Université de Nantes), Jean-Pierre Briot and Christophe Dony (researchers at the CNRS and at Rank-Xerox) and Henry Lieberman (researcher at MIT-Media Lab) for their careful reading, their relevant observations and for the interest that they showed for my work.

I would also like to thank all those who contributed to this work, from research teams at:

- Computer Science Department (Université Paris VIII): Patrick Greussay, Harald Wertz, Jacqueline Signorini, Elen Sparer, Vincent Lesbros, Jean-Claude Nkou, Daniel Goossens, Jean Méhat, Marie-Solange Touzeau, Jean-Christophe Petithory, Marie-Noëlle Rozin;
- Thomson-CSF/DSE (Bagneux) and Computer Science Laboratory (Paris-Nord): Yo Keller, Robert Bourgeois, Daniel Kayser, Renaud Tricot, Catherine Moulinoux, Christine Desclaux and Gilles Demailly;
- Ranx-Xerox-LITP Laboratory: Pierre Cointe, Jean-Pierre Briot, Christophe Dony and Henry Lieberman;
- LAPORIA: Jean-François Perrot, Gil Blain, Robert Voyer and Jacques Ferber;
- ACKIA company (TRIEL-DDP group), where I was able to pursue my research while applying it in an industrial setting: Christian Dujardin, Gil Blain and Robert Voyer.

Finally, and most importantly, I want to thank all those around me, friends and family, for their confidence, their understanding, their encouragement and their patience.

à Florence...
Part I

Prototyping and object-oriented languages

We normally consider that knowledge is in itself a positive thing, but it is in fact only useful when we can use it to help us attain our objectives [132].

The first part of this book presents the main contributions of object-oriented languages to prototyping. The concepts of model and prototype are first considered from an epistemological point of view. These two concepts are then placed in a more concrete setting, software engineering. The key aspects of object-oriented languages are then presented by referring to concepts used in prototyping; as an example, a calculator is prototyped. An explanation of the cognitive framework for prototyping is followed by a description of the state of the art in prototyping environments.
Chapter 1

Modeling and prototyping concepts

Definition 1. Model: (1) What serves or should serve as an object to be imitated while making or reproducing something. (2) Arts: Person or object of whom an artist reproduces an image. (3) Person, fact or object containing basic qualities or characteristics that make of it a representative of a category. (4) A particular object from which similar objects are produced, in many copies. (5) An object of the same kind as a larger object, but made in miniature form. (6) Simplified representation of a process, of a system. [154]

Definition 2. Prototype: (1) Education: First (main, original) model. (2) Normal usage: First implementation of a model (of a mechanism, of a vehicle) built using mass production. [154]

It is impossible to define prototyping without referring to model: the two are closely linked. By definition, a prototype is an original model that precedes the final model. As a model, it represents, imitates and simplifies a ‘perceived world’. It can have a descriptive, explanatory or demonstrative rôle.

In the discussion below, the term reality refers to a certain perceived reality and not to the characteristic of what actually exists. In so doing, we accept the idealist arguments of Watzlawick: ‘Of all illusions, the most perilous consists of thinking that there is only one reality. In fact, what exists are different versions of reality, some of which can be contradictory, and which are all the result of communication and not the reflection of objective and eternal truths’ [185].

To better understand each of these aspects, different forms of model and prototype, depending of the applicable field of study, are examined below.

The concept of model has been much used in all scientific disciplines. It is used in the so-called exact sciences, such as mathematics, physics, chemistry or astronomy, and even in the (formerly known as) experimental sciences, such as biology, botany or meteorology, as well as in the social sciences such as psychology, linguistics, economics and sociology. Each of these disciplines has its own accepted notion
of model and its own *modeling tools*, which include any physical, electronic, mathematical or computer tools that allow the building and manipulation of models. However, each of these models can be regrouped into two large families:

- *Mathematical* or *theoretical* models, which manipulate mathematical objects such as equations, systems of equations, or probabilistic or statistical laws, are generally used to abstract and generalize a theoretical or empirical process.
- *Figurative* models manipulate concrete, tangible representations or objects from the field of study (e.g., a miniature, a drawing or an electrical circuit). These models are generally used to give interpretative and demonstrative support, for example in the statement of a scientific theory.

This double perspective of modeling is represented in Figure 1.1. A model can be perceived as the focal point between the theoretical field that it interprets and the empirical field that it synthesizes.

![Figure 1.1 On the use of models](image)

1.1 Mathematical models

Abstraction, necessary for the development of mathematical models, allows the use of *analogies* between situations found in different fields [144, 151]. Their use in exact sciences such as physics, chemistry or astronomy allows a complementary formalism that helps validate the development of a theory.

For example, in cosmology, there are a large number of mathematical models to formalize theories about the origins of matter, galaxy formation (*big bang* model [133], *chaotic* model) or the representation of the universe (*elliptic*, *Euclidian* or *hyperbolic* model [5]).

These different models refer to the same problems, although they may confirm diametrically different theories. In fact, it is not uncommon to find abstract or figurative models that validate different perceptions of the same reality.

Even though they are primarily used in the exact sciences, mathematical models are used more and more in disciplines based on observation, such as biology, ecology, meteorology, sociology and demography, in order to validate empirical approaches used in these domains.

Therefore, mathematical models are used in biology or ecology to study the effective increase of a population over time, as in the exponential model, or to represent the evolution of two mutually dependent populations by modeling symbiosis,
competition and predator–prey relationships, as in the coupling model [55, 113]. These models allow the representation of a biological system through the use of equations.

Mathematical models are also used in social sciences, such as psychology, economics, sociology and civil planning. Through the extrapolation of statistics, they offer approximations, in order to anticipate socio-economic phenomena and human behaviors.

Such models allow the elaboration of predictive theories, for example the demographic and geographic evolution of a population [126].

Note that computer science has significantly contributed to the design and growth of computational models, which use numerical computations. The high degree of plasticity of computer models, in which many different parameters may be freely manipulated, simplifies their creation and their refinement, by ensuring consistency as well as efficiency and utility.

1.2 Figurative models

Figurative models can be separated into three large groups: objective references, simplifiers or imitators, and analogies.

1.2.1 Objective references

These models describe and explain phenomena observed solely for demonstrative purposes. Building this kind of model consists mainly in adjusting parameters so that the model best corresponds to the observations made in the studied domain.

This kind of model can be found, for example, in biology. The structure of a model is a geometric construction that explains the physical arrangement of atoms or a group of atoms in a molecule. The inter-atomic links and the angles between them can in fact be computed for simple molecules. The model is built up from experimental data obtained through various physico-chemical processes. The model is verified through observation, using the same processes. It is then modified until it takes into account ‘as accurately as possible’ the results obtained from observation.

Some aspects of simulation can also be considered to be objective references. In fact, many simulation models attempt to be an objective view of a particular perceived reality.

This kind of model can be found in aeronautics, with flight simulators, or in architecture, with models of hydro-electric dams. These models attempt to precisely reproduce certain aspects of reality. Those using these models must have the impression and the sensation of examining and manipulating the modeled reality.
1.2.2 Simplifiers or imitators

These are the most common models; they are often used in software engineering. The model is perceived as a canvas that temporarily replaces the – too large – complexity of the problem under study.

Defining such models consists of building a hypothetical simplifying scheme, which must be adjusted according to the known empirical or theoretical data.

Generally, these models are built to simplify speculative reasoning about particular characteristic properties of the reality they are imitating. This reality does not in fact exist if the context is software prototyping.

In biology, artificial membranes are built using elementary components of the membrane. Such implementations are imitator models; they can be compared to natural membranes, by subjecting them to the same experiments. Similar experiments are used in medicine to test artificial hearts [120].

In botany, plant growth is studied by manipulating the parameters of plant growth simulation software [57, 153]. This kind of simulation allows the ‘acceleration of time’, thereby allowing a landscaper to visualize a garden, or a sylviculture expert a forest, five or ten years after planting.

An architect creates different simplifying models of a building under construction. First, miniature models are built to show the external form of the building (e.g., it is made of glass or of wood). Mood sketches – also known as misleading graphical documents – are drawn to put forward certain aspects of the project, such as the human and urban environment and the surrounding vegetation. These models are followed by volume sketches that show how the floorspace and the volume of the building are organized. Finally, the models are made more precise: façades are drawn, as are more detailed plans.

As for software prototyping, the different phases that the architect used during the construction of a building are anticipatory models: they offer a simplified representation of the reality to come.

1.2.3 Analogies

Sometimes, the behavior of a phenomenon is unclear. Analogy is then used, in an attempt to create a perceived image of reality based on a known and understood phenomenon.

For example, comparing the heart and a pump allowed Harvey to show that blood circulation followed hydraulic laws. Similarly, Lavoisier, by making an analogy between breathing and combustion, showed that a physiological function could be analyzed using physical and chemical concepts.
1.3 From model to prototype

Figure 1.2 shows how modeling is undertaken. Modeling consists of defining a language and a formalism that allow the conceptualization, i.e. the creation of a general and abstract mental representation, of the objects in the field of study and the relationships between them; the manipulation, using formal rules, of these objects in an area other than the field of study; and the interpretation of the experimental results within the framework of the area under study.

![Figure 1.2 Modeling](image)

Figure 1.3 shows how a model is built and incrementally improved [183]. It contains four phases, with an arbitrary starting point.

![Figure 1.3 Building and improving a model](image)

The deductive phase consists of deriving, from an existing theoretical model, an empirical model containing observable variables. The latter can be used to test the assumptions and conclusions of the theoretical model.

The predictive or interpreting phase consists of devising, from the theoretical model, experiments that can test it. Prediction can mean asserting that an event will occur (a real prediction) or asserting that an event has occurred.

The descriptive phase consists of integrating, into a new or existing empirical model, empirical observations, thereby allowing the assumptions that were made about the parameters in the hypothetical model to be validated or computed empirically.
The inductive phase consists of evaluating the differences between the hypothetical model and the validated model. This analysis determines, taking into account all empirical models, either the modifications that must be made to the existing theoretical model or the structure of a new theoretical model.

From these specifications, the concepts of model and prototype can be defined.

**Definition 3.** A model \( M \) is a pair \((D, T)\), where \( D \) is a symbolic or figurative representation of a particular domain and \( T \) is a set of tools that allow the manipulation and exploration of \( D \).

**Definition 4.** A prototype \( P \) is an experimental model, i.e. a model of model \( M \). Therefore \( P = (D_P, T_P) \), where \( D_P \) is a representation of \( M \) and \( T_P \) is a set of specialized tools for the manipulation and exploration of \( D_P \).

Consider an example in biology (see Figure 1.4). The experimenter observes a phenomenon and wishes to model it. Reality is the observed phenomenon. To model it, the experimenter builds partial and simplified models to discover the different mechanisms that will have to be included in the final model for it to properly imitate reality. These different partial models are the prototypes. The experimenter will then have two different kinds of tools: tools to manipulate and validate the different prototypes \((T_P)\) and tools to manipulate, experiment with and explore the observed reality, through the final model \((T)\).

![Diagram of reality, model, and prototype](image)

**Figure 1.4** Prototyping through abstractions

In the case of an automobile, the milestones are different, but the concepts are the same. A mechanical engineer designs a new car: this is the view of ‘reality to come’. Generally, the esthetics and the mechanics of the imagined car are perfect. Then come the first plans and miniature models. Already, the engineer’s view of the car has changed. Then the first life-size model, the prototype, is built.

Different tests, such as gas consumption or reliability, are effected on the prototype to validate and improve its characteristics. Economic and political requirements will force the final version of the prototype to be modified yet again so that it can be sold commercially. This last version is the **standard model** of the vehicle.

To conclude, there exist at least two different levels of abstraction with respect to a perceived reality: the model allows the manipulation of the representation of some real phenomenon; the prototype allows this representation of reality to be built, using the corresponding tools.
Prototypes and software engineering

Software engineering [26] is the branch of computer science that attempts to master the complexity of developing software. It includes several areas, including programming and prototyping environments, user interfaces, specification and design methodologies, and project productivity and management. Software engineering tools are used throughout the life cycle, which includes the design, implementation and maintenance phases of a piece of software.

Two approaches to software can be distinguished. The user perceives software as static, at least temporarily, offering a set of functions and attributes to the user. The developer sees software as something that is always being changed, i.e. is in an experimental state. Nevertheless, the developer is in fact also a user of the software engineering tools.

These two approaches (see Figure 2.1) correspond to the distinction drawn in the previous chapter between model and prototype. For the user, a piece of software is a model; for the developer, it is a prototype. During the development phase, some privileged or expert users might have access to the prototype.

![Diagram of two approaches to software]

**Figure 2.1 Two approaches to software**
2.1 Software model

For a user, a piece of software is a conceptual and functional formalism that is well suited to the problem at hand. For example, a program that plays chess should have a graphical representation for each piece (king, queen, rook, bishop, knight and pawn). Furthermore, it must allow each player to select a piece of the right color to be played, according to the rules of the game. The program might also be able to suggest moves to the player, depending on the status of the game.

As a model, a chess-playing program gives a graphical representation of the pieces and board, as well as a set of functions allowing this representation to be manipulated according to a definite set of formal rules, in this case to play chess. A chess-playing program models a game of chess.

Hence, the definition for model can be applied to software, yielding the following definition.

**Definition 5.** A software model is a pair \((D, T)\), where \(D\) is the computer representation of the concepts in the domain at hand and \(T\) is a set of procedures that manipulate \(D\). The software model links \(D\) and the outside world, i.e. the model’s user.

The concept of software model will be more carefully studied later, in particular when defining the Model–View–Controller (MVC, Chapter 4) and the Model–Point of View–Controller (MPVC, Part II) approaches.

2.2 Software prototype

The definitions of software model and of prototype yield the following definition for a software prototype.

**Definition 6.** A software prototype \(P\) is an experimental piece of software, i.e. a model of software \(S\). Consequently, \(P = (D_P, T_P)\), where \(D_P\) is a representation of \(S\) and \(T_P\) is a set of specialized tools to build and manipulate \(D_P\).

When there is no ambiguity, *program* is used for software model and *prototype* is used for software prototype. Normally, the term ‘tools’ refers to the different programs that are needed to design a program, rather than to the different procedures in the application software.

As for software prototyping, here is its definition [9].

**Definition 7.** Software prototyping is a process of building software whose purpose is to attain information about the adequacy and the value of a software design. The prototype is normally used as the precursor to the final software. A prototype differs from the final product in that it is developed more quickly and it is more easily parameterized and manipulated, at the expense of efficiency and performance.
The prototype is therefore used to quickly extract information about the software to come.

The prototyping approach to building software is quite recent (1980s), and corresponds to a new paradigm in software engineering: 'software engineering must provide a set of methods, techniques and tools that are necessary for the building of industrial quality programs, during the entire life cycle of the product.... A technique based on repairs must be transformed into a technique based on sound designs and a priori guarantees of quality' [104].

This new paradigm is explained below, by showing how prototyping can be integrated into the basic software life cycle.

2.2.1 Boehm scheme

This basic scheme supposes that a program's life cycle can be decomposed into a series of more or less independent stages of development (see Figure 2.2). Each of the stages has a validation step that leads either to the next stage or to the previous stage. This scheme can be reduced to five fundamental stages [2].

![Software life cycle - waterfall model](image_url)
Specification consists of describing what the program must do and in making a detailed analysis of the different structural and functional requirements of the program, as defined by the client.

Design consists of describing how the program will meet those requirements. The data structures, such as file formats or variable types, are defined, as is the modular structure of the program.

Coding refers to the translation and rewriting of the design into a programming language.

Testing ensures that the program behaves correctly, i.e. is consistent with the specified requirements.

Evolution includes maintenance and subsequent improvements to the software.

The Boehm scheme [26] considers each stage in a program's development to be of equal importance. In each state, successful validation allows the next step to be begun, while failure returns to the previous stage.

This approach is often criticized for the maintenance cost that it entails. First, until coding has taken place, it is impossible to test the behavior of the software; often there are misunderstandings between the client and developer of the program. Second, since the stages are independent, each modification or update of the program requires a re-enactment of each of the stages.

2.2.2 Prototyping in the life cycle of software

'Sound designs and a priori guarantees of quality' is a paradigm that requires information about the adequacy and the utility of a program's design, right from the design phase. The software prototype becomes necessary.

Under this paradigm, the specification and design stages are fundamental, and must offer views of the program being developed; these views can be partial or complete, simplified or not. So, the life cycle must include the design of one or more prototypes during the specification stage. There are three ways to integrate prototyping into the life cycle.

First, prototyping can precede the writing of formal specifications. The resulting prototypes then only offer a glimpse of what is to be. The Boehm scheme would be preceded by a prototyping stage (see Figure 2.3).

In this case, the prototypes provide scenarios or simulations [44] that describe the behavior of the program. Simulation is carried out using precise input values that describe some of the behavior of the final product.

Second, prototyping can be an integral part of the specification phase (see Figure 2.4). There can then be several prototypes, each of which offers a partial view of the program (e.g., data structures, functions, user interface). Furthermore, if the prototype is written in an executable formalism, i.e. a programming language, then it can be tested and validated.

The prototype is then a kind of proof. It describes, for a precise set of input values, the behavior of the prototyped program. Some portions of the prototype
Figure 2.3 Prototyping before specification

Figure 2.4 Prototyping as part of specification
can be recuperated and used in the final product, and so form the basis for the final product's code.

Finally, prototyping can provide the first implementation (see Figure 2.5). In this case, the prototypes implement in a simplified but executable manner the program's different functions. Once the prototype is tested and validated by the user, the successive transformations can take place, resulting in an efficient program. This transformational approach normally takes place manually, right up to the compilation phase, although much research in artificial intelligence and software engineering is trying to automate this transformational process [10, 46, 142].

**Figure 2.5 Prototyping as part of implementation**

The prototype is seen as the first implementation, *version 0*, which only works under certain special conditions. In this case, the prototype also serves as feasibility test. Generally, the final product is obtained by completing the prototype: certain functions are optimized and others are added; the program becomes product.

Although the last two approaches resemble each other, they differ fundamentally in their perception of prototyping. In the latter, the prototype must be better specified, more detailed, while in the former, it can remain more abstract, more declarative.
2.3 Rapid prototyping

A software prototype is designed for a privileged user, hereafter referred to as an expert, who differs from the average user in that he or she participates in the design and development of the product. The expert must criticize the prototype, providing the raw data that is necessary to understand the problem, in addition to keeping a watchful eye on the developing program. The mutually dependent relationship between expert and developer requires a dialogue between the two; the prototype is a support for the dialogue.

Prototyping can therefore be seen as an interactive and iterative decision-making process between the expert and the developer [175] (see Figure 2.6 [121]).

![Figure 2.6 Prototyping life cycle](image)

Together, the expert and the developer define the specifications and the requirements of the critical parts of the system. Next, the developer develops a first prototype. This stage is often difficult, as it requires developing an adequate formalism for the subject; this fundamental problem of knowledge representation does not necessarily have an immediate solution [70, 182].

Once the prototype has been built, the expert evaluates it, through various experiments and tests, both structurally and functionally, according to the requirements. If the prototype does not correspond to the specified requirements or if it has behavioral inconsistencies, the expert identifies the different problems and restarts the prototyping stage with the developer. For example, during a graphical animation, if the prototype does not take into account the relative time to execute two tasks, then the prototype must be redesigned.

This process continues until the expert validates the prototype, i.e. accepts it as model.
For prototyping to be done effectively, it must be done in a specialized prototyping environment. This environment must allow rapid prototyping, i.e. it should be faster than normal program development. The developer must be able to quickly change specifications, improve prototypes and, consequently, provide the expert with different working versions of the prototype. Furthermore, the prototype must be easily understood. The prototype is to be used by an expert, normally a non-computer scientist, and is an experimental version of the program. It must therefore provide the functional, user-friendly and ergonomic means for the manipulation of the prototype's parameters [11].

Now that the requirements of a prototyping environment have been specified, its components must be addressed.

2.3.1 Prototyping languages

The developer uses a programming language as formalism to produce the prototype. Although the model formalism is often specific to the problem or to a restricted class of problems, the formalism used to produce the prototype must be sufficiently generic – as opposed to specific – to allow the production of several different prototypes. Here, we consider only those programming languages that are well suited to prototyping.

To facilitate prototyping, a formalism must satisfy the three following attributes.

First, the formalism must allow modularity, i.e. the developer should be able to break up prototypes into independent modules. Studies have shown that most problems that arise during software modification are due to a too great interdependency between the program’s modules [114]. Good modularity should simplify succeeding modifications of the prototype.

Second, the formalism must encourage reuse, i.e. the developer should be able to recuperate existing modules and incorporate them into a new prototype. These modules should have a well-defined interface, defining the externally visible operations, which give access to the values and functions in the module, allowing the developer to manipulate the modules without knowing their implementation details. The set of reusable modules constitute an integrated complementary formalism [94], since they are written in the prototyping language.

Third, the formalism must allow data abstraction, i.e. the developer should be able to represent and manipulate knowledge from different fields, such as user interfaces (window, mouse, menus) and parallel programming (multiprocessors, multitasking, process synchronization).

This diversity of concepts can be found on large projects that combine different fields. For example, a real-time application of rockets is subject to several kinds of constraint: timing (the rocket's position is transmitted every tenth of a second), geographic (the application receives information from tracking radars that are geographically separated), graphic (the rocket's trajectory must be followed on a control screen) and ergonomic (the application is connected to a flight panel with
switches to view different aspects of the flight, such as trajectory, altitude, impact point and loss of trajectory).

The abstraction of concepts improves the structural modularity of prototypes and eases the creation and exchange of reusable functional and structural components.

Solutions to these formalization needs are sought among the three main state-of-the-art approaches in knowledge representation [70, 174].

**Analytical approach**

This approach treats knowledge as a set of assertions about a particular domain. These include laws about the domain (e.g., physical laws), statements about the behavior of the system (e.g., ‘the motor turns the fanbelt’), facts drawn from experience (e.g., ‘motors stall in the cold’) and general statements (e.g., ‘rust on the body shows the wear and tear on the car’). These assertions can then be handled by deductive systems, such as inference motors [181, 182], which, starting from basic axioms, attempt to draw all inferences. The assertions are expressed in a formal language with a compositional syntax and semantics, i.e. where the meaning of a sentence is obtained from the meaning of its components.

In this case, prototyping consists of specifying, implementing and adjusting the facts and rules that make up the prototype.

**Structural approach**

This approach treats knowledge as a network of concepts and is used in AI with semantic networks [149, 150], schemes [161] and frames [131]. Knowledge of a subject consists of the transitive closure of the relations between nodes or *cognitive atoms* [190] of the network.

This kind of knowledge representation is used, for example, in natural language understanding [21, 34, 99], where a semantic graph represents the deep structure of the analyzed sentence.

Most concepts manipulated by class languages [123], such as property inheritance and the difference between generic (class) and individual (instance) concepts, fall within this framework. These include Smalltalk [75, 76], ObjV Lisp [49] and Simula [22]; other object-oriented families include frame languages (KRL [25]), actor languages (*PLASMA* [87]) and hybrid languages (*LOOPs* [24] and MERING [68]).

In this case, the prototype defines a framework modeling the facts. Prototyping tests, refines and validates this framework. This formalism is examined in detail in subsequent chapters, with the presentation of object-oriented programming.

**Interactionist approach**

This is the most recent approach. It is based on the principle that structure, in particular cognitive structure, is the result of aggregation and composition, which follow from the encounter of individual actions performed by interacting autonomous *actors* or *agents*.
Knowledge is no longer perceived as a system whose structure is the result of a single thinking being but, rather, as a universe composed of agents in interaction. Minsky writes:

I'll call 'Society of Mind' this scheme in which each mind is made of many smaller processes. These we'll call agents. Each mental agent by itself can only do some simple thing that needs no mind or thought at all. Yet when we join these agents in societies— in certain very special ways— this leads to true intelligence [132, p. 17].

Knowledge is therefore distributed to each agent, depending on its specificity. Decision making then takes place in an interactive manner, through the use of cooperation and negotiation mechanisms that relate the expert agents' different points of view.

The prototype is then a micro-society-of-mind. Adjusting it includes setting up adequate mechanisms to resolve social conflicts that are found within the prototype [86], as in the use of compromise. For example, there is a conflict between the builder agent and the wrecker agent within a child playing with cubes. One possible compromise: 'Please, Wrecker, wait a moment more till (sic) Builder adds just one more block: it's worth it for a louder crash' [132, pp. 33].

2.3.2 Prototyping tools

Prototyping is supposed to quickly present the expert with a relevant prototype, i.e. one that takes into account the structure and behavior of the prototyped software. To succeed, the developer uses easily used software tools, called prototyping tools, that allow the rapid generation and adaptation of different versions of the prototype.

There are a great number of prototyping tools. This diversity arises from the variety of situations in which prototyping takes place and from the diversity of prototypes. Carey and Mason [44], in studying prototyping techniques, tools and methodologies, considered in 1983 that the diversity of prototyping tools showed that developers did not have tools that were well suited to a prototype development methodology.

Today, although it is clearly too soon to speak of a prototyping methodology, most prototyping environments have an essentially identical approach, based on objects [82]. In fact, the software prototype paradigm became popular in the 1980s, at the same time as object-oriented languages.

Most prototyping tools can be placed in one of two categories: application generators and toolboxes.

Application generators
First developed as programming aids, application generators [162] allow the developer to describe a prototype by filling in, completing and assembling graphical or
textual prefabricated schemes. A program scheme is a data structure or program skeleton that the developer completes, adapts and refines according to the needs of the system.

For example, a graphical application generator allows the developer to draw the application by piecing together graphical representations of, among others, program schemes (e.g., loop, conditional, assignment, subroutine call), VLSI constructions (e.g., bus, register, multiplexer, gate [82]) and user interfaces (e.g., activation button, scrollbar, dialog sequence [36, 51, 93, 115]).

Each family of schemes is typically placed in a specialized database, where the developer finds the prototype's components. By doing this, the presentation of different applications is unified.

Generators follow a construction process, including task sequencing, that is specific to the domain. For example, a file management application generator asks the developer what file formats are needed, then what kinds of index are required. Since these functions are all known, the developer indicates what functions are required (e.g., file creation, insertion, file deletion and label generation).

Some generators offer assistance to the developer at all steps. This assistance can be implicit, such as by limiting the possible choices with respect to previous choices, or explicit, such as online documentation.

Once the description is finished, the generator automatically completes the implementation details of the application and generates executable code.

The major problem with generators is that their efficiency, be it at the item, tool or generated code level, is correlated with the degree of specialization: the most efficient generators are those that cover a restricted part of a precise domain and that can therefore offer a complete database for the domain.

Toolboxes

Toolboxes are generic applications [163]. They provide general tools that are used to develop applications, such as window managers, scrollbars, keyboard and mouse event managers. They are supplied as object boxes, and the developer uses the required structures and functions.

The formalism used to implement toolboxes directly uses concepts developed for object-oriented programming. More than just simple libraries of functions and structures, these generators offer a knowledge-representation-oriented formalism as well as a data-directed application design, called object-oriented approach, further developed in the following chapters.

Unlike application generators, toolboxes are generally associated with a programming language. For example, Mac App [163] is based on Object Pascal [164], while Flavors [134] and Ceyx [92] are object-oriented extensions of LISP. Therefore, the developer has direct access to pre-existing objects, making it possible to adapt, modify or reuse them, depending on the prototype's need.

Generally, toolboxes are less specialized than generator databases. However, it is possible to develop specialized object sub-boxes.
The main problem with these generic generators is that there is no construction process. The developer must do everything. Learning generally takes place by reading the code of existing applications (e.g., data structures used for a particular representation, or mechanisms to intercept keyboard events) or by using tools offered by the programming environment. Tracers, steppers and breaks from the programming environment [155, 177, 190] provide interactive means to view certain aspects of the dynamic behavior of the program and to control the system's state at pre-determined moments. Formal verification systems [71, 83, 88, 102], symbolic evaluators [77], partial evaluators [165] and syntactic editors [64] offer means to formally develop programs.

There now exist a number of relevant formalisms for prototype construction. What is missing are tools that allow these formalisms to be quickly used to the maximum extent.
Chapter 3

Object-oriented languages

Smalltalk-80 is more than a programming language. It is also an operating system, a programming environment, even a methodology for software design. It is the ideal environment for rapid prototyping [17].

To answer ever-increasing programming needs, many languages have been invented. Among them, object-oriented languages [123, 146, 172] offer a new approach to programming based on abstract data types [72], data and functional encapsulation in the same object [53], hierarchical organization of the objects [38], subtyping polymorphism [43] and a generic activation mechanism for message passing [69].

This approach, called object-oriented programming, is used as base formalism for knowledge representation [32, 70, 182], as programming method [48, 53, 56, 128], as design method for strongly interactive applications [8, 27, 33, 105] and as rapid prototyping method [82, 108].

This chapter presents the essential aspects of object-oriented languages, by focusing on prototyping.

3.1 Objects, messages and encapsulation

3.1.1 Objects

The Simula language [22, 56], designed for simulation problems, was the first to combine data and programs in the same entity, called an object. In 1968, Alan Kay took up the fundamental aspects of Simula to create a 'user-friendly computer' [97]. He wanted it easily programmable and integrated new peripherals: the mouse and the bitmap screen. To this end, he created the Flexible Extensible Language (FLEX) [96], then Smalltalk-72. ('We called Smalltalk Smalltalk so that no-one would expect anything of it.' [98])
In an object-oriented language, an object is an entity that consists of two components. The static component consists of the object’s data. These data, the object’s memory [117], describe the object’s structural characteristics and its state when consulted. The dynamic component consists of the object’s behavioral or functional characteristics (the methods), i.e. what the object can do. The set of available methods corresponds to the object’s processing power [117].

The object is the basic unit of an object-oriented application, just as the action is the basic unit of functional or imperative applications written in languages such as Pascal [194] or LISP [124]. This kind of programming is called data-directed, as opposed to treatment-directed.

Focusing on the data rather than on programs is reminiscent of the speed-aperture duality in photography. When the diaphragm is considered more important, the focus is placed on the environment in which the subject evolves: depth of vision. For speed, on the other hand, the focus is placed on the activity, more precisely on the speed at which the subject evolves. These two aspects, opposites but inter-related, deal with the same subject, each in a different manner. In fact, the speed must be set correctly when the focus is placed on aperture; the photo might be blurred if the speed is too low. Similarly, the aperture must be correctly set according to the speed, to ensure a proper depth of vision.

This distinction can be illustrated with a LOGO programming example: learning the word ‘house’ [141, p. 81]: ‘Pamela began by teaching the computer the words square and triangle. Then she noticed that she could make a house: all she needed was to place the triangle above the square.’

In LOGO, a member of the same family of programming languages as LISP, the house is created through the necessary actions to visualize it. The house is seen as the juxtaposition of the drawings of the different parts, as was determined by a functional analysis of its graphical representation (see Figure 3.1).

```logo
for HOUSE
  SQUARE
  right 30
  TRIANGLE
end
```

Figure 3.1 A house in LOGO

The two sub-drawings are juxtaposed by an extra bit of code that places the cursor in the right place in the right direction, in order to properly locate the second drawing.

The house drawing program in a language such as LOGO is therefore the sequential description of the different actions to obtain the drawing. There is no programming representation for the house concept, independent of the actions. Only the program exists, and it only offers a particular point of view for the concept.
In object-oriented languages, the action is less important than the structure of the manipulated data. Drawing a house is just a behavioral aspect of the conceptual representation of house.

Consequently, 'learning' a house is first done structurally: a house consists of a wall and a roof, these two still to be defined. The static component of the House class consists of the two variables myWall and myRoof (see Figure 3.2):

![Figure 3.2 An object-oriented house](image)

### 3.1.2 Messages

Objects communicate by transmitting queries or messages (see Figure 3.3). A message asks an object to execute one of its methods, the object's set of dynamic behaviors. It is up to the receiver of the message to fulfill the request; in some languages, the object can actually refuse to fulfill the request or can pass it on to another object; it can even lie, if the object is obsolete [116].

![Figure 3.3 Message passing](image)

A message is a pair (selector, arguments), where selector is the name of the desired behavior and arguments are the method's actual parameters.

*Drawing a house* is an example request. It is only activated when the house concept is enriched with a graphical description.

```scheme
(drawYourself
  "Graphical representation of the message receiver, here a house"
  myWall drawYourself.
  myRoof drawYourself)
```
This method, part of the dynamic part of an instance of the House class, describes the required behavior when a house receives the message drawYourself. The graphical representation of the receiver requires the graphical representation of its two instance variables, myWall and myRoof. In this case, the functional decomposition is deduced from the structural decomposition.

In LOGO, the reverse was true: the structural decomposition was derived from the functional decomposition, which showed that a house could be drawn by placing a triangle above a square.

3.1.3 Encapsulation

An object’s data are private, i.e. local to the object. The external world only has access to an object’s structure, be it for consultation or modification, through the dynamic part, i.e. through the methods offered by the object. Some languages even allow objects to have private methods, known only to the members of the same class (e.g., in Eiffel) or to a precise set of classes (e.g., in C++), that are not available to other objects.

The set of messages that can be sent to an object, i.e. the object’s set of behaviors, forms the interface between the object and the external world, i.e. the other objects. The methods are said to encapsulate the object, in particular, its internal structure [53] (see Figure 3.4).

![Figure 3.4 Encapsulating the internal structure](image)

Object-oriented programming consists of representing physical objects (e.g., calculators, rockets or plants) and mental concepts (e.g., functions, characters, word lists or relations between entities) as computer objects, i.e. (structure, methods) pairs.

A programmer must identify and define the objects that constitute the application’s universe. Then the behaviors of each object must be analyzed and written. Running these behaviors means running an object-oriented program: each object detains a particular aspect of the software; this shared tasking can be found in Minsky’s agents [132].
This approach is well suited to prototyping. The programmer must model each of the component objects of the domain under study. The \( (\text{structure}, \text{methods}) \) pair from object-oriented languages corresponds perfectly to the \( (\text{domain}, \text{tools}) \) pair that describes software prototypes.

### 3.2 Classes, instances and inheritance

#### 3.2.1 Classes and instances

A **class** is a structure that regroups objects of the same kind, i.e. that belong to the same family and that understand messages in the same manner. A class is a mold for objects, similar to an abstract data type. Objects from the same mold are the **instances** of that class.

A class describes the **generic** characteristics of its instances. The generic notion refers to what is shared by all instances, as well as to the notion of being **generated**. The static characteristics, also known as **instance variables** or **attributes**, describe the structure of the family of objects. The dynamic attributes are the different behaviors; each of these is described by a specific method, parameterized by a receiving instance of a message (**self** or **super** pseudo-variables in Smalltalk-80), the instantiated variables of the receiver and the method’s formal parameters.

Object-oriented programming consists of (1) defining the object classes, thereby formalizing, or designing, the structures and (individual and relational) behaviors of the entities of the domain under study; and (2) instantiating these classes, creating representatives to study their individual and collective behaviors, i.e. manipulating the entities in a micro- or macrocosm outside of the original domain.

To design **good** objects, the following approach can be used [27]. First, identify the objects in the real world by determining the static component of each class of objects. Next, identify the actions (the dynamic component) that each object can undergo or provoke. Then, in turn, identify the relations between objects, determine the precise interface between an object and the external world and, finally, implement the objects.

The two steps in the building and manipulation of an object application are reminiscent of the first two steps in modeling (see Section 1.3). Hence, designing classes actually means designing a software prototype; the developer experiments with object structures and behaviors by creating classes and methods. Similarly, instantiating classes means fixing, temporarily at first, the structures and behaviors in the individual objects in the domain and testing and manipulating the resulting experimental object model. The result is, of course, the prototype–model double point of view for software (see Figure 3.5).

Therefore the object-oriented approach is well suited to prototyping, simply because it is its descendant. The flexibility and the ease of implementation and use of such an experimental model, i.e. a prototype, are fundamental for prototyping.
Figure 3.5 Object-oriented view of prototyping

An object-oriented prototype development environment must ease the building, reuse and modification of classes, i.e. the manipulation of prototypes, as well as the instantiation of classes and the consultation and manipulation of the resulting instances.

One of the fundamental aspects of this ease of use comes from the dynamic typing of the manipulated objects. There are, in fact, two major schools on typing for object-oriented languages (see reference [123] for a full discussion).

The Scandinavian school includes the statically typed object-oriented languages (e.g., Simula [56], C++ [173] or Eiffel [128]) that require complete typing of an application, of instance variables, local variables, formal parameters and method results. This completeness requirement is well suited to the coding in the final phase; it allows the object-oriented approach to be retained, while taking advantage of the efficiency of compiled code. But it does so by reducing flexibility and ease of use.

The Smalltalk school includes object-oriented languages with dynamic typing (e.g., Flavors [134], Ceyx [92], Formes, ObjVLisp [49], CLOS [23, 100] or LORE [15]), in which the types of manipulated objects are only known at execution time. Dynamic typing gives the programmer maximum freedom, at the cost of efficiency and reliability, which are not fundamental preoccupations in prototyping. Building and experimenting with a formalism can be done in parallel, by adjusting, as need be, the manipulated structures and the implemented behaviors.

3.2.2 The rectangle class

Our statements are illustrated through the prototyping of the 'rectangle' geometric concept, which can be addressed in at least three ways (see Figure 3.6).

The most commonly used approach is to view a rectangle through the two endpoints of a diagonal, i.e. the origin and the corner. This representation is well suited to the vectorial graphic representation of rectangles: no computation is nec-
necessary to find the coordinates of the four line segments, thereby improving efficiency (see Figure 3.7).

Let origin = \((X_o, Y_o)\) and corner = \((X_c, Y_c)\). The line segments are computed as follows:

\[
S_1 = \{ \ (X_o, Y_o) \rightarrow (X_c, Y_c) \},
S_2 = \{ \ (X_o, Y_o) \rightarrow (X_c, Y_c) \},
S_3 = \{ \ (X_c, Y_c) \rightarrow (X_o, Y_c) \},
S_4 = \{ \ (X_c, Y_c) \rightarrow (X_o, Y_o) \}.
\]

A rectangle can also be defined by its length, its width and by the position of one of its right angles. This approach, commonly found in schoolbooks, has the advantage that examination of the static attributes gives an immediate idea of the rectangle's form. A rectangle defined as

\[
\text{origin} = (98.219, 23.931), \text{corner} = (101.365, 27.077)
\]
gives no idea that the rectangle is in fact a square, while this fact is immediate in

\[
\text{origin} = (98.219, 23.931), \text{length} = 3.146, \text{width} = 3.146.
\]

Finally, a rectangle can be defined by its barycenter, its area and its width. This approach focuses on other attributes of the same geometric object and uses other attributes.

These different representations of the same geometric concept are of no consequence to the expert, who has access to the Rectangle class's instances through the interface methods supplied by the developer. However, the developer must
choose the most appropriate representation, i.e. the one that is best adapted to the needs of the prototype.

Table 3.1 compares the three Rectangle classes, along with the resulting interface methods. The ‘~’ means that the result of the following expression is returned as the method’s result. The binary ‘@’ symbol creates a Point class instance, the x coordinate receiving the operator and the y coordinate as argument. In Smalltalk, the self pseudo-operator is used to refer to a message’s receiver.

These different formalizations allow the developer to experiment with different representations of the same concept, the rectangle, in accordance with the expert’s needs. These examples illustrate the data abstraction and encapsulation capacities of object-oriented languages.

<table>
<thead>
<tr>
<th>Class:</th>
<th>Class:</th>
<th>Class:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>Rectangle</td>
<td>Rectangle</td>
</tr>
<tr>
<td>Attributes:</td>
<td>Attributes:</td>
<td>Attributes:</td>
</tr>
<tr>
<td>origin</td>
<td>origin</td>
<td>origin</td>
</tr>
<tr>
<td>corner</td>
<td>width</td>
<td>surface</td>
</tr>
<tr>
<td>Methods:</td>
<td>Methods:</td>
<td>Methods:</td>
</tr>
<tr>
<td>origin</td>
<td>origin</td>
<td>origin</td>
</tr>
<tr>
<td>~origin</td>
<td>~origin</td>
<td>~origin</td>
</tr>
<tr>
<td>extremity</td>
<td>extremity</td>
<td>extremity</td>
</tr>
<tr>
<td>~corner</td>
<td>~origin +</td>
<td>center +</td>
</tr>
<tr>
<td></td>
<td>(width @ height)</td>
<td>(width/2 @ (self width/2))</td>
</tr>
<tr>
<td>center</td>
<td>center</td>
<td>center</td>
</tr>
<tr>
<td>~(origin+corner)/2</td>
<td>~origin +</td>
<td>~center -</td>
</tr>
<tr>
<td></td>
<td>((width/2) @ (height/2))</td>
<td>((width/2) @ (self height/2))</td>
</tr>
<tr>
<td>height</td>
<td>height</td>
<td>height</td>
</tr>
<tr>
<td>(corner-origin) x</td>
<td>width</td>
<td>width</td>
</tr>
<tr>
<td>width</td>
<td>(corner-origin) y</td>
<td>area</td>
</tr>
<tr>
<td>area</td>
<td>self height</td>
<td>height*width</td>
</tr>
<tr>
<td>self width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.2.3 Inheritance

Each of the three Rectangle classes, defined in the previous subsection, allows the geometric concept to be manipulated independently of any graphical concept. Adding graphical attributes, both static and dynamic, means specializing the concept for the graphical representation of geometric objects on a bitmap screen.

To represent a rectangle graphically, the initial concept must be enriched with three new attributes: the internal color, the edge color, and the edge thickness (see Figure 3.8).

![A graphical rectangle](image)

**Figure 3.8** A graphical rectangle

This graphical specialization can be made by defining a new RectangleGraphic class that regroups the general rectangle attributes and the new graphic ones, giving the following partial definition.

<table>
<thead>
<tr>
<th>Class</th>
<th>RectangleGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>origin, corner</td>
</tr>
<tr>
<td></td>
<td>colorBackground</td>
</tr>
<tr>
<td></td>
<td>colorEdge</td>
</tr>
<tr>
<td></td>
<td>edgeThickness</td>
</tr>
</tbody>
</table>

Implementing refinements in this manner would be neither practical nor economical; knowledge would be shared inefficiently, i.e. would be redundant.

A class can be perceived as a reservoir of knowledge from which new, more specialized classes can be defined; the latter complete the knowledge in the parent class. An inheritance relation [170] relates a class to its parent class, the superclass; the child class is the parent's subclass. This transitive relation makes available the attributes of the superclass and the other classes that are above.

So, the Rectangle class is specialized by the creation of the RectangleGraphic subclass, which inherits the attributes of the Rectangle superclass and specializes the concept for graphical purposes (see Figure 3.9).

The inheritance mechanism can be seen as a virtual copy mechanism. It avoids the redefinition of structures and the rewriting of methods that can be shared by different classes of objects.
More generally, any class is defined as a subclass of another class, i.e. classes are organized hierarchically; consequently, so is the knowledge represented by these classes. The graphical representation is called an inheritance graph, which has a single root, the Object class; the latter includes the attributes that are shared by all objects in the system, e.g., the class method returns the receiving object’s class and the copy method returns a copy of the receiving object. Figure 3.10 shows the class sub-hierarchy for some geometric objects.

Therefore, the object-oriented approach to knowledge representation consists of successive refinements. Creating an object-oriented program consists of defining more specialized subclasses, i.e. specializing the existing knowledge environment by adapting it to one’s own knowledge.

As the development environment is specialized by the expert and the developer for the particular domain, prototyping becomes simpler, as less new knowledge must be incorporated.

An analogous representation for knowledge can be found in semantic networks, such as KL-ONE [34, 99], where the inheritance relation is an a kind of (ako) relation (subset) and where the instantiation relation is an is a (isa) relation (set membership). The parallel between an inheritance graph and a semantic network does not remove essential problems in knowledge representation, such as how to interpret multiple inheritance, where a class may have several immediate superclasses [30, 187]: depending on the manner in which multiple inheritance is used, either the intersection or the union of inherited concepts may be expressed [63].
Figure 3.10 A geometric object class sub-hierarchy

Nevertheless, the author feels that this parallel with semantic networks cases, in most cases, the designation and building of a coherent inheritance graph for the concepts in the field under study.
Chapter 4

Prototyping a calculator

This chapter shows how object-oriented programming contributes to prototyping, in particular to prototype programming. The example used throughout this chapter is the rapid prototyping of a postfix calculator, i.e. the implementation of an application object simulating the behavior of a postfix calculator that resembles the one in Figure 4.1.

![Calculator Interface](image)

**Figure 4.1 Sample calculator interface**

This example presents the Model–View–Controller (MVC) scheme [105], which is used to implement interactive applications in Smalltalk-80. It is the basis for the Model–Point of View–Controller (MPVC) scheme [111], the subject of the second part of this book.
4.1 Conceptualization

The first step in prototyping is to conceptualize and to reify the entities in the field under study, in this case a calculator.

A calculator consists of the central unit, which handles the data and does the calculations; the screen, which visualizes the results; and the keys, which allow the data to be entered and the operations to be enabled.

For each kind of object, the static and dynamic attributes must be defined and regrouped in a class. The prototype consists of instances of the defined classes.

4.1.1 The central unit

The central unit (CU) has two static attributes. First, the accumulator, a numeric register, contains either the last value entered or the result of the last computation. Second, the CU has a connection with the screen, in order to transmit the last computation’s result, or to read the number in the screen’s buffer.

The CalculatorPostfix class models the CU. It has two instance variables, accumulator and screen, which correspond to the two static attributes. There is no need to inherit anything special, so it is defined as a subclass of Object.

```
Class : CalculatorPostfix
Superclass : Object
Attributes : accumulator
            screen
```

The accumulator is initialized to 0; during computations, it contains the numeric values. The screen refers to the instance of class Screen (described below) that corresponds to the current calculator.

There are also two dynamic attributes and two special functions.

The CU can apply a unary arithmetic operation to the number on the screen. The number is recovered, the operation is applied and the result is returned to the screen. The screen’s buffer therefore serves as memory register, just like the accumulator. (By doing this, a unary function may be applied while another operand for a binary operation may be stored in the accumulator, as in the expression \(\text{fac}(5)+\text{fac}(3)\).) Here is the definition.

```
execOperationUnary: operator
  "Recover the screen value, send the operator argument to that value and transfer the result to the screen"
  screen value: (screen value perform: operator)
```

where perform: sends a computed unary message to the receiver; the evaluated argument must return the message selector.
The CU can also apply binary operations, whose operands are the accumulator and the number on the screen. In this case, the CU applies the binary operator to the accumulator, the screen being the argument. The result is memorized in the screen. Here is the definition.

```plaintext
execOperationBinary: operator
  "Send the operator argument to the accumulator, along with the screen value as argument, and return the result to the screen"
  screen value: (accumulator perform: operator with: screen value)
```

where the perform:with: message is used to send messages with several arguments; the first argument returns the message selector and the second argument returns an array with the actual parameters for the message.

The init function, activated by the C key, reinitializes the accumulator and sets the screen value to 0.

```plaintext
init
  "Initialization function"
  accumulator <- 0.
  screen value: accumulator
```

The enter function, activated by the enter key, finalizes the screen value.

```plaintext
enter
  "Input validation function"
  screen validate
```

### 4.1.2 The screen

The screen consists essentially of a buffer containing the computed values. It also has a Boolean flag indicating whether the next entered digit should replace the entire buffer or simply be appended; replacement occurs after the number has been validated with the enter key or after a computation has taken place. Finally, there is a connection to the CU, i.e. the calculator, so that the entered value may be transferred; since the screen buffer is used as a register by the CU (for unary operations) when the screen buffer is to be replaced, its value is sent to the CU, which places it in its accumulator.

Therefore, the Screen class has three instance variables.

<table>
<thead>
<tr>
<th>Class</th>
<th>Screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass</td>
<td>Object</td>
</tr>
<tr>
<td>Attributes</td>
<td>buffer replace calculator</td>
</tr>
</tbody>
</table>
where buffer is a sequence of characters corresponding to the entered digits or the
digits of the result of the last computation; replace is a Boolean value indicating if
input should replace the existing buffer; and calculator is the calculator instance
attached to this screen.

A Screen instance must receive an input digit and either concatenate it to the
buffer or replace the buffer with the digit, depending on the value of replace.

```plaintext
input: aDigit
  "A digit has been input:
    Should it replace the buffer?"
  if True: [calculator accumulator: self value.
      buffer <- aDigit.
      recover <- false]
  if False: [buffer <- buffer, aDigit]
```

When replace is true, the screen first transfers the buffer contents to the calcula-
tor's accumulator, then it places the new digit in its buffer and toggles replace's
value.

The CalculatorPostfix class definition requires that the screen send it the
numeric value associated with the string of digits in its buffer. The screen responds
to this request through the value method, which uses the asNumber method of the
String class to convert the string of characters to a number.

```plaintext
value
  "Return the numeric value corresponding
to the buffer's contents"
  buffer asNumber
```

Similarly, the screen must receive a number sent by the CU and convert it to a
string of characters before placing it in the buffer. This is done with the value:
method.

```plaintext
value: aNumber
  "Set the buffer with the string equivalent"
  "of the number and validate the input"
  buffer <- aNumber printString.
  self validate
```

Finally, the validate method sets replace to true.

```plaintext
validate
  "The next input digit must replace the buffer"
  replace <- true
```
4.1.3 The keys

The calculator keyboard contains numeric keys (0, 1, ..., 9, .), to enter numbers, and function keys. The latter are divided into binary function keys (+, -, *), which take as operands the accumulator and the screen buffer, unary function keys (fib, fac), which take as operand the screen buffer, and special keys (C, enter), which operate on the entire calculator.

The Key class

The Key class models the attributes held in common by all keys. In particular, they all have a label on their surface, indicating their function or their value.

Class : Key
Superclass : Object
Attribute : label

The label variable is the character string on the key’s surface.

The only dynamic attribute of a key is the action to be performed when it is pressed. Since this action is not universal, the behavior cannot be described at this level of the hierarchy; it must be described for each subclass. This is known in Smalltalk-80 as subclass responsibility, and is presented in more detail in Section 8.2.2.

The KeyNum class

Specialization for numeric keys deals with both static and dynamic issues. A numeric key is directly linked to the calculator’s screen, so that it may send its label when it is pressed. Class KeyNum is a subclass of Key, with a new instance variable, screen, indicating the screen to which the label must be sent.

Class : KeyNum
Superclass : Key
Attribute : screen

There is also a new method, activate, which is called when the key is pressed.

activate
   "Inform the screen that a digit has been input,
    by sending the label"
   screen input: label

When an instance of KeyNum receives the activate message, its label is transmitted to the screen by sending the input: message defined in class Screen.

The KeyFunction class

Function keys need specialization with respect to the static attributes. Each key needs to know the associated function, as well as the CU that is supposed to effect the action. Class KeyFunction is also a subclass of Key.
Class : KeyFunction
Superclass : Key
Attributes : function
calculator

The function variable is the identifier of the function to be activated when the function key is pressed.

Just as for Key, the method for the key’s action cannot be defined; the function’s type (unary, binary or special) is required. This problem is solved by creating three subclasses for KeyFunction.

The **KeyFuncSpecial** class
This class models a special function key:

Class : KeyFuncSpecial
Superclass : KeyFunction

Activating a special key implies the immediate transmission of the special function to the calculator.

```plaintext
activate
   "Call the CU to execute the special function"
   calculator perform: function
```

The **KeyFuncUnary** class
This class models a unary function key.

Class : KeyFuncUnary
Superclass : KeyFunction

Activating a unary function key implies sending the `execOperationUnary:` message to the calculator, with the function as argument.

```plaintext
activate
   "Call the CU to execute the unary operation"
   calculator execOperationUnary: function
```

The **KeyFuncBinary** class
This class models a binary function key.

Class : KeyFuncBinary
Superclass : KeyFunction

Activating a binary function key implies sending the `execOperationBinary:` message to the calculator, with the function as argument.

```plaintext
activate
   "Call the CU to execute the binary operation"
   calculator execOperationBinary: function
```
The **Key class subhierarchy**
Figure 4.2 contains the inheritance graph describing the hierarchical organization of the different keys.

![Figure 4.2 Key class sub-hierarchy](image)

### 4.2 Testing the prototype

The next step in prototyping is to test and manipulate the implemented prototype. It is now time to test the first version of a calculator. Instances of numeric keys, function keys, screen and central unit are all created and initialized (the previous section did not present initialization methods, but they do exist for each class), and the proper links between instances are made. Figure 4.3 shows an organization scheme for the different instances.

![Figure 4.3 Variables of the different instances](image)
Table 4.1 presents the input and the execution trace of the expression $30 + \text{fac}(5)$. The first column shows the sequence of keys that are pressed to achieve the desired computation. The second column shows the successive states of the screen buffer and the third column the successive values of the CU's accumulator.

<table>
<thead>
<tr>
<th>Activated keys</th>
<th>Buffer value</th>
<th>Accumulator value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>'3'</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>'30'</td>
<td>0</td>
</tr>
<tr>
<td>enter</td>
<td>'30'</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>'5'</td>
<td>30</td>
</tr>
<tr>
<td>fac</td>
<td>'120'</td>
<td>30</td>
</tr>
<tr>
<td>fac</td>
<td>'150'</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.1 Evaluating expression $30 + \text{fac}(5)$

Here is how the execution takes place.

1. Key 3 is pressed. Immediately, as for all keys, message activate is sent to the appropriate key, in this case, the instance of KeyNum with label '3'. The activate method is executed, and it sends message input: to the screen, with argument '3'. When the message is received, the replace variable being true, the buffer's value of '0' is sent to the accumulator as 0. The buffer's value is then replaced with the value '3'. The replace variable changes to false.

2. Key 0 is pressed. The same scenario takes place, but since replace is now false, the digit is concatenated to the buffer, which now contains '30'.

3. Key enter is pressed. The activate method of the appropriate instance of KeyFuncSpecial is called. The result is that the CU is sent the enter message, which sets the replace variable to true to allow new input to be typed in.

4. Key 5 is pressed. The same scenario as for action 1 takes place: the buffer's value of '30' is sent to the accumulator as 30; it is then replaced with the value 5.

5. Key fac is pressed. The activate method of the appropriate instance of KeyFuncUnary is called, and so the CU receives the execOperationUnary: message with fact as argument. The CU executes the message by applying fact to the screen's buffer ('5'), and places the result of '120' inside the same buffer; the replace variable is reset to true.

6. Key + is pressed. The activate method of the appropriate instance of KeyFuncBinary is called, and so the CU receives the execOperationBinary: message with + as argument. The CU sends this operator to the number in the accumulator (30), along with the value in the screen buffer (120) as
argument, i.e. the CU sends the message ‘+ 120’ to the number 30. The result of the addition (150) is placed in the screen buffer.

The prototype satisfies the expressed requirements, i.e. it simulates the internal behavior of a postfix calculator. It can be considered to be a satisfactory model; it is in fact the notion of model found in MVC systems. Only the graphical interface, which simplifies the use of the calculator and gives it its final appearance, needs to be defined.

4.3 Dependency mechanism

One of Smalltalk's particular features, in fact one of its strongest points, is the dependency mechanism [109, 127]. It is the basic mechanism for implementing the propagation of constraints between objects [29, 152]. Through a message, objects depending on another object can be informed that the latter has been modified. Using this mechanism to implement a graphical interface for a Smalltalk application ensures that the existing code is not cluttered. The dependency link between the model and its graphical representations informs the representations, in a quasi-transparent manner, of the changes undergone by the model.

4.3.1 Example

Before explaining the dependency mechanism’s implementation, a non-graphical example of this mechanism is presented: propagating existing constraints between command keys on a cassette player (see Figure 4.4).

![Figure 4.4 Interface for a cassette player](image)

There are mechanical constraints between all the keys: pressing a key, i.e. activating a command, triggers the deactivation of any command under execution (RECORD, PLAY, REWIND, FAST FORWARD, STOP).

Formalizing these constraints requires that the command keys first be modeled. There are two kinds of key: the immediate command key (STOP) and the switchable command keys (RECORD, PLAY, REWIND, FAST FORWARD), which toggle between active and inactive states.
Therefore, two subclasses of the Key class must be defined. KeyCmdImmediate prototypes the dynamic attributes of the immediate command key.

Class : KeyCmdImmediate
Superclass : Key

The KeyCmdSwitchable class does the same for the static and dynamic attributes of switchable command keys; the Boolean state variable describes whether the command is active.

Class : KeyCmdSwitchable
Superclass : Key
Attribute : state

The basic dynamic attribute of each of these keys is its behavior when they are pressed: the STOP key informs the others that it has been activated so that they may be deactivated; each switchable key informs the others of its activation, switches to the active state, and begins its execution.

In Smalltalk, these links are represented by pointers between an object and its dependents (see Figure 4.5).

The dependency mechanism uses these pointers to find dependent objects and to inform them of changes. The activation of the mechanism is made by the object undergoing the change; at that instant, it is the only one capable of doing so. It sends the changed message, which triggers the mechanism. This method is defined
at the Object class level so that it it can be accessed by all the objects in the system.

changed
"The dependencies are traversed and"
"each is sent the update message"
self dependencies notNil
ifTrue: [self dependencies do: [:obj | obj update]]

The dependent objects (or simply, the dependents) are informed, one by one, of the change by the update message which is sent to them. The method associated with the update message can be redefined for each dependency class, thereby ensuring that each has an autonomous and specific behavior.

The activate message for the KeyCmdImmediate simply informs its dependents of its activation.

activate
"Inform the dependents of the activation"
self changed

It is not necessary to define the update method for this class, since immediate keys, here STOP, depend on no other key.

The activate message for the KeyCmdSwitchable class must not only inform its dependents, it must also toggle its state.

activate
"Inform the dependents of the activation"
"and change to the active state"
self changed.
state <- true

For a switchable key, the update method is defined to deactivate the key if it is activated.

update
state ifTrue: [state <- false]

As will be seen for the implementation, the dependency mechanism allows dependency links between objects to be masked. One way to do this consists of sending the changed message to self, thereby avoiding explicit access to the dependencies.

In Smalltalk-80, the described mechanism is more complete [76]. In fact, the changed message can be given several arguments, such as the sender of the change. It is also possible to send a message other than update to the dependents. The broadcast: message, used with an argument instead of changed, sends that argument to the dependencies (changed ≡ broadcast:#update).
4.3.2 Handling dependencies

Smalltalk offers two ways to store and handle dependencies. First, a specific instance variable (dependents) is allocated to each object. This variable is defined in the Model class, an immediate subclass of the Object class. Adding a new dependency to an object is done through the addDependent method, defined in the Object class.

```
addDependent: anObject
  dependents add: anObject
```

An object's dependents are reached through the dependents message, defined in the Object class.

```
dependents
  ^dependents
```

This technique, implemented by the Model class, forces each object to allocate the necessary space for dependents, even if it is not necessary.

Second, the global variable Dependents can be used as a dictionary, to store all the dependents. In Smalltalk-80, a global variable is simply a class variable of the Object class. Using this possibility requires another definition for addDependent.

```
addDependent: anObject
  "If the object did not yet have any dependents,"
  "then an entry is allocated for it in the dictionary"
  ifTrue: [Dependents
    at: self
    put: OrderedCollection new].
  (Dependents at: self) add: anObject
```

The dependents are accessed indirectly, through the dictionary.

```
dependents
  ^Dependents at: self
```

This technique ensures that space is allocated only for those objects that have dependents. On the other hand, the garbage-collector cannot automatically recover the space occupied by abandoned objects, since they are still referenced in the dictionary. Therefore, these objects must be explicitly dereferenced in the Dependents dictionary, using the release method defined for the Object class.

4.4 Model–View–Controller (MVC) systems

To free the user and developer from doing all sorts of tasks (editing, compiling, browsing, maintenance), each with its own interface, the Smalltalk designers integrated all these tasks into a windowing system [180]. Each task has its own window.
To effect a new task, a new window is opened, and to take up an interrupted task, one returns to the appropriate window.

These tools, including graphical editors, text editors, debuggers, browsers (to examine and modify Smalltalk classes) and window managers, are an integral part of the system: they are classes of the Smalltalk environment. Since the definitions for these classes are freely available, a user can easily modify the work environment or can develop interactive applications that reuse and enrich the predefined environment. Furthermore, shifting from one application to another does not bother the user, since all the interfaces are similar. This application homogeneity can also be found in MacIntosh applications [50], as well as in application generators.

Therefore, the Smalltalk environment itself exemplifies how to use the object-oriented model for the creation of user interfaces, since Smalltalk objects are being viewed by other Smalltalk objects. The set of interactive applications, written in Smalltalk, is built using the Model–View–Controller scheme (see Figure 4.6), where the model represents the structure and the behaviors to be represented and with which one works; the view is the output interface, the external representation of the structure and the visualization of the behaviors; and the controller is the input interface, which controls the interaction between the model and the view.

![Figure 4.6 The MVC trilogy](image)

This elegant scheme is so pervasive that the literature includes references to MVC systems [91, 127], to MVC principles [123] and to MVC methodologies [18]. Other systems offer similar representations, such as the Aida library [61] of the LeLisp language [45].

### 4.4.1 The model

Since an application is to be decomposed into three distinct but interdependent components, it is natural to begin by prototyping the structural and functional aspects of the application, i.e. the model, separately from any graphical interface that might improve the user-friendliness. The graphical interface can later be added to the model.

For the calculator, there are three classes of models: the central unit (class CalculatorPostfix), one of whose instances serves as main model; the screen
(class `Screen`), one of whose instances serves as graphical representation of the
screen; and the keys (classes `KeyNum`, `KeyFuncSpecial`, `KeyFuncUnary` and `KeyFuncBinary`), whose instances serve as model for the graphical representation of
the keys.

The MVC scheme makes apparent the dependencies between a model and its
views. The model must inform its representations of all changes resulting from
various manipulations.

For the dependency links to be handled implicitly, i.e. using an instance vari-
able, the classes whose instances serve as models in an MVC system are defined as
subclasses of the `Model` class (see Table 4.2).

<table>
<thead>
<tr>
<th>Class:</th>
<th>Class:</th>
<th>Class:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CalculatorPostfix</code></td>
<td><code>Screen</code></td>
<td><code>Key</code></td>
</tr>
<tr>
<td>Superclass:</td>
<td>Superclass:</td>
<td>Superclass:</td>
</tr>
<tr>
<td><code>Model</code></td>
<td><code>Model</code></td>
<td><code>Model</code></td>
</tr>
<tr>
<td>Attributes:</td>
<td>Attributes:</td>
<td>Attributes:</td>
</tr>
<tr>
<td><code>accumulator</code></td>
<td><code>buffer</code></td>
<td><code>label</code></td>
</tr>
<tr>
<td><code>screen</code></td>
<td><code>replace</code></td>
<td></td>
</tr>
</tbody>
</table>

The `Model` class is an abstract subclass of the `Object` class; it implements and
handles the dependency mechanism using an instance variable called `dependents`,
which contains the collection of the model's dependents, here the views that repre-
sent it. By making `Model` an immediate subclass of `Object`, the instance variable
`dependents` is not available to all objects. There are therefore two possible ways
of handling dependencies: (1) using a global variable for subclasses of the `Object`
class and (2) using an instance variable for subclasses of the `Model` class.

4.4.2 The view

After having tested and validated the internal behavior of the object-oriented pro-
gram, the developer adds to this structure the input/output interfaces for the
application.

This approach ensures the separation between internal representation (structural
and functional) aspects of a program and external representation (visualization,
user-friendliness) aspects (see Figure 4.7).

The MVC breakdown considers an input/output interface as a visualization and
access filter for the application's internal representation. There can clearly be
several visualization filters, i.e. different views of the same model, and several
access filters, i.e. different ways of accessing the same model, for example, using
text, graphics and menus.
So, for an MVC system, the main window is decomposed into as many subviews as there are views of the same model. Each of these subviews is part of an MVC subsystem whose submodel is the main model or a part thereof. A hierarchical organization for the different views and subviews can be defined (see Figure 4.8).
A view is defined in Smalltalk-80 as an instance of the View class or of one of its subclasses. The variables of a View instance describe the generic structure of a view: the represented model, the associated controller, the superview (if applicable), the subviews, the dimensions (using the screen’s or the superview’s coordinates), the thickness and color of the frame and the background color. The basic behaviors of a view include handling subviews, placing objects in a view and accessing the different coordinate systems. When an object is placed in a view, the given coordinates can refer to those of the view, the superview or of the screen. The WindowingTransformation class handles coordinate system shifts, including translation and scaling.

The View class contains numerous subclasses, used to represent different kinds of objects, for, among others, text (DisplayTextView, TextView, FillInTheBlankView), item lists (SelectionInListView), buttons (SwitchView), icons (FormView, IconView) and menus (FormMenuView).

4.4.3 The controller

The controller is an object that handles events coming from, say, the keyboard or the mouse. For each view, there is a specific controller, which allows events to be handled in a contextual manner, depending on the associated view. The general mechanism of a view controller, implemented through the Controller class, is presented here.

Active/inactive controller
At any given moment, the Smalltalk-80 development environment shows some open windows, each of which corresponds to some task (e.g., editing, maintenance or working window). One of these tasks is active and is linked to the running controller, while the others are waiting. At each moment, there is one running controller, for the active task, and a number of controllers waiting to be activated (see Figure 4.9).

More precisely, the active controller is the controller linked to the active subview of the active window. The list of active and inactive controllers is handled by an instance of the ControllerManager class, referenced through the global variable ScheduledControllers.

As soon as a controller finishes its task, one of the system processes (class Process) consults the list of potential controllers and activates the next controller. The controller manager sends the isControlWanted message to each of the potential controllers; the first to respond in the affirmative is immediately activated with the startUp message and enters its top-level.

Normally, a controller responds in the affirmative when a mouse button has been pressed inside the associated view. It is also possible to locally redefine the method associated with the isControlWanted message so that the control is requested
Figure 4.9 Passing control between tasks

when a certain delay has ended or if an independent process requires new data to continue its computations.

Controller activation cycle
The activation cycle of a controller (see Figure 4.10) begins with the startUp message.

```
startUp
  self controlInitialize.
  self controlLoop.
  self controlTerminate

controlLoop
  [self isControlActive]
  whileTrue:
    [Processor yield.
     self controlActivity]`

The cycle is initialized by the controlInitialize message, which might, for example, refresh the view (redraw its contents) and highlight its title. The default method defined in the Controller class does nothing.

The real cycle then begins. The isControlActive test is sent to the current controller. So long as it remains true, the controller continues to run. The default
method is to ensure that the mouse still points to the same view and that the blue button has not been pressed (see Figure 4.11). Of course, subclasses might want to redefine this method to take into account other constraints. For example, a game might decide whether a player can continue or not.

The controlActivity message runs the appropriate task. For example, a menu can be activated by pushing a button (MouseMenuController), keys struck on the keyboard can be received (ParagraphEditor), or objects can be moved in the view. When the controller decides to pursue the task no longer, once the isControlActive test is no longer satisfied, the activation cycle is terminated by sending the controlTerminate message. Normally, this message restores a state that existed prior to the controller's activation. For example, the view's title might be restored.
to normal (StandardSystemController) or the controller might be removed from
the set of activable controllers (FillInTheBlankController).

The numerous subclasses of the Controller class allow different events and
views to be handled, such as keyboard events (TextController, FillInTheBlank-
Controller), mouse events (MouseMenuController), or button or item selection
(SwitchController, SelectionInListController).

4.4.4 Class creation as application generator

A developer can instantiate existing view or controller classes to design an ap-
lication interface or can inherit the basic mechanisms to generate new kinds of
controllers (subclasses of Controller) or views (subclasses of View).

Therefore, defining a new controller consists of redefining the methods control-
Initialize, isControlActive, controlActivity and controlTerminate in the
new controller subclass. The Controller class can be seen as the application
skeleton of an event handler. A subclass of Controller is therefore an application
generator, similar to the ones described in Section 2.3.2: the developer completes
a prefabricated scheme by filling in the missing methods.

Furthermore, defining a new graphical representation often consists of designing
a new window with several views, each an instance of an existing class and asso-
ciated with a particular kind of controller. In this case, the classes are seen as
toolboxes, i.e. generic object boxes.

4.4.5 Three examples of views in the Smalltalk-80 system

Three Smalltalk views are presented here. Each is used to build the graphical
representation of the calculator.

StandardSystemView
An instance of the StandardSystemView class, along with an instance of the con-
troller class StandardSystemController, has an interface allowing the user to
manipulate the view and its subviews: change of size or of location, collapsing
(forming an icon, instance of class IconView), expanding (re-opening an icon) or
closing. These views have a title in the upper left-hand corner, indicating what
is going on inside the view and its subviews, for example, Workspace, System
Browser, System Transcript or File List.

Because all these functions are offered, instances of StandardSystemView gen-
erally serve as the root view of the view hierarchy in an MVC system. This will be
the case for the calculator. Views appear on the screen as in Figure 4.12.

When a view is being created, the designer must at least indicate the associated
model, the title and the minimum size; in addition, the background color and
the thickness of the frame may also be specified. The model allows the view to indicate a dependency link between the model and its new view. The title is simply a character string. Finally, the minimum size is a coordinate pair handled by the controller in such a way that the user can never reduce the view’s size to below the minimum. The instantiation-initialization appears as follows:

```
StandardSystemView
model: aModel "Model associated with the view"
label: 'Activity Title'
minimumSize: aSize
```

**TextView**

Instances of class TextView are used to show and to handle text. The default controller for these views is an instance of class TextController, which handles the basic text editor functions, including insertion, deletion, perusal and selection.

The subclasses of TextView allow specialized editor functions. For example, the text editor becomes a Smalltalk code editor in subclass CodeView. Among the extra functions related to code are the compilation or the pretty-printing of the code in the view.

When instances of class TextView are created, the designer must provide the model associated with the view, as well as three messages to be sent to the model, one to gain access to the text to be displayed, one to send it a new text that has been typed and validated by the user, and one to retrieve the view’s menu when the yellow mouse button is pressed. Of course, the designer must also define the methods associated with those three messages. The following method, used by the calculator’s screen, creates such a view.

```
TextView
on: aModel "Model associated with the view"
aspect: aspectMsg "Access to the model"
change: changeMsg "Update the model"
menu: menuMsg "Access to the menu"
```

*Functional arguments*, here selectors, are often used in object-oriented programming. This technique allows the explicit designation of the privileged *entry points*
of the model into its graphical interface, thereby encapsulating access to the latter.

A similar form of programming is used for views of lists (SelectionInListView). These views visualize a list of items, for example the list of a class’s methods. Creating an instance of this class is done by providing the messages to be sent to the model: one to obtain the list of items to be displayed, one to inform the selected item and one to activate the view’s menu when the yellow button is pressed.

SwitchView
Instances of class SwitchView are labeled views, allowing switchable models, i.e. objects with two possible states, active or inactive, selected or unselected, etc.

The default controller is an instance of the SwitchController class. It is activated when the user clicks on the red mouse button within the associated view; the resulting activity is to toggle the associated model.

To create an instance of SwitchView, the user must provide the switchable model associated with the view and the controller, the view’s label and two messages that can be sent to the model: one to retrieve the model’s state and one to inform it that it has been selected.

The model’s entry points are defined using functional arguments.

    aSwitchView <- SwitchView new model: aModelSwitchable.
    aSwitchView label: 'my label' asDisplayText.
    aSwitchView selector: #isActive.
    aSwitchView controller selector: #activate.

Should either of the selector: messages require arguments, these can be provided by sending the arguments: aTable message, where aTable is the list of actual parameters.

These views are generally used to describe toggle switches, called radio buttons by some designers of human–machine interfaces [36, 50]. Despite the fact that the calculator keys are not in fact switchable, they are visualized on the screen by instances of SwitchView. In this case, activating the associated controller restarts the activity associated with the key.

View hierarchy
This hierarchy is built as different views and subviews are defined. The designer does this by informing the view of additional subviews. Normally, the addSubview:in:borderWidth: message is used, with arguments for the subview, its position in the superview and the width of the edge of the border of the view:

    aView addSubview: aNewSubview
    in: aRectangle
    borderWidth: aThickness

The user can also position a subview with respect to another already extant subview, using addSubview:toLeftOf: or addSubview:toRightOf:...
4.5  Graphical interface for the calculator

It is now time to present the interface between the calculator and the different views that have been presented above.

4.5.1  The CalculatorView class

An instance of the CalculatorView class serves as root view of the hierarchy of views of the MVC system for the calculator model. To ensure that the view can be manipulated (resized, moved or iconized) just like any other Smalltalk-80 window, it is defined as a subclass of StandardSystemView.

Class : CalculatorView
Superclass : StandardSystemView

To create a view for the calculator, a class method is used. In Smalltalk, a class is an object, just like any other object. Each class is an instance of another class, called its metaclass. In former versions of Smalltalk, all classes were instances of a single metaclass. To make things easier, in Smalltalk-80, each class is an instance of its own metaclass, and each of these metaclasses is an instance of the class MetaClass.

A class method for CalculatorView is therefore a method defined at the level of CalculatorView's class, i.e. its metaclass, so that in fact only CalculatorView may respond to such methods, since it is in fact the only instance.

The class method openOn:, which actually builds an MVC system for the calculator, takes as argument the Central Unit, the MVC system's main model.

openOn: aCalculator
   "Create the root view of the system"
   "using method model:label:minimumSize:"
   "defined in superclass StandardSystemView"
(s self model: aCalculator
   label: aCalculator class name asString
   minimumSize: 50@100)
   "Add the screen"
   addScreenViewOn: aCalculator screen;
   "Add the numeric keys"
   addKeysNumViewFor: aCalculator screen;
   "Add the function keys"
   addKeysFuncViewFor: aCalculator;
   "Activate the associated controller"
   controller open

The openOn: method uses the model:label:minimumSize: method, defined in the StandardSystemView superclass. It creates the view and initializes the attributes, for example for an instance of the CalculatorPostfix class.
The openOn: method also uses the addScreenViewOn:, addKeysNumViewFor: and addKeysFuncViewFor: methods, all defined below, to add, respectively, the screen, the numeric keys and the function keys.

Finally, the actual activation of the controller, instance of the StandardSystemController class, takes place by sending it the open message.

The addScreenViewOn: method adds a TextView to model the calculator screen. The TextView is initialized with the model, which is an instance of class Screen, and a message to send to the model to retrieve the text, here in buffer.

```plaintext
addScreenViewOn: aScreen
"Create a screen and its view"
self
addSubview: (TextView on: aScreen aspect: #buffer
"Message to access the screen buffer"
"The screen cannot be modified directly"
"and there is no menu"
change: nil menu: nil)
in: (0@0 extent: 1@0.2)
borderWidth: 2
```

No message is provided for a possible update of the model by the view or by a menu associated with the view. The calculator's view serves only as an output interface, for viewing.

When the TextView is added as subview of the receiving CalculatorView, it takes up the upper part of the view. This is done through the specification

```
Rectangle: (0@0 extent: 1@0.2)
```

This Rectangle indicates the size of the subview, expressed in relative terms with respect to the root view. Point 0@0 indicates that the upper left-hand corner is the origin of the superview. Point 1@0.2 indicates that the subview has the same width and one fifth of the height of the superview (see Figure 4.13).

![Figure 4.13 Screen dimensions](image)

For TextView to be able to retrieve the buffer contents, a buffer method for Screen must be added. It transforms the string into an instance of class Text before returning the contents.
The addKeysNumViewFor: method creates all the numeric keys (instances of class KeyNum) needed by the calculator, providing a label and a pointer to the screen, required during their activation.

addKeysNumViewFor: aScreen
"List of the numeric keys to create"
"Each key is designated by a triple"
"(label, xPos, yPos)"
#( {'7' 0 0.36) ('8' 0.25 0.36) ('9' 0.5 0.36)
('4' 0 0.52) ('5' 0.25 0.52) ('6' 0.5 0.52)
('1' 0 0.68) ('2' 0.25 0.68) ('3' 0.5 0.68)
('0' 0 0.84) ('.' 0.25 0.84) )
do: [:triplet |
  self addKeyViewOn: (KeyNum new
    label: (triplet at: 1)
    screen: aScreen)
  in: ((triplet at: 2) @ (triplet at: 3)
    extent: 0.25@0.16)]

Each KeyNum is provided a particular view, created using addKeyViewOn:in:.

addKeyViewOn: aKey in: area
"Create and add the SwitchView to the key"
"passed as argument"
self
  addSubview: (SwitchView new
    model: aKey;
    label: aKey label asParagraph;
    selector: #isActive;
    controller selector: #activate)
in: area
borderwidth: 1

This method adds a new subview, representing a key, to the CalculatorView receiver. To initialize this subview, an instance ofSubview, the user must send as arguments the key, its label, and two messages that are sent to the model, one that replies with the model’s state, here the isActive method defined below, and one to signal the view’s activation, here the activate message.

To be coherent with all the classes that have been defined up to now, the isActive method must be defined for all keys in the environment. However, the keys in general are not toggle switches. In this case, a key should return the value false by default, hence the definition for isActive in class Key.
isActive
- false

For keys that are toggle switches (instances of KeyCmdSwitchable), they can be in an active or an inactive state. So their isActive method returns the current value of the state.

isActive
- state

The addKeysFuncViewFor: method is similar to addKeysNumViewFor:; the only difference is that it creates function keys.

addKeysFuncViewFor: aCalculator
"List of function keys to create"
"A key is a triple (label, xPos, yPos)"
"Binary functions"
#( ('+' #+# 0 0.2) ('-' #- 0.25 0.2)
  ('*' #* 0.5 0.2) ('/' #/ 0.75 0.2) )
do: [:triplet |
  self addKeyViewOn: (KeyFuncBinary new
    label: (triplet at: 1);
    function: (triplet at: 2);
    calculator: aCalculator)
  in: ((triplet at: 3) @ (triplet at: 4)
    extent: 0.25@0.16)].

"Unary functions"
#( ('fac' #factorial 0.75 0.36)
  ('fib' #fibonacci 0.75 0.52) )
do: [:triplet |
  self addKeyViewOn: (KeyFuncUnary new
    label: (triplet at: 1);
    function: (triplet at: 2);
    calculator: aCalculator)
  in: ((triplet at: 3) @ (triplet at: 4)
    extent: 0.25@0.16)].

"Special ENTER function"
self addKeyViewOn: (KeyFuncSpecial new
  label: '=';
  function: #enter;
  calculator: aCalculator)
in: (0.75@0.68 extent: 0.25@0.32).

"Special C function"
self addKeyViewOn: (KeyFuncSpecial new
  label: 'C';
4.5.2 Modifying the original model

So that the model can take into account the fact that it is linked to a graphical representation that must be informed of any changes, the original code must be modified. The only affected class is Screen, since only the screen need inform its dependent of changes; this is not the case for the keys, which undergo no visible changes, nor for the accumulator, which is not represented in the MVC system. Each time the buffer is modified, the screen informs its dependency of the change that it must make to the buffer. From a programming point of view, as soon as the buffer instance variable is changed, the dependency mechanism must be launched to update its graphical representation.

The two methods that modify the buffer are input: and value:. Here is their new form.

```plaintext
input: aDigit
    "Should the buffer be recovered?"
    replace
        "Send the number to the calculator"
        "and overwrite the buffer"
    ifTrue: [calculator
        accumulator: self value.
        buffer <- aDigit.
        recover <- false]
        "Otherwise append the digit to the buffer"
    ifFalse: [buffer <- buffer,aDigit]
    "Modification for graphical interfaces"
    self changed: #buffer

value: aNumber
    "Put the numeric value in the buffer"
    self validate.
    buffer <- aNumber asString.
    "Modification for graphical interfaces"
    self changed: #buffer
```

Should the model have several graphical dependencies, the #buffer argument for the changed: method, identifies, if need be, the view that should take the change into account.
This mechanism can be automatically handled in languages such as loops [24], lore [15] and VLISP [192], which can attach active values to instance or class variables. These active values can describe daemons that are triggered automatically, as soon as the value of a variable has been read or modified.

4.5.3 Creating a graphical calculator

Now the postfix calculator prototype can be created.

```
CalculatorView
openOn: CalculatorPostfix new
```

The result is to create a view like the one in Figure 4.14. From now on, the user no longer needs to simulate pressing keys by sending the activate message to the appropriate key. Instead, he or she clicks with the mouse button on its graphical representation, thereby improving the calculator's ease of use.

![Calculator interface](image)

**Figure 4.14 Calculator interface**

4.5.4 The hidden function $x^n$

The calculator, as built, actually contains hidden functions, which are not directly available from a key. One of the most common hidden functions is the exponentiation operator $x^n$, where $x$ is the value in the accumulator. This hidden function can be effected by using the calculator’s multiplication (*) operation.

Consider a postfix calculator. Input the digit 3 and validate it with the key enter. Input once again the digit 3, in order to do the multiplication $3 \times 3$. Pressing once
on the * key yields 9 (\(= 3 \times 3 = 3^2\)) on the screen. Pressing the * key again yields 27 (\(= 3 \times 3^2 = 3^3\)), again yields 81 (\(= 3^4\)), 243 (\(= 3^5\)), etc.

Doing this sequence of operations on the postfix calculator prototype will give the same results. By building the prototype, such hidden functions can be discovered and explained. In this case, when the * key is pressed, the calculator multiplies the accumulator's contents by the number on the screen, and the result is then directly stored into the screen buffer. When the * key is repressed, the accumulator value has not changed, it is still 3. The resulting function is raising to the power \(n\), where \(n\) is the number of times the * key has been pressed.

Another hidden function is the multiplication table of a number, by simply replacing the * key by the + key.

4.6 Creating an infix calculator

One of the essential functions of prototyping is to be able to quickly develop several prototype software systems, in order to provide several points of view [110, 111] about the product being developed. This notion of multiple points of view is elaborated in the second part of this book, which deals with the MPVC approach. Nevertheless, as an example, the difference between the already described postfix calculator and an infix calculator is given below. Building a second prototype will allow a comparison of the two kinds of calculator, showing the commonalities and the differences.

The screen and key behavior should be independent of the kind of calculator that is being used. Hence those parts of the prototype, i.e. the Screen class and the various key classes, should not be changed in any way. However, the Central Unit of the calculator should be changed to create an infix calculator, i.e. so that it can input the operations in an infix manner. Below, each characteristic of the CalculatorPostfix class is examined in order to determine what has to be changed.

4.6.1 Structural differences

The CU of an infix calculator has an accumulator and a connection to its screen, just like the postfix calculator. It also needs a register to memorize the function corresponding to the last pressed binary key: when such a key is activated in infix syntax, the second operand has yet to be input. To ensure that the function label is not lost, the CU memorizes it in a new register.

Given these remarks, an abstract Calculator class is created, vertically factoring the common characteristics of the two CU classes, the accumulator and the screen.
Class: Calculator
Superclass: Model
Attributes: accumulator, screen

The `CalculatorPostfix` class is then redefined as a subclass of `Calculator`, and so inherits its characteristics, including its instance variables.

Class: `CalculatorPostfix`
Superclass: `Calculator`

The `CalculatorInfix` class is also a subclass of `Calculator`, but with a new instance variable `function`, a register for storing the binary function label.

Class: `CalculatorInfix`
Superclass: `Calculator`
Attributes: `function`

From a purely programming point of view, the class `CalculatorInfix` could have been defined as a subclass of `CalculatorPostfix`, and `Calculator` would then not have had to be defined. But it is better to consider the inheritance relation as an *ako* (a kind of) relation in the semantic network formalism, i.e. to consider that a class `SC`, subclass of class `C`, is a kind of instance of class `C`. Clearly, a `CalculatorInfix` is not a kind of `CalculatorPostfix`. However, they are both kinds of `Calculator`.

### 4.6.2 Behavioral differences

To deal with the behavioral differences between the two calculators, each of the methods of class `CalculatorPostfix` is studied.

The `execOperationUnary`: method applies a unary arithmetic function to the number in the screen. In the infix calculator, the unary functions use a postfix syntax, i.e. the functions are applied after the operand has been read and computed. Hence the `execOperationUnary`: method is common to both prototypes, so it should be removed from the `CalculatorPostfix` class and added to the `Calculator` class. Its definition is repeated here.

```ruby
execOperationUnary: operator
  "Get the number on the screen"
  "Apply the function and"
  "return the value to the screen"
  screen value: (screen value perform: operator)
```

The `execOperationBinary`: method applies a binary function to two operands. Unlike for the postfix calculator, the application in an infix calculator is not immediate. Therefore a new `execOperationBinary`: method must be defined for the `CalculatorInfix` class.
execOperationBinary: operator
    "Is a function memorized? If so, apply it"
    "to the accumulator with the buffer as argument"
    "Store the result in the screen buffer"
function notNil
ifTrue: [screen value: (accumulator perform: function
                           with: screen value)]
ifFalse: [screen validate].
"Memorize the operator"
function <- operator

A check must be made to determine if a function has already been stored in the
CU's function register. If it is, then the accumulator already contains an input
or computed value and the user has just activated a binary function to be applied
to the number and the number to be input. Therefore the current operation must
be applied and displayed on the screen. If no function has been memorized, then
the value that had been input should be validated. Then the new binary function
should be memorized in the function register.

The init initialization method differs from that of the postfix calculator, since it
must initialize the function instance variable. To accentuate the fact that it is an
addition, the init method for CalculatorPostfix is transferred to its superclass
Calculator. This transfer is appropriate since all calculators will have to do this.
Its definition is repeated here.

    init
    accumulator <- 0.
    screen value: accumulator

For the CalculatorInfix class, the init method first calls the init method
for its superclass, then it initializes the function variable.

    init
    super init.
    function <- nil

    Last, but not least, is the enter method. The = key in the infix calculator
is the closest to the enter key in the postfix calculator, although the behaviors
are quite different. When the user types the = key on an infix calculator, the
previously input binary arithmetic function should be applied to the operands, the
accumulator and the number in the screen.

    enter
    "Is a function memorized? If so, apply it"
    "to the accumulator with the buffer as argument"
    "Store the result in the screen buffer"
function notNil
ifTrue: [screen value: (accumulator perform: function with: screen value)]
ifFalse: [screen validate].
function <- nil

Note that the execOperationBinary: method of the CalculatorInfix method contains the enter method for the same class. When a user has typed a binary operator and wishes to type a second to compute further results, the = key is not retyped to compute the intermediate results. The next operator is immediately input, and it is up to the calculator to implement the correct operation, i.e. the calculator must implicitly effect the = operation. Hence the new definition for execOperationBinary:

execOperationBinary: operator
  "Simulate the = key"
  self enter.
  "Memorize the function"
  function <- operator

4.6.3 The infix calculator

Creating the infix calculator is done in the same way as for the postfix calculator.

    CalculatorView openOn: CalculatorInfix new

The result is a calculator that, apart from the view's title and its behavior, appears to be the same as the postfix calculator.
Chapter 5

Solving problems through prototyping

The first chapter presented prototyping from an epistemological perspective, defining the concepts of prototype and model. The following chapters placed prototyping in a programming context, and object-oriented languages and programming were shown to form an appropriate programming environment. The overview is completed here with the cognitive context in which prototyping occurs.

Prototyping takes place between two actors: the designer and the expert. This dialog and the interaction of complementary knowledge are great sources of ambiguity and of misunderstanding. Prototyping aims to help model correctly the field under study.

The psychological context for programming thought processes [192] that take place during prototyping is presented in the rest of this chapter.

5.1 Psychology and programming

The first applications of psychology to programming were performance evaluations of computer tools [167], using criteria such as learnability for a novice, designability and understandability. Since then, psychologists have studied the very act of programming, as well the conceptual models used during this activity, with the objective of adapting computer tools to their users. Programming is studied in the areas of cognitive ergonomics [171], cognitive psychology [3], psycholinguistics [79, 188] and industrial and organizational psychology [122].

5.1.1 Cognitive ergonomics

Cognitive ergonomics studies the existing links between tools and their human users. The links are not made explicit, rather cognitive ergonomics tries to understand how these tools affect human behavior.
In the computer context, the tools are software and hardware, including programming languages, formalisms, graphical interfaces and input/output peripherals (mouse, light pen, touch screen). Ergonomics attempts to understand and to describe, for example, how a graphical representation helps the rapid and correct understanding of a concept.

5.1.2 Cognitive psychology

Cognitive psychology studies how designers perceive, organize, handle and memorize information, as well as the nature of the different cognitive abilities and how they differ from one designer to another.

Current psychological theories propose three or four different levels of memory: sensory, short-term, long-term and, possibly, intermediate. This classification has no known relation to neurophysiological geography; it is the result of conceptualization of experimental data [4, 47, 103, 138].

In the context of computer programming, cognitive psychology addresses the following questions:

- How does a programmer perceive the tools that he or she uses, i.e. what is expected of a development environment [192] and how is it interpreted, given what the programmer already knows?
- How can a programmer best use the information in his or her possession, given the a priori limited short-term memory capacities [129]?
- How can programming tools be used to take best advantage of short-term memory [190]?
- How can the programmer organize and best use the information stored in long-term memory in order to solve a new problem?
- How can the learning of programming best be improved [28, 107, 192]?
- What are the necessary cognitive abilities to fulfil different programming tasks [89]?

5.1.3 Psycholinguistics

Psycholinguistics is a field that grew out of the interaction of linguistics and cognitive psychology. It first examined the effect that a language – computer or not – can have on cognitive development and on the ability to handle information [193]. Hence the first programming language that a student learns has important repercussions on his or her approach to problem solving using computers [141]. For example, solving problems in LISP does not resemble solving problems in Basic or COBOL.
5.1.4 Industrial and organizational psychology

This field studies human actions in their professional environment. It uses aspects of psychosociology, personality theory, management theory and knowledge of organizations.

The industrial psychologist specialized in computer science must devise, with managerial help, selection tests for programmers in order to create an efficient work team, evaluate the performance of a programming team, guide programmers in their career choices, identify training needs, pinpoint trouble zones – relational or organizational – within a work team, identify the factors that motivate personnel, including salary and benefits, and create a work environment that improves productivity.

The industrial psychologist’s work consists mainly of analyzing the relations between the different workers, as well as between workers and their work environment.

5.1.5 Psychology of programming

Putting together these four psychology fields, as applied to programming gives Table 5.1, which is the set of different fields in the psychology of programming [54], also called cognitive ergonomics of programming [60]. Prototyping activities take place in the boxes where the text is in italics. The remainder of this chapter focuses on the identification and the formalization of problems, as well as on the interaction between the designer and the expert; these appear in bold italics in the text.

Table 5.1 Prototyping in the psychology of programming

<table>
<thead>
<tr>
<th>Cognitive ergonomics</th>
<th>Specification tools and methods</th>
<th>Design tools and methods</th>
<th>Finetuning and testing tools</th>
<th>Coding tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cognitive psychology</td>
<td>Complete understanding of the problem’s scope</td>
<td>Problem’s formalization</td>
<td>Error search strategy and understanding of programs</td>
<td>Translation of the design into procedural terms</td>
</tr>
<tr>
<td>Psycholinguistics</td>
<td>Specification languages</td>
<td>Design languages</td>
<td>Error messages and software documentation</td>
<td>Programming languages</td>
</tr>
<tr>
<td>Industrial and organization psychology</td>
<td>Designer-client interaction</td>
<td>Designer-expert interaction</td>
<td>Inter-team dynamics</td>
<td>Team’s performance</td>
</tr>
</tbody>
</table>
5.2 Problem solving

The term *problem to be solved* can only refer to situations for which the subject does not have a set of standard responses or precise instructions. It is not the complexity of the subject that constitutes the problem but, rather, the fact that it is new for the subject [171, p. 58].

Programming consists of *problem solving* [112, 137, 189, 192]. However, up to now programming has not been differentiated from modeling. So, to the extent that a *software prototype* is a *software model*, prototyping can be seen as building problem solution sketches, where the sketches are the prototypes.

5.2.1 The *problem* in prototyping

For the discussion below, Harald Wertz’s definition of problem [192, pp. 29–30] is assumed.

We are confronted by a *problem* if we have an initial state $\alpha$ that we wish to transform into a final state $\beta$, not knowing how to achieve this transformation immediately.

In other words, a *problem* consists of searching for a sequence of *operators* [in a programming language, either primitive operators or the combination of primitive operators] that transform an initial state $\alpha$ into a desired final state $\beta$.

Wertz also notes that solving a problem requires existing knowledge of the operators to be used and that it is not possible to speak of a problem in the programming context without it being ‘well-defined’.

Prototyping does not seem to be compatible with this remark. Its purpose is to define and formalize the underlying problems in the preliminary specifications, which certainly gives the impression that it is not a ‘well-defined’ problem. This is in fact not the case: the initial and final states for prototyping do not correspond to the initial and final states for programming.

Prototyping involves two actors: the designer who implements prototypes and the expert who validates them, so there are two points of view (see Figure 5.1).

The designer must reify the expert’s domain, then develop a set of prototypes that formalize the requirements for the software. As a result, the initial state of the designer’s problem corresponds to the expert’s precise understanding of the requirements and of the field of study; the final state is a series of software prototypes, representing the designer’s perception and interpretation of the field under study, which allow the visualization, manipulation and testing of the formalism
developed by the designer with the expert's help. The designer must therefore suppose that at each moment, she has a perfect understanding of the problem at hand. Hence, for the designer, the problem is well-defined. The difference with the initial state for programming is that this knowledge evolves over time, with the expert's different validations.

The expert, on the other hand, must formalize the field of study and then validate the different prototypes developed by the designer. So the initial state of the expert's problem is the complete understanding of the field of study and the final state is a software model that allows the formalization and solution of problems that are particular to the domain. It is clear that at each instant the expert has a precise idea of his problem, in fact that is how the validation can actually take place.

Formalizing the field of study is not always easy:

Formalizing the knowledge and the strategies used by professionals at work poses problems. The assertion that professionals are capable of making explicit their knowledge in ordinary situations is naive. There is in fact, given our understanding from the cognitive sciences, a large gap between what subjects say they do at work and what they actually do when at work [60, p. 173].

The experts that designers appreciate the most are generally those who are predisposed to computer formalisms and schemata, i.e. they have a knowledge of the basic vocabulary and are able to translate their analysis strategies into functional and organizational organizations. Nevertheless, in the future, prototyping environments should assist an expert, even with no formal background, to formalize and to conceptualize his domain of study, i.e. they should take the first phase completely into account.
To solve the prototyping problem, the designer can use the operators in the prototyper development environment. Thanks to these operators, she can implement a formalism that is well-suited to the domain and build the operators needed by the expert to visualize, manipulate, test and validate the formalism. The MPVC system, presented in the second half of this book, illustrates the ambiguity between manipulation operators and test operators, using the concept of point of view.

### 5.2.2 Solving problems

Most problems outside of primary or secondary school are not carefully defined in a formal, usually mathematical, language [112]. Most of the time they are ‘open’, i.e. partially written in ambiguous or subjective languages (natural, spoken, gestural, graphical, etc.).

Clearly, before solving problems raised in the requirements, the designer must, with the help of the expert, remove the ambiguities in that document. After that, the initial and final states of the problem can be formalized into what Wertz calls a *cognitive unit* [192].

Formalizing the informal specifications [196] means that the statement can be made more explicit, and redundancies, incongruities and extraneous information can be removed, as can ambiguities or implicitly added constraints that make a solution impossible.

The *nine-point* example illustrates the preceding remark about ‘implicitly added constraints that make a solution impossible’. The nine points in Figure 5.2 are to be connected by four straight lines.

![Figure 5.2 Nine points to be connected by four straight lines](image)

Most people who meet this problem for the first time introduce an extra constraint: they think that the points form a square and that the solution must lie within the square, thereby adding a condition that is not in the requirements and that makes the solution impossible. This constraint is due to the human visual system, which interprets the perceived reality [112] and which imposes a framework in which to find the solution [186]. In fact, the solution to this problem lies outside the fictive zone, containing the four points, that was implicitly imposed as a constraint (see Figure 5.3). If the mathematical constraint of points having no thickness is removed, then it is in fact possible to draw a solution using only
three straight lines (see Figure 5.4). Adams [1] presents other solutions for the nine-point problem using, for example, a well-chosen folding of a cylinder or of a sphere.

![Figure 5.3 Nine points connected by four straight lines](image)

![Figure 5.4 Nine 'points' connected by three straight lines](image)

Watzlawick presents problem resolution as a state-changing activity, of which there are two kinds.

A type-1 change takes place within the system containing the problem, where the system itself remains invariant (Watzlawick here uses analogies to group theory). For example, when winter comes and the temperature drops, rooms must be heated and one must put on more clothes. As the temperature drops, more warmth and clothes are needed. In other words, a change takes place to restore the norm, be it to restore comfort or survival. Solving the problem consists of applying the inverse of what created the deviation (here, warmth as opposed to cold).

This kind of solution or change can only be applied in certain situations. In others, it can worsen the situation rather than bring about a solution.

Alcoholism is a serious social problem. Alcohol consumption must, therefore, be regulated. When this restriction no longer suffices, it is enforced to the hilt, i.e. prohibition is imposed. But prohibition is, as remedy, worse than the ill itself: alcoholism increases, clandestine industry sets itself up and sells low-quality products that are actually more dangerous, a police force specialized in tracking clandestine distributors is created and quickly becomes corrupted, etc. Since the problem gets worse and worse, prohibition is made more and more strict.
Surprisingly, more of the same does not lead closer to the desired solution, in fact the ‘solution’ aggravates the problem, itself becoming the worst of all problems: not only is there a high and unchanging level of alcoholism but, also, new alcoholics join the ranks and there is a dramatic increase in contraband, corruption and gang warfare.

... The observer of this kind of change feels as if he or she is looking at two sailors who are acting as counterweights on either side of a boat to ensure its stability: as one leans over, so must the other to compensate for the instability created by the first, who was trying to stabilize the boat. The boat itself would be stable without their acrobatic efforts to stabilize it. It is easy to see that to get out of such an absurd situation, one of them should do something that would appear at first to be unreasonable, namely to stabilize less and not more, which would force the partner to compensate in turn, unless he or she prefers to fall in the water. In time, they might find themselves in a comfortable situation in a stable boat [186, pp. 49–50, 54].

The latter example introduces the observer (not necessarily neutral [85]) who, being able to oversee the entire problem, can see the solution. The observer’s system allows a reframing of the problem, which leads to type-2 changes.

Type-2 changes involve a change of context, from one logical level to another. This change can be used to analyze or to modify the system containing the problem. The nine-point problem is a typical example of this kind of change. Here is another change of context.

During one of the numerous Parisian revolts of the nineteenth century, an officer received an order to clear the square by firing on the ‘rabble’. He gave his soldiers an order to take up position and to aim at the crowd. At that moment, everything became quiet, he drew his sword and yelled.

‘Ladies and gentlemen, I have just received an order to fire on the rabble. But since I can only see in front of me many honest and respectable citizens, I ask them to leave so that I can fire without risk at the rabble.’

The square was emptied in a few minutes [186, pp. 101–102].

Instead of responding to the crowd’s hostility with yet more hostility (type-1 change), which would have only aggravated the social tension, the officer changed the original framework (the crowd against the army) by pretending to be on the crowd’s side (second kind of change). This reframing allowed him to satisfy two parties, the crowd and his senior officers.

Here is another example [112] where the reframing is indispensable in dealing with an erroneous specification.

Statement: a cruel king places a young woman in jail because she refuses to marry him. After a year in jail without her changing her
mind, the king has her brought to the court in his castle and offers her a deal.

'I am going to pick up two stones, one black and one white, and hold them hidden in my hand. You will freely choose one of the two. If you draw the white one, you will be free; if, on the other hand, you draw the black one, you will marry me.'

The young woman accepts the deal with great apprehension. But it soon turns to panic when she sees that the king reaches down and picks up two black stones.

What can she do?

Solution: The young woman quickly grabs one of the stones and drops it immediately on the ground with the others.

'Pardon me, Sire, but the color of the other stone also decides my fate.'

The other is black, of course, and she is free.

In programming, finetuning tools can be used to observe and to intervene in a program's behavior, for such aspects as function-call traces, step-by-step evaluation or symbolic evaluation. These evaluation semantics [110] offer a reframing of the program's execution. These tools are implemented in the MPVC system as meta-interpretations of a program's execution.

5.2.3 Resolving prototyping problems

Returning to the programming context, it is clear that a programming environment includes tools for formalizing problems so that their solutions may be sketched. These tools should serve to clarify the problem's statement (initial and final states), to reduce ambiguities or implicitly added constraints, and to eliminate redundancies and incongruities. Therefore, they must allow one to work within the system itself as well as to change frameworks. Within the system, the designer must be able to model the field under study, its properties and its functionalities; the expert or user must be able to easily manipulate this domain. As for the reframing, better known in programming as a change of representation, these are covered by the MPVC system, with its notions of point of view and meta-point of view.

Research in knowledge representation and, consequently, in problem representation [112] shows that the search for a proper representation of the problem is an essential step in its resolution.

As an example, consider the four-knight problem in the 3 × 3 chess-board in Figure 5.5. In a minimum set of moves, the two white knights and the two black knights are to exchange their positions. Two knights cannot simultaneously be on the same square, and knights move one horizontal step and two vertical steps, or the other way around.
After playing on a real chess board, the first representation that comes to mind is a Cartesian representation of the knights on the board. The coordinates $(000)$ correspond, for example, to the center of the board. Hence the possible moves of the black knight in the upper left-hand corner of the board are:

$(-1 \ 0 \ -1) \leftrightarrow (1 \ 0 \ 0) \land (-1 \ 0 \ -1) \leftrightarrow (0 \ 0 \ 1)$.

This representation uses heavy notation and does not take into account the symmetry of the problem. Furthermore, it considers the position of each of the pieces on the board rather than its movement, which is, after all, the core of the problem. The physical layout of the board is of no consequence, only the connections between squares on the board created by the possible movements.

**Figure 5.6 Naming of squares**

So, each of the squares is named (see Figure 5.6), and the links between them must then be defined. A knight on square $A$ can move in one step to square $H$ or to square $F$. From $H$, the knight can move back to $A$ or go to $C$. Continuing, it can go to $D$, then to $I$, then $B$, then $G$ and finally back to $A$. Figure 5.7 illustrates the connections.

Square $E$ is unreachable from this graph, hence is not part of the problem. Using this new representation, it is now possible to reformulate the statement: 'Effect a minimum number of changes to pass from the initial state $\alpha$ to the final state $\beta$.' Note that a knight can, in a given step, move to only one of the neighboring squares from its current square.

The solution is now clear (see Figure 5.8): it suffices to have all of the pieces to make a half-turn around the ellipse, taking the knight on $C$ to $G$, the one on $A$
to $I$, on $G$ to $C$ and on $I$ to $A$. This rotation, which takes place in 4 steps per knight, or 16 steps in all, is minimal.

Changes of representation are often forms of *analogy* [132, 137]. Faced with a new object or concept, one attempts to represent it as something similar to an already known object or concept. For example, in an MPVC system, the control structure of a program is a directed graph, called an *ordering graph*.

Hence, we try to bring each new concept to something more ordinary: we are using analogies.

Suppose an alien architect has invented a radically new way to go from one room to another. This invention serves the normal functions of a door, but it has a form and mechanism so far outside our experience that to see it, we would never recognize it as a door, nor guess how to use it. All its physical details are wrong. It is not what we normally expect a door to be – a hinged, swinging, wooden slab set into a wall. No matter: just superimpose on its exterior some decoration, symbol, icon, token, word, or sign that can remind us of its use. Clothe it in a
rectangular shape, or add to it a push-plate lettered EXIT in red and white, and every visitor from the planet Earth will know, without a conscious thought, just what that pseudoportal's purpose is, and use it as though it were a door [132, p. 57].

This change of reference, similar to the previously cited reframing, means that the prototyping environment must have a large number of structural and functional concepts, thereby offering many representations (visual, audio or tactile, interpreted or not - the number will increase with multimedia applications) for possible analogies.

It is clear that a prototyping environment must allow an intelligent and efficient management of these representations and interpretations. Intelligent means that these representations can be mutually enriched by adding data, constraints or conditions [29, 33, 152].

The MVC methodology shows that it is possible to have several different graphical representations for the same internal representation. The second part of this book shows that it is also possible to design and to handle multiple interpretations, called points of view or evaluation semantics, for the same internal representation, with several different graphical representations.
Object-oriented methods have been used in industrial circles to develop object-oriented database management systems [62, 82], prototyping environments [12, 27, 66, 82, 110, 140, 184] and software engineering toolkits [136, 158].

The basic structural and functional characteristics of object-oriented methods and languages meet industrial needs, which include data and procedural encapsulation, application modularity, reusability and human–machine interfacing.

Furthermore, prototyping is the key to the quick solving of a software problem. There has been much research in the last decade, both in academia and in industry, into this software paradigm.

There are actually two main research directions. The first is the search for knowledge representation models that are adequate for this field (formalisms JSD [42], MERISE [176], WQN [145], PEEP [40], SADT [159], SDL [14] or Statecharts [84]). The second is the development of efficient tools for the conceptualization, manipulation and interpretation of the modeled knowledge base and software, using the above formalisms.

To illustrate these two directions, three examples of object-oriented programming environments, academic and industrial, are given below. The MPVC trilogy is shown to be compatible with these kinds of environment.

6.1 The DesignNet model

Many specialized object-oriented prototyping environments [82] offer a graphical interface that manipulates a very large database. These object-oriented databases initially contain items in the field under study. The user can augment them using aggregation and inheritance techniques. The DesignNet model [119], presented below, illustrates this kind of environment.
6.1.1 General presentation

During the software life cycle, from specification to coding, many different kinds of information are manipulated, including text, graphics, audio, video [195], electronic mail, source code, object code and binary executables.

In general, the dependency relationships between these different kinds of information are only approximately specified. For example, when the specifications of a particular function are changed, it is difficult to determine what documents must be redrafted, what program modules must be modified in turn, what functions, such as compilations and tests, must be reinvoked, what manual pages must be brought up to date and what persons should be responsible for the different tasks.

This kind of management information is often only available on paper, sometimes it only resides in the heads of a few people. Change orders are given orally to development or maintenance teams. For an important project, developed over an extended period of time, personnel can change and the management of a project can be threatened by the lack of indicators about the dependencies between the different kinds of information.

Most project management systems look at only one particular aspect of the software life cycle. For example, SCCS [156] and RCS [179] handle multiple versions of individual source code files, make under Unix [67] handles dependencies between a program's different modules, and SODOS [90] covers the definition and manipulation of documents.

The DesignNet system uses a object-oriented representation model and a graphics interface to completely describe and handle the software development process. It uses its own graphical formalism to describe the static organization of a project and Petri nets [13, 35, 168] to represent dependencies, parallelism between the activities, resources and products (all documents, programs and files) of a project.

6.1.2 Graphical representation

To describe an object, DesignNet uses three kinds of component: places, structural operators and transitions.

There are five kinds of place: projects, activities, resources, products and project status reports. Each of these represents a different kind of information necessary for the understanding of the software life cycle.

Structural operators allow places to be decomposed in a hierarchical manner. Information at different levels of detail can then be simply aggregated (similar hierarchy mechanisms can be found in the SADT formalism [159]).

Transitions, which generate the dependency links, define the relations between the different kinds of information.

Figure 6.1 is a DesignNet example of the specification/prototyping stage of a project. It states that
activity places needs analysis and prototype implementation, combined by the structural operator and, form the decomposed activity specify and prototype;
product places contract, specs, test plans, design docs and interface specs, combined by the structural operator and, constitute the software specs;
activity needs analysis is triggered when product place client contract is generated and when the resource places experts and analysts are present;
phase specs takes place once needs analysis has been validated;
and so on.

The places and transitions form a static description of this part of the project. Below it is called the **project management net**.

Describing the behavior of the project is done with tokens, as for Petri nets, which pass through the net, from place to place. Each transition creates a new token, which acquires information as it passes through the network. This kind of information might include temporal information to describe a project's history; status information about the different documents to be produced; different tasks to be made by members of a team, including the time allocated; estimated costs, in person/years, of a particular phase; or critical phases of a project.

### 6.1.3 DesignNet objects

Each component in the DesignNet model is represented by objects grouped in an object-oriented database. Figure 6.2 is part of DesignNet's inheritance graph.

Each object can have properties or attributes connecting it to a primitive object (e.g., a number or a string), a composed object or a collection of objects. Figure 6.3 presents some of the relations. It shows that each place and each transition have a `createTime` property, which indicates their creation date. It also shows the hierarchical decomposition of a project:

- the `decomposedType` property is initialized to `null` when there is no decomposition, otherwise this property refers to an `and` or `or` operator to indicate the kind of decomposition;
- the `decomposedInto` property references a collection containing the element places of the decomposition;
- the `parent` property references the place that encompasses the current place.

The DesignNet system use writes these relations in a textual manner, in TDL (Type Definition Language).
Figure 6.1 DesignNet example
Figure 6.2 DesignNet inheritance graph
% Time object
define Type TemporalRelation is enum(BEFORE,AFTER,EQUAL);
define Type TimeScale is enum(MINUTE,HOUR,DATE,MONTH,YEAR);
define Type Time
    supertype = {Entity};
    properties =
        {  tstamp: Integer;
            minute: Integer;
            hour: Integer;
            date: Integer;
            month: Integer;
            year: Integer;
        };
    operations = {  ...  };
end Time;

% Structural Operator
define Type SOperator is enum(None,And,Or);

% Place
define Type Place
    supertype = {Entity};
    properties =
        {  ...  
            decomposedType: SOperator::=None;
            parent: optional Place;
            decomposedInto: optional distributed Set [Place]
            inverse $Place$parent;
            ...  
        };
end Place;
6.1.4 Project management and prototyping

The DesignNet environment is a prototyping environment for project management. It can be used by a project head to finetune and adjust a project's organization; each activity can be executed and simulated many times, by varying the parameters. The object-oriented approach in this kind of environment provides several advantages. The object class hierarchy and the encapsulation of properties provide different degrees of abstraction to a system's developer or to its user. Each degree of abstraction allows different levels of knowledge to be manipulated, from the general order of project activities to the detailed manipulation of a particular activity.

This kind of environment also shows the need for several different formalisms in a prototyping environment, in order to describe the different aspects of the domain. The DesignNet formalism describes the static organization of the functions, resources and products pertaining to a project. The Petri net formalism is used for the dynamic organization of a project: activity synchronization and parallelism between different activities. The TDL formalism defines the objects in the domain (this could be done graphically). Finally, any number of formalisms can be used to visualize the different results required by the manager, including global planning of a project, status reports on different tasks, planning for individuals and cost per activity.

6.2 An interface generator

The general availability of graphical workstations that can manipulate windows, icons, menus and pointing devices (mice, optical pens, touch screens, etc.) has encouraged research in graphical programming environments [74, 118, 166] and in methods for designing graphical software [78, 115].

These environments can reduce the learning curve for the tools that are made available, increase productivity and, more generally, reduce the programming to be done by the user [52]. The last includes the small programs and files that a user must write to configure the work environment. The Apple Macintosh environment completely eliminates such programming. Of course, this approach does have its critics, who feel that they should be able to use an operating system's command language.

The important rôle played by application generators in the software prototyping phase has already been examined. Below, an object-oriented graphical interface generator developed at the Technological University of Delft (Netherlands) [65] is presented. This generator decomposes highly interactive applications into three parts: static, dynamic and viewing.
6.2.1 General presentation

The major problem when building interactive applications is that the user interface may well be linked to the very semantics of the software being developed and, as such, must be considered to be part of it. The diverse forms of dialog and the number of possible interactions between the application and the user make the design, the implementation and the maintenance of this kind of application very difficult.

To simplify these tasks, they are separated into three distinct parts: static, dynamic and viewing. This decomposition simplifies the implementation of each of the phases in the life cycle.

Static part
The static part consists of the definition of classes manipulated by the application. It describes the modeled objects: their types, their instance variables, and the relations between them and the other objects (aggregation or inheritance).

Most objects manipulated by these applications have a graphical representation on the screen: buttons are represented by rectangles with rounded corners, boxes by rectangles with sharp corners and connections between boxes by arrows. The static part defines their state at a particular instant of the interaction between the user and the machine.

Dynamic part
The dynamic part describes the different possible states of the dialog when the application is running. To describe the states and their transitions, the system uses a set of production rules of the form:

\[ \text{StateActPos}_i \times \{ \text{Test} \} \times \text{Action} \rightarrow \text{StateActPos}_{i+1} \times \{ \text{ChangeSystem} \} \]

where \( \text{StateActPos} \) (states, actions, possibilities) is a possible state in the application's dialog with the user, the optional \( \text{Test} \) is a constraint that must be satisfied by the objects described in the static part, \( \text{Action} \) is the user's action and the optional \( \text{ChangeSystem} \) is the update of the objects in the application.

Here is how a production rule is interpreted. If the dialog is in state \( \text{StateActPos}_i \) and the application's objects satisfy \( \text{Test} \) and the user executes action \( \text{Action} \), then the dialog will switch to state \( \text{StateActPos}_{i+1} \) and the application's objects will undergo change \( \text{ChangeSystem} \).

Hence the dynamic part is a function

\[ f : D \times S \times E \rightarrow D \times S \]

that corresponds to the possible production rules, where \( D \) is the set of possible dialog states, \( S \) is the set of possible states for the objects defined in the static part and \( E \) is the set of events that the user can trigger.
Viewing part
The viewing part describes how the objects, in their different states, should be viewed on the screen. It is a function \( g : S \rightarrow G \), where \( S \) is the set of possible states for the objects and \( G \) is the set of graphical representations of the objects on the screen.

The function is implemented as a set of production rules of the form

\[
Test \rightarrow \text{ChangeScreen},
\]

where \( Test \) is a constraint to be satisfied by the objects and where \( \text{ChangeScreen} \) is the screen update.

The production rules in the dynamic part are validated as soon as the user executes an action. However, in the viewing part, the production rules are only validated when the static part changes state.

So, from the MVC or MPVC point of view, in this three-part model, an interactive application is described using a static model, a function \( f \) describing the dynamic part and a function \( g \) describing the viewing part (see Figure 6.4).

![Diagram showing the relationships between different components of an interface generator](image)

**Figure 6.4** Representing functions in the execution cycle

### 6.2.2 Design of a graphical editor

This model is illustrated by the description of a graphical editor that draws dots and line segments. The editor’s window (Figure 6.5) has two subviews: the work area to draw dots and lines, and the three-button command bar. The editor handles a cursor, which can be used to draw or to select a command.

**Static part**

Once the application’s objects are defined, the designer enters the modeling phase, in which the editor’s static part is defined.

- A window has a cursor with coordinates.
- A window has a command bar.
- A window has a work area, including possible points and lines.
- A command bar has three buttons with coordinates.
Figure 6.5 Graphical editor

Figure 6.6 illustrates the aggregation relations and Figure 6.7 the inheritance relations. Command bars and work areas are both subviews, and each has a coordinate for its upper left-hand corner, a width and a height.

Figure 6.6 Aggregation relations

Figure 6.7 Inheritance relations

Dynamic part
Figure 6.8 illustrates the editor's state transition diagram. The D<sub>i</sub> nodes are the dialog's different states, the arrows are transitions between states, and the labels
on the arrows are the user's actions that provoke the state changes. Here is the textual form, as production rules.

- **D0** x Open
  -> **D1** x View1 "View.New
  -> **D0** x View1.Close
- **D1** x View1.WorkArea.InArea(View1.Cursor.Pos) x PressButton
  -> **D1** x Point.New(View1.Cursor.Pos)
  -> **D2**
  -> **D0** x View1.Close
- **D2** x View1.WorkArea.InArea(View1.Cursor.Pos) x PressButton
  -> **D3** x NewLine "Line.New(View1.Cursor.Pos)
- **D3** x View1.WorkArea.InArea(View1.Cursor.Pos) x MoveMouse
- **D3** x View1.WorkArea.InArea(View1.Cursor.Pos) x ReleaseButton
  -> **D2** x NewLine.SetEnd(View1.Cursor.Pos); NewLine "NoObject

As can be seen, the graphical representation is easier to understand – faster to understand, as well – than the textual representation [118]. It would have been useful if the decomposition model had been used to implement a state transition diagram. Such an editor would certainly have made the implementation of such a trilogy, hence of an interactive implementation, more efficient.
Viewing part
Defining the viewing part consists of defining the viewing methods for the objects on the screen: windows, buttons, lines and dots.

This decomposition paradigm is another example, along with MVC and MPVC, of an object-oriented decomposition method for applications. All three illustrate an application’s polymorphism. In functional and imperative languages, such as LISP, Pascal or C, only the functional organization of an application is made explicit. Since it uses a class hierarchy, object-oriented programming ensures that modules composing a program can adequately represent the stated problem and the chosen method for solving it: ‘The concept of main program, a necessary step in the access to the system (in functional programming), does not exist. It is possible to send a message to any object. Adding a new module takes place without upsetting the rest of the system’ [63, p. 74].

6.3 The ECCAO project
Most current software engineering environments attempt to automate one or several phases of the software life cycle: specification [33, 196], design [148], program transformation [142], software reuse [16] or maintenance [135]. Furthermore, these environments are usually specialized for a given domain: management [178], real-time [6], parallelism [13], project management [119], quality control [7] or safety [95]. So, each of the tools offers one or more formalisms that are adapted to the phase of the life cycle being examined and to the particular domain to be handled.

We choose to present the ECCAO project [31, 33, 59, 106]. Not only is it the origin of the MPVC environment presented in the second part of this book but it is also a platform that can evolve, integrating and handling different formalisms and covering the entire specification process of a computer application.

6.3.1 General presentation
The ECCAO project’s aim is to build an interactive, graphical specification environment that integrates formalisms as varied as

- the graphical editors SADT [159], WQN [145], StateCharts [84] and CHN [58];
- the natural language editor KL-ONE [34];
- symbolic evaluation methods [77], which must ensure that the simultaneous use of different models remains consistent;
- partial evaluation methods [165] of specifications to ensure the mutual enriching of formalisms; and
- graphical animation mechanisms [39] for these different formalisms.
The general organization of the project is given in Figure 6.9. The entire project is being developed using Smalltalk-80.

![Diagram of the ECCAO project](image)

**Figure 6.9** The ECCAO project

The original idea of the project was to describe all of the formalisms to be used by employing the same internal representation, a semantic network in KL-ONE format [21, 34]. The different specification models then attach themselves to the semantic network by taking advantage of the concepts that they hold in common and by adding their own concepts. Using this *homogeneous* internal representation, evaluation mechanisms for the semantic network can be implemented to test, validate, manipulate and augment each of the models.

### 6.3.2 Implemented formalisms

**KL-ONE**

Objects in KL-ONE, similar to Minsky’s *frames* [131], are used to manipulate semantic networks [21, 32, 106, 123]. To do this, KL-ONE distinguishes two levels of presentation: the *primitive* level for generic concepts and the *derived* level for individual concepts.
A concept is completely defined by its structure. It is linked to its roles (its attributes). The links express the relations between the network's elements:

- An individual concept creates a generic concept. This subsumption relation is expressed by a kind-of link.
- A role-of link ties a concept to one of its roles.
- A restricted-to-value link ties a role to a concept.
- An exception or counter-exception ties a concept to a link.

For example, the SADT formalism's generic concept of box is formalized in KL-ONE in Figure 6.10. An SADT box is a kind of Node with three kinds of input: inputs, controls and mechanisms. Each of these is an SADT arrow, itself a kind of Arc. In addition to inheriting the attributes output and label, an SADT box also has a task attribute, which is a kind of Activity.

Using a semantic network as incremental knowledge base provides a homogeneous structure that allows the mutual enriching of different models and the easy implementation of new concepts, in addition to providing an intelligent query method, allowing both inductive and deductive reasoning.

In the ECCAO environment, the user can augment the system with new concepts by using the graphical editor KL-ONE. Already existing generic concepts can be consulted or modified; individual concepts can be created and individual roles can be consulted or changed.
To do this, the entire Smalltalk-80 environment is available. Nevertheless, the user will not normally do this, since the editor for a given formalism itself looks after creating and managing the concepts and roles that it knows how to manipulate.

**SADT**
Ross's Structure Analysis and Design Technique [159] can be used to specify a problem by building a model for it, thereby allowing a deeper understanding of the problem.

In this formalism, a problem is analyzed in a top-down, modular, hierarchical and structured manner (see Figure 6.11). SADT distinguishes two main kinds of semantic entities: *data* and *activities*. The designer uses SADT as a graphical language to define relations between entities: the hierarchical aggregations, the data, control and mechanism flows, and the activity ordering.

![SADT Diagram Hierarchy](image)

**Figure 6.11** SADT diagram hierarchy

Diagrams are formed of labeled boxes and arrows. Each box can be decomposed into a more detailed subdiagram. There are two kinds of diagrams. Actigrams describe the data flow between activities; boxes are the activities and arrows are the data, control or mechanism flows. Datagramms describe the activity flow over the data; boxes are then (possibly composite) data and arrows are individual activities.

**WQN**
The Wait Queue Network (WQN) formalism [145], used in the Performance Analyst's Workbench System (PAWS), models network wait queues. Combined with the PAWS graphical interface, it can model communications between software, hardware and humans.

WQN is used to specify complex communication systems, multiprocessor configurations, unusual input-output configurations, database systems, local networks and operating systems. A WQN model representation can be used to get statistics about the performance of the described system, including cost, reliability, feasibility and saturation (for example, of a peripheral).
Figure 6.12 describes the transactions between a CPU and three disks. Upon leaving the CPU, a transaction has a 1/2 chance of reaching disk 1. The PARTS transactions have a 1/4 chance of reaching disk 2, and the same holds for disk 3. On the other hand, PAYROLL transactions have a 0.4 probability of reaching disk 2 and 0.1 for disk 3.

![Diagram showing CPU and three disks](image)

**Figure 6.12** CPU and three disks in RFA

*Statecharts*

Statecharts [84] was invented for the specification of real-time and parallel systems. As with the above formalisms, it uses a graphical interface. The two semantic entities that the formalism manipulates are *states* and *events*, which provoke state changes.

As an example, Harel examines a quartz watch with four buttons (see Figure 6.13). The Statecharts subdiagram for the stopwatch function is given in Figure 6.14. It shows that the *Stopwatch* function can only be entered if the system is in the *Chime* function and button *a* has been pressed.

![Diagram of a quartz watch](image)

**Figure 6.13** Stopwatch

The *H* icon indicates that when returning to the *Stopwatch* function, the system should return to the *Stopwatch* substate that it was in when it last left that function. On the first entrance, the chosen state is *zero*.

Pressing button *b* initiates the simultaneous activation of the substates *Display* and *Run*. Pressing button *b* again starts up the stopwatch. Pressing button *d* when the stopwatch is running freezes the display to show the lap time (state *Lap*). When the stopwatch is interrupted, pressing the *d* button resets the stopwatch (state *Zero*), and so on.
CHN
The CHN formalism [58] is a interface language equivalent to Ada packages [139], interface specifications or C libraries [101]. With CHN, one can manipulate a software's design at the highest level (e.g., general organization of the modules and libraries) or at the lowest level (e.g., organization of procedures, scope and type of variables, overloading). Figure 6.15 presents a library and its procedures, along with the dependency links (who uses whom) between them.

6.3.3 Evaluation mechanisms
The diversity of the formalisms used in the ECCAO project means that many different software concepts are being manipulated, as are concepts from individual domains: kinds of activity, dataflow, transactions, software and hardware components, stochastic laws, parallelism, real-time, and so on.

These concepts must be expressed in a consistent internal form so that the different formalisms can interact and constraints can pass from one formalism to another. The KL-ONE representation used in ECCAO answers this need.
A formalism used in the ECCAO project is therefore perceived as a particular point of view about the semantic network. Graphical and functional tools that have a formalism are used as access filters to the network’s concepts. These filters isolate information that they can manipulate. It matters little that information was provided in one formalism, what does matter is that the information be made available to all, so long as they can manipulate it.

The MPVC environment, described in the second part of this book, can handle interpretation, animation and validation tools for the different formalisms. Each MPVC trilogy, in the ECCAO project, is a functional filter. The model in the trilogy is the entire semantic network. The implemented controllers describe different active filters. The points of view describe the evaluation semantics (symbolic, partial, finetuning, animation) that can be attached to the active filters. The MPVC trilogy offers a functional structure that is coherent and homogeneous, thereby facilitating the building and the handling of the evaluation mechanisms that are implemented by the above formalisms.

In this kind of environment, an MPVC system can be seen as a set of controllers and points of view for each formalism. For a given formalism, the user selects the controller and the desired points of view for that formalism and then starts up the trilogy.

Although it was not done, it would have been perfectly appropriate to design trilogies that were directly related to the KL-ONE formalism, thereby allowing a global point of view for the semantic network, with facilities for saving the network and reorganizing the network after searching for concepts with similar structure but different names.
Part II

The Model–Point of View–Controller approach

This part of the book presents the design and implementation of an interactive system for building prototyping environments, using the MPVC approach.

This approach can be used to build any program by combining three distinct aspects:

- the model, which defines the organization of the data manipulated by the program;
- the controller, which traverses the data; and
- the point of view, which defines the interpretation for a traversal.

These three parts form an MPVC trilogy. Combined together, several of these trilogies form an MPVC system.

The MPVC approach is an object-oriented design method that simplifies the implementation of multiple interpretation, manipulation and simulation tools for data structures, including semantic networks, syntactic rules and programs in general. The mechanical aspects (the instructions in a program dealing solely with the manner in which data is traversed) of an application are separated from the evaluation or semantic ones. In so doing, multiple traversal mechanisms, interpretations and representations of the same structure become viable.

As an example, an MPVC system implementing multiple interpretations of McCulloch and Pitts’ neural nets is given. This implementation shows that this approach is in fact a prototyping method, since it clearly separates the mechanism for traversing a representation model from its multiple interpretations. In so doing, it simplifies the writing of interpreters for representation models. The MPVC environment is a tool for designing interpreters of multiple representation models.
Implementing this approach in the MPVC environment allows the designer to:

- graphically design multiple traversals of the same model;
- design multiple points of view;
- experiment with MPVC systems, i.e. the combination of different traversal mechanisms with different points of view;
- build many debugging tools optimized for personal use, such as video or step-by-step traces, and pre- and post-tests;
- study the behavior of these systems by observing the model’s behavior in a sequential or in a parallel manner;
- finetune the tools in the basic environment, or augment the latter with new tools – points of view – that can examine or modify the behavior of these systems.

The last capability arises from the reflexive nature of our system. In fact, the MPVC environment itself is an example of the design, implementation and use of reflexive systems.

These characteristics encourage a speculative approach by the user–designer towards a problem: interpreters and tools can be built simultaneously so that they can be experimented with and validated. The resulting environment is a platform for mechanical (the controllers) and semantic (the points of view) schemata for traversing representation models.

The current MPVC environment is an experimental environment for problem resolution. It allows multiple levels of abstraction, in terms of both diversity of representation models that can be designed and the kinds of reasoning that can be implemented. Each aspect of a problem statement can be covered and prototypes can be built quickly for each resolution method.
The Model–Point of View–Controller trilogy

7.1 Multiple representations

In industry, many different methods and tools, with different functionalities, are used to specify programs and systems [41]. For example, the SADT formalism [159] (see Figure 7.1) can be used to specify a computer system’s functions (the boxes) in a hierarchical manner, along with the data and control flows (the arrows) between these functions.

![Figure 7.1 SADT actigram](image)

The Wait Queue Network (WQN) [145] (see Figure 7.2) can model the communications between a system’s software and hardware components, quantifying the use of resources in a probabilistic manner.

![Figure 7.2 WQN diagram](image)
These formalisms can allow specifications to be expressed in a textual form (constraints expressed using logical or mathematical assertions) or in a graphical form (icons representing software or hardware components). They are high-level languages that allow the direct manipulation of entities such as the hard disk, the printer and the CPU, as well as complex logical and statistic operators adapted to a particular domain.

During the specification and design phases of a software system, the designer must create several different representations, at different levels, in order to describe and represent the different aspects of hardware and software that must be implemented. To do this, either already existing formalisms must be used or new ones must be invented for the field under study. The MPVC approach allows the cohabitation and the cross-fertilization of different representation models.

7.2 Multiple interpretations

A software engineering toolkit that only offers graphical tools to describe a computer system in one or more formalisms is nothing more than a system of specialized graphical editors. It is therefore indispensable to augment such editors with interpretative capabilities. Not only must a specification respect the formalism's syntax, but the induced semantics of a set of interconnected components must be tested and validated. This task can be performed by symbolic evaluators [77, 192], partial evaluators [165] or graphical animation.

For example, syntactical restrictions for the SADT editor include not allowing more than six boxes to be created per diagram or a control port arrow at the top of the diagram to be connected to an input data port arrow at the left of the diagram.

As for the semantics of an SADT diagram, if the input values are known, then a symbolic evaluation can take place (see Figure 7.3), in order to test type consistency (do connected input and output ports have the same type?), to test the completeness of specifications (are all used data and activities actually specified?) and to augment the description of computed information during the evaluation (e.g., to determine the type of output data).

![Figure 7.3 Symbolic evaluation of an SADT diagram](image-url)
The MPVC system allows the traversal and the implementation of multiple interpretations of an SADT diagram [110] (see Figure 7.4). This system allows information to be extracted from the data flows passing through the diagram and the control flows that affect the data. This kind of partial evaluation allows certain aspects of the system that are expressed in the WQN formalism to be validated. In addition, SADT specifications can be augmented by statistical results about the performance and usage of the described system.

![Diagram](image.png)

**Figure 7.4** Partial evaluation of an SADT diagram

In general, a partial evaluator [165] traverses a representation model in a selective manner, extracting information that can be treated by different specification tools.

**Graphical animation** is another form of interpretation and validation of specifications. Graphical representations allow, in real time, the relationships between different components of an image (height, color and placement of objects) to be seen. The animation of a specification's graphical representation, even if it offers no information that can be used by other tools in the system, allows the designer, the expert and the client to have a global view of the system's behavior and to understand it better without attending to the details of the formalism.

For example, the graphical animation of an SADT diagram shows the order of application of the functions, as well as the circulation of data flows through the network (see Figure 7.5).

The above list of possible interpretations is hardly exhaustive. It depends on the kind and number of formalisms being used, the information that the user wishes to extract and the tests to which the specifications are subjected. From now on,
these different interpretations are called *evaluation points of view, or evaluation semantics*, of the representation model.

### 7.3 Factorization of the traversal

Implementing different interpretations of a representation model shows that there are sections of code that are identical under all interpretations; this result holds because all interpretations traverse the *same* model. See Figure 7.6.

---

**Figure 7.6** Structured analysis

For example, the three interpretations of the SADT diagram, the symbolic evaluator, the partial evaluator and the graphical animator, share a certain section of code. By *factoring* out the common part, the instructions that are specific to a particular interpretation can be singled out. The specific instructions describe the behavior intrinsic to the interpretation, while the factored instructions describe
the general model traversal mechanism. Hence, extracting the symbolic evaluator's traversal mechanism means that the instructions specific to that point of view have been isolated (see Figure 7.7). These instructions describe the different behaviors to be adopted by the symbolic evaluator during each of the steps in the SADT diagram's traversal.

![Diagram showing the relationship between point of view, symbolic evaluation, controller, control mechanism, and model.](image)

**Figure 7.7 MPVC factorization**

This factorization of the traversal mechanism for a model's representation is the basis for the MPVC approach, which systematically implements any interpretation of the model using three distinct parts: a *model* that describes the representation to be multi-evaluated, a *controller* that describes the factorized traversal mechanism and a *point of view* that describes the intrinsic components of the interpretation. The following definition concludes the discussion.

**Definition 8.** MPVC approach: Let $A(M)$ be an application traversing model $M$, $C_A$ be the traversal mechanism of model $M$ by $A(M)$ and $P_A$ be the evaluation semantics for $A(M)$. Then the MPVC approach is said to induce the following transformation:

$$A(M) \rightarrow \{C_A, P_A, M\}$$
Chapter 8

The kernel of a Model–Point of View–Controller system

This chapter defines the classes and methods that are required to implement the MPVC approach. Each section presents one step in the development of the example, in so doing illustrating the different implementation phases of an MPVC system.

8.1 Defining the model

The model in the MPVC trilogy describes the data structure that is to be multi-evaluated in an MPVC system. For this reason, the first step in the design of such a system is the building of the model.

Implementing the model is initially done independently of any MPVC system, i.e. the implementation of the rest of the system need not be taken into account; hence the model can be developed according to the programmer's usual methods. The MPVC system that is tied to this model is understood as an 'interpretation layer' that is added to and adapts itself to the model's structure.

In other words, a model only defines, at least in the first stages, the structure and the interface methods of the different composing objects, the browsing and interpretation methods being reserved for the MPVC layer.

8.1.1 An example model: neural nets

The basic environment of this model is illustrated by building an example MPVC system for neural nets [130]. These were invented by McCulloch and Pitts [125], researchers in neurophysiology, who used them to attempt to formalize the brain's behavior. Minsky integrated their work into his own research on finite and infinite state automata. A neural net specifies the interactions that can exist between neurons, using a finite state automaton to model the interactions. The example system presented below prototypes the formal model for these automata.
A neural net's basic elements are called cells or neurons (see Figure 8.1). Each neuron has a fibre, called an axon, which can branch out. The branches allow the neuron to be connected to other neurons, through dendrites. A dendrite therefore connects its neuron to one of the branches of another neuron, possibly the same one. There are two kinds of dendrite: exciting and inhibiting (see Figure 8.2). The number of branches of an axon is unlimited, as is the number of dendrites of a neuron.

![Figure 8.1 Cross-section of a neuron](image)

Each neuron is assumed to be a finite state automaton operating at discrete instants, which are synchronized between all the neurons of the same network. At each instant $t$, a neuron transmits or does not transmit an impulse along its axon. For each of the two possible states, a signal is transmitted along the branches of the axon. In turn, the branches transmit the signal to the connected dendrites. These dendrites can then transmit the signal to the neurons that they are part of.

In the given representation, each neuron is labeled with a number, the neuron threshold. Its value defines the value at which the neuron changes state, i.e. emits an impulse along its axon. This change of state is computed according to the kind of dendrites that the neuron has, as well as the signals actually being transmitted along those dendrites.
The state change algorithm can be described as follows. A neuron transmits an impulse at instant \( t + 1 \) if and only if at instant \( t \), (1) none of the inhibiting dendrites was transmitting an impulse, and (2) the number of exciting dendrites transmitting an impulse is greater than or equal to the neuron's threshold value. The preceding \textit{inhibiting} algorithm can be described more formally. Let \( N \) be a neuron, \( s \) its threshold, \( e_t \) its state at instant \( t \); if \( N \) transmits an impulse, then \( e_t = 1 \), otherwise \( e_t = 0 \). Suppose that \( N \) has \( p \) exciting dendrites \( D\text{Ex}_{i_t} \), \( i = 1 \ldots p \), and \( q \) inhibiting dendrites \( \text{DIn}_{j_t} \). Then

\[
e_{t+1} = 1 \quad \text{iff} \quad \sum_{i=1}^{p} D\text{Ex}_{i_t} \geq s \quad \text{and} \quad \sum_{j=1}^{q} D\text{In}_{j_t} = 0
\]

\[
e_{t+1} = 0 \quad \text{otherwise}.
\]

Figure 8.3 shows certain neuronal configurations for a single neuron.

![Figure 8.3 Neuronal configurations for a single neuron](image)

The names used for these configurations describe their behavior in terms of Boolean algebra. Such objects allow the creation of more complex automata. For example, Figure 8.4 shows a serial binary adder [130, p. 45].

![Figure 8.4 Serial binary adder](image)

The preceding automaton corresponds to the following Boolean expressions:

\[
c_{t+1} = a_t \lor b_t \lor d_t
\]
\[ 
\begin{align*}
    d_{t+1} &= (a_t \land b_t) \lor (a_t \land d_t) \lor (b_t \land d_t) \\
    e_{t+1} &= a_t \land b_t \land c_t \\
    f_{t+2} &= c_{t+1} \land \neg d_{t+1} \\
    g_{t+2} &= e_{t+1} \\
    h_{t+3} &= f_{t+2} \lor g_{t+2} \\
    &= (a_t \land \neg b_t \land \neg d_t) \lor (\neg a_t \land b_t \land \neg d_t) \lor (\neg a_t \land \neg b_t \land d_t) \lor (a_t \lor b_t \lor d_t)
\end{align*}

This brief presentation of neural nets should suffice to understand the MPVC implementation for this example. This model is used to illustrate several important aspects of the MPVC methodology, through the rapid development of evaluators and a graphical simulation of such nets.

8.1.2 Implementing the basic model

As was seen above, implementing the model is the first step in the design of an MPVC system. This requires enumerating and implementing the basic classes necessary for the construction of the object model, here the formal model for neural nets.

Since neural nets consist of neurons connected by dendrites, it is natural to define classes for each of these.

The `ObjectImpulsive` class

Most of the objects in this model share the characteristic of being impulsive. An object can emit (for a neuron) or transmit (for a dendrite) an impulse. To define a notion that is shared, an abstract class is used: only instances of subclasses are to be created, not instances of the class itself.

The `ObjectImpulsive` abstract class does not inherit any particular attributes, and so is an immediate subclass of `Object`. It is defined as follows:

```plaintext
Object subclass: #ObjectImpulsive
  instanceVariableNames: 'state'
  classVariableNames: ',
  poolDictionaries: ',
  category: 'Neural Net'!
```

This declaration has five arguments. The new class is called `ObjectImpulsive`. Each instance of `ObjectImpulsive` has a single instance variable (`state`), which gives the instance’s impulsive state (1 if there is an impulse, 0 otherwise). There are no class variables, which are shared by all instances. There are no pool variables either (see below). Finally, the class’s category is `Neural Net`; categories
have no functional semantics, but allow classification according to the domain or application.

Pool variables are accessible by instances of different classes that are not necessarily related in the inheritance graph, i.e. not all of these classes need be superclasses or subclasses of others. To define a set of variables shared by several classes, a pool dictionary, itself a global variable, is used. There is only one predefined pool dictionary in Smalltalk-80, TextConstants, which contains the constants that are shared by all classes manipulating text.

Here are the consultation and modification methods for the state instance variable.

```objectImpulsive methodsFor: 'access'!
  state
    ~ state!
  state: valueBinary
    state <- valueBinary
  resetState
    state <- nil! !
```

The first line indicates the class for which the following methods are being defined, as well as the protocol. Protocols are method categories. As with categories, protocols have no semantics; they simply allow methods to be classified according to a particular theme. Here the methods are all organized under the access protocol.

**The Neuron class**

The Neuron class specifies the structure and behavior of neuronal cells. As before, a neuron is characterized by its threshold, its dendrites, the branches of its axon and the impulsive state at a given instant. Hence Neuron is defined as a subclass of ObjectImpulsive and its instances inherit the state instance variable.

```objectImpulsive subclass: #Neuron
  instanceVariableNames: 'threshold dendrites axon'
  classVariableNames: 'classVariableNames:
    poolDictionaries: ''
  category: 'Neural Net'
```

The threshold instance variable is a number corresponding to the neuron’s transition threshold; dendrites contains the neuron’s dendrites and axon contains the dendrites that are connected to the neuron’s axon branches.

When a neuron is created, its instance variables must be initialized. To do this, the class method new for class Neuron is redefined locally. In Smalltalk-80, this is easily implemented since the Neuron class is a subclass of its metaclass (see Section 4.5.1). Once the instance has been created, the initialize message is sent to the new neuron.
Neuron class methodsFor: 'instantiation'!

new
  "Instantiation"
  ^ super new initialize!

Neuron methodsFor: 'initialization'

initialize
  "Initialization of instance variables at instantiation"
  threshold <- 0.
  dendrites <- OrderedCollection new.
  axon <- OrderedCollection new!

The threshold variable is arbitrarily initialized to 0. As for dendrites and axon, they are initialized to an empty ordered collection. In Smalltalk-80, a collection, instance of the Collection class, is a grouping of objects, not necessarily all of the same kind, that can be manipulated either individually or collectively. The Collection abstract class describes the common behavior for any kind of collection. Its subclasses include Array, Set and OrderedCollection.

Here are the methods needed for reading and modifying a neuron's instance variables.

Neuron methodsFor: 'access'

threshold
  "Returns the neuron's threshold value"
  ^ threshold!

threshold: aNumber
  "Updating the threshold value"
  threshold <- aNumber!

input
  "Returns the input dendrites"
  ^ dendrites!

deductInput: aDendrite
  "Add an input dendrite"
  dendrites add: aDendrite!

output
  "Returns the output dendrites"
  ^ axon!

addOutput: aDendrite
  "Add a dendrite to the axon's branches"
  axon add: aDendrite!

The Dendrite class

Like the ObjectImpulsive class, the Dendrite class is an abstract class: it specifies the behaviors that are common to both exciting and inhibiting dendrites.
A dendrite connects two neurons: the end, which has an axon branch connected to the dendrite, and the origin, of which the dendrite is a part. Since the dendrite transmits an impulse, it is a ObjectImpulsive with an impulsive state. So, like the Neuron, it is a subclass of ObjectImpulsive with two new instance variables, end and origin.

ObjectImpulsive subclass: #Dendrite
    instanceVariableNames: 'end origin'
    classVariableNames: '
    poolDictionaries: '
    category: 'Neural Net'

The initialization routine is called end:origin:, which takes as arguments the end and origin neurons.

!Dendrite methodsFor: 'initialization'!
end: objectEnd origin: objectOrigin
    "Initialization of dendrite"
    origin <- objectOrigin.
    end <- objectEnd.
    objectOrigin addInput: self.
    objectEnd addOutput: self!!

Not only are the instance variables origin and end initialized, but the connecting neurons are informed of the new dendrite. Therefore, the transmission

Dendrite new end: nEnd origin: nOrigin
creates dendrite D in Figure 8.5. No symbol, either an arrowhead or a circle, has been placed at the origin of the dendrite, since it is still not known whether it is exciting or inhibiting. This information is added at the subclass level.

\[
\begin{array}{c}
\text{impulse flow} \\
\text{direction}
\end{array}
\]

Dendrite new end: nEnd origin: nOrigin
creates dendrite D in Figure 8.5. No symbol, either an arrowhead or a circle, has been placed at the origin of the dendrite, since it is still not known whether it is exciting or inhibiting. This information is added at the subclass level.

\[
\begin{array}{c}
\text{impulse flow} \\
\text{direction}
\end{array}
\]

**Figure 8.5** Creating a dendrite

Two methods allow access to the two instance variables.

!Dendrite methodsFor: 'access'!
origin
    "Returns the origin neuron of the dendrite"
    origin!
end
    "Returns the end neuron of the dendrite"
    end!!

Two tests for receptor dendrites are defined. The `isExciting` method returns a Boolean value indicating whether the dendrite is an exciting dendrite that is transmitting an impulse. The `isInhibiting` method indicates whether the dendrite is an inhibiting dendrite that is transmitting an impulse.

```
!Dendrite methodsFor: 'testing'!

isExciting
"Default method indicating if the receptor"
"is an exciting dendrite and if this"
"dendrite is transmitting an impulse"
^ false!

isInhibiting
"Default method indicating if the receptor"
"is an inhibiting dendrite and if this"
"dendrite is transmitting an impulse"
^ false!
```

Both of these methods are default methods, or default behaviors. Implementing these methods at this level ensures that all dendrites can respond to these messages; by default, an instance of this class is neither exciting nor inhibiting, so both methods return the Boolean value `false` (technically, an instance of class `False`, pointed to by the pseudo-variable `false`).

### The `DendriteExciting` and `DendriteInhibiting` classes

The general behavior of a dendrite can now be modified to specify whether it is exciting or inhibiting. This is done by creating two subclasses of the `Dendrite` class: `DendriteExciting` and `DendriteInhibiting`, and

```
'Dendrite subclass: #DendriteExciting
 instanceVariableNames: 
 classVariableNames: 
 poolDictionaries: 
 category: 'Neural Net'!

'Dendrite subclass: #DendriteInhibiting
 instanceVariableNames: 
 classVariableNames: 
 poolDictionaries: 
 category: 'Neural Net'!
```

The Smalltalk classes `True` and `False` define a similar duality: they are both subclasses of the abstract class `Boolean` and were created in order to define the behavior of the Booleans `true` and `false` when they receive the conditional message `ifTrue:ifFalse:` or the connective messages `or`, `and`, or `not`. The implementation of the conditional is certainly one of the most powerful examples of object-oriented programming.
Every language designer must somehow implement the conditional. Normally this is done using a primitive conditional operator. In Smalltalk, however, because of the existence of the two subclasses True and False, the conditional has the form

```
ExpressionTest ifTrue: TrueBlock ifFalse: FalseBlock
```

A block is a sequence of messages to be sent upon explicit demand, i.e. there is delayed evaluation. A block, instance of the BlockContext, includes a context closure, thereby preserving the environment in which it was defined. Blocks are regularly used by control structures.

Therefore, the message

```
ifTrue: TrueBlock ifFalse: FalseBlock
```

is sent to the test’s result, either true (instance of class True or false (instance of class False). Therefore the method associated with the #ifTrue:ifFalse: selector must at least be defined in the abstract superclass Boolean, possibly in the two subclasses as well. It is, in fact, defined in all three. The Boolean class declares it to be the responsibility of the subclasses.

```
!Boolean methodsFor: 'controlling'!
  ifTrue: trueBlock ifFalse: falseBlock
  self subclassResponsibility!
```

An instance of class True simply evaluates the first argument of the message.

```
!True methodsFor: 'controlling'!
  ifTrue: trueBlock ifFalse: falseBlock
  ^ trueBlock value!
```

As for False, it evaluates the second argument of the message.

```
!False methodsFor: 'controlling'!
  ifTrue: trueBlock ifFalse: falseBlock
  ^ falseBlock value!
```

Not only are these definitions elegant, but the very implementation of an object-oriented language models the programming mechanisms.

Back to the model example. Decomposing the Dendrite means that the two subclasses can specify the two kinds of dendrites in the formal model. The dendrites simply behave differently when they receive the messages isExciting and isInhibiting.

For an exciting dendrite, testing its exciting state simply means testing its impulsive state.

```
!DendriteExciting methodsFor: 'testing'!
  isExciting
  ^ state=1!
```
The behavior is identical for an inhibiting dendrite: it tests the value of its impulsive state, which gives the inhibiting state.

*DendriteInhibiting methodsFor: 'testing'!
  isInhibiting
    state=1! !

It is not necessary to redefine the two methods in the two classes. Since an exciting dendrite is a non-inhibiting dendrite, it must have the same behavior as all dendrites that are not inhibiting, i.e. the default behavior defined for the Dendrite class.

Therefore, by not redefining isInhibiting in the DendriteExciting class, the instances of that class inherit the method definition for the Dendrite class, hence they behave like any non-inhibiting dendrite. The behavior is similar for an inhibiting dendrite and the isExciting method.

These two methods did not need to be defined at the Dendrite class level. They could simply have been defined at the subclass level; no programming error would have resulted. However, the resulting semantics would be different. The above organization of the two methods means that any dendrite can respond to the messages isExciting or isInhibiting. If these methods had only been defined at the level of classes DendriteExciting and DendriteInhibiting, then the interpretation would have been that these behaviors are exclusively reserved for exciting and inhibiting dendrites.

An analogous situation can be found for the isNil and notNil messages in Smalltalk. These messages ask the receiving object if it is an instance of the class UndefinedObject. The associated default methods are defined at the Object class level.

'*Object methodsFor: 'testing'!
  isNil
    ^ false!
  notNil
    ^ false! !

So, by default, any object is non-Nil. For an instance of the UndefinedObject class, these methods are explicitly redefined to have the opposite behavior.

'*UndefinedObject methodsFor: 'testing'!
  isNil
    ^ true!
  notNil
    ^ false! !

Two new entities
The set of classes defined up to now is sufficient for Minsky’s formal neural net model [130]. However, to simplify the manipulation and handling of the model,
two new entities, *neuro-emitters* and *neuro-receptors*, are added (see Figure 8.6) to the object representation of this model. The concepts are modeled by classes *NeuroEmitter* and *NeuroReceptor*.

![Figure 8.6 Neuro-emitter and neuro-receptor](image)

A neuro-emitter is considered to be a source emitting signals, at successive instants, to the *initial dendrites* of the network, i.e. to the dendrites with no axon as origin. A signal can be either an impulse or the lack of an impulse. A neuro-emitter is synchronous with the neurons in the net. In this representation, each initial dendrite must be connected to one neuro-emitter, and one neuro-emitter can be connected to several initial dendrites. The number of neuro-emitters is therefore bound by the number of initial dendrites.

A neuro-receptor is a sink for signals. It collects the signals emitted by a net’s *final axons*, which have no neurons connected to their branches. In this representation, each final axon must be connected to a single dendrite, whose origin is a neuro-receptor. Two final axons cannot have the same neuro-receptor, so there are as many neuro-receptors as there are final axons.

The dendrite connected to a final axon can be either exciting or inhibiting, since this distinction only affects the behavior of a neuron. Nevertheless, for readability reasons, the dendrite connected to a final axon is always presented as an exciting dendrite.

*The NeuroEmitter class*

A neuro-emitter consists of the ordered collection of impulses that it must transmit for its *output* dendrites, i.e. the dendrites for which it is the end. At a given instant, the neuro-emitter is a particular impulsive state, corresponding to the current impulse that must be transmitted, similar to that in a neuron or a dendrite.

```plaintext
ObjectImpulsive subclass: #NeuroEmitter

    instanceVariableNames:  
      'impulses output'

    classVariableNames: 

    poolDictionaries: 

    category:  
      'Neural Net'
```

The impulses instance variable contains the ordered collection of impulses that are to be sequentially emitted to the output dendrites. As for output, it consists of the dendrites whose end is the neuro-emitter.

When a new neuro-emitter is being created, the instance variables are initialized to be empty, through the redefinition of the new method.
!NeuroEmitter class methodsFor: 'instantiation'
new
"Instantiation"
- super new initialize!

!NeuroEmitter methodsFor: 'initialization'
initialize
"Initialization of instance variables at instantiation"
impulses <- OrderedCollection new.
output <- OrderedCollection new!

Here are the basic methods.

!NeuroEmitter methodsFor: 'access'
impulses: aCollection
"Buffer initialization"
impulses <- aCollection!
output
"Return output dendrites"
- output!
addOutput: aDendrite
"Add an output dendrite"
output add: aDendrite!
impulses
- impulses!
extImpulse
- impulses removingLast!

The nextImpulse method means that the impulses are extracted and transmitted from the collection in impulses in last-to-first order. This ordering is appropriate for examining a number during binary addition.

The NeuroReceptor class
A neuro-receptor has two attributes: the dendrite connecting it to the final axon (the input) and the buffer collecting the signals transmitted to its dendrite.

Unlike the other entities described above, a neuro-receptor does not have an impulsive state: it only registers the successive signals that arrive on its dendrite. Therefore the NeuroReceptor class is a subclass not of the ObjectImpulsive class but, rather, of the Object class.

Object subclass: #NeuroReceptor
instanceVariableNames: 'buffer input'
classVariableNames: '
poolDictionaries: '
category: 'Neural Net'!
The instance variable buffer collects the successive binary values. As for input, it is the single input dendrite.

As for the other entities, the new method is redefined, in order to initialize the buffer.

```
!NeuroReceptor class methodsFor: 'instantiation'
new
  "Instantiation"
  ~ super new initialize!

!NeuroReceptor methodsFor: 'initialization'
initialize
  "Initialization of instance variables at instantiation"
  buffer <- OrderedCollection new!
```

Here are the methods.

```
!NeuroReceptor methodsFor: 'access'
input
  "Return the input dendrites"
  ~ input!
addInput: aDendrite
  "Initialize the input dendrite"
  input <- aDendrite!
buffer
  ~ buffer!
collectImpulse: anImpulse
  buffer addFirst: anImpulse!
```

The collectImpulse: method collects, in reverse order, the received impulses.

The `NetNeural` class
Finishing the basic model now only requires the definition of the `NetNeural` class, representing a complete neural net. It includes the neuro-emitters (inputs), the neuro-receptors (outputs), the dendrites (dendrites) and the neurons (neurons).

```
Object subclass: #NetNeural
  instanceVariableNames: 'inputs outputs dendrites neurons'
  classVariableNames:
  poolDictionaries:
  category: 'Neural Net'
```

The redefinition of the new method simply initializes all the variables.

```
!NetNeural class methodsFor: 'instantiation'
new
```
"Instantiation"
~ super new initialize! !

!NetNeural methodsFor: ‘initialization’!
initialize
"Initialization of instance variables at instantiation"
inputs <- OrderedCollection new.
outputs <- OrderedCollection new.
dendrites <- OrderedCollection new.
eurons <- OrderedCollection new!

Here are the basic methods.

!NetNeural methodsFor: ‘access’!
inputs
"Returns the neuro-emitters"
~ inputs!
addInput: aNeuroEmitter
"Add an input neuro-emitter"
inputs add: aNeuroEmitter!
outputs
"Returns the neuro-receptors"
~ outputs!
addOutput: aNeuroReceptor
"Add an output neuro-receptor"
outputs add: aNeuroReceptor!
dendrites
"Returns the dendrites"
~ dendrites!
addDendrite: aDendrite
"Add a dendrite"
dendrites add: aDendrite!
eurons
"Returns the neurons"
~ neurons!
addNeuron: aNeuron
"Add a new neuron"
eurons add: aNeuron! !

The inheritance graph of the neural network model
The first step in defining the MPVC example is complete. The classes in the object model that corresponds to the formal neural net have all been defined. The resulting hierarchy is shown in Figure 8.7.
Creating a network
An example of the use of these classes is the neural net that implements the exclusive or (XOR) Boolean operation (see Figure 8.8). The net consists of four initial dendrites connected to two inputs and of one final axon. Therefore the object representation contains two neuro-emitters, impulse sources for the net, and one neuro-receptor that collects the final axon’s successive states, in addition to the neurons and the dendrites described by the formal model. The object model is shown in Figure 8.9.

```
| nn ne1 ne2 nr n1 n2 n3 n4 |
"Create the new net"
  nn <- NetNeural new.
"Create the input buffers"
  nn addInput: (ne1 <- NeuroEmitter new).
  nn addInput: (ne2 <- NeuroEmitter new).
"Create four neurons"
  nn addNeuron: (n1 <- Neuron new threshold: 1).
  nn addNeuron: (n2 <- Neuron new threshold: 1).
  nn addNeuron: (n3 <- Neuron new threshold: 1).
  nn addNeuron: (n4 <- Neuron new threshold: 1).
```
"Create the output buffer"

nn addOutput: (nr <- NeuroReceptor new).

"Create the dendrites and interconnections"

"between neurons and buffers"

nn addDendrite: "d1"
(DendriteExciting new end: ne1 origin: n1).

nn addDendrite: "d2"
(DendriteInhibiting new end: ne1 origin: n2).

nn addDendrite: "d3"
(DendriteExciting new end: ne2 origin: n1).

nn addDendrite: "d4"
(DendriteInhibiting new end: ne2 origin: n1).

nn addDendrite: "d5"
(DendriteExciting new end: n1 origin: n3).

nn addDendrite: "d6"
(DendriteExciting new end: n2 origin: n3).

nn addDendrite: "d7"
(DendriteExciting new end: n3 origin: nr).

8.2 Defining the controller

The MPVC approach accentuates the fact that several algorithms can traverse the same model in the same manner, i.e. defining the sequence of steps for the model’s traversal is the same for each of the algorithms. Changing algorithms simply means redefining the local behavior at each step. The controller handles the instructions describing the model’s ordering mechanism.

Definition 9. Let \( \{A_i(M)\} \) be a set of algorithms traversing a model \( M \). Then the algorithms traverse their model in the same manner if and only if there exists
a traversal mechanism $C$ such that

$$
\forall i. \ A_i(M) \rightarrow \{C, P_i, M\},
$$

where $P_i$ is the evaluation semantics specific to algorithm $A_i$.

This more formal statement shows that the controller is in fact the factorization of the instructions that form the model's traversal mechanism. These instructions are common to all the algorithms traversing the same model in the same manner.

The previous section explained that an MPVC system is built from its basic model; so defining a model constitutes the first step in the MPVC trilogy. The second step deals with the 'mechanical' aspect, i.e. the system's controller, which uses finite state automata. It is preferable to have a general overview of the ordering of each of the traversal steps before entering into the details of the different evaluation semantics to be implemented.

The MPVC trilogy, along with its different kinds of link, is presented in Figure 8.10.

![Figure 8.10 MPVC trilogy](image)

Activation links describe the triggering mechanisms. The controller triggers the activities defined by its point of view, respecting, of course, its ordering. The next section describes an evaluation semantics that activates processes tied to the model, for example, processes needed for the graphical animation of the model.

Consultation links are used for reading or writing access to the model. As Figure 8.10 shows, the point of view does not have access to its controller, so cannot influence it directly. However, since both the controller and the point of view have access to the same model, a point of view modifying the model's structure can also modify, indirectly, the controller's activity. The choice not to offer a consulting link between the point of view and the controller – for it was a choice in the implementation of MPVC systems – is justified and explained in the next section.

### 8.2.1 Ordering rules, links and graphs

To formalize the controller's traversal mechanism, the ordering rule is introduced: it describes the behavior that a controller must have at a given step in the traversal
of the model. There are three phases.

First, the activation test inquires if the two following steps of the active rule can be activated. The model's current state is checked, more precisely, the state of the object actually being traversed by the controller is checked.

The second step, only triggered if the activation test was positive, activates the controller's point of view. The latter is asked to execute a precise task for the current rule and the current object.

Finally, the controller's next states, i.e. the next objects in the model to be traversed, are computed, as are the ordering rules to be applied. This phase is described by a ordered collection of ordering links. The order of these links determines the sequence in which the referenced rules must be executed (see Figure 8.11). By extension, an ordering graph (see Figure 8.12) refers to any graph whose nodes are ordering rules and whose arcs are ordering links.

![Sequencing rule]

![Box with nodes: Activation Test, View point activation, Next Step(s)]

![Sequencing link]

![Box with nodes: Next rule, Next objects in model]

![Sequencing rule]

![Box with nodes: Activation Test, View point activation, Next Step(s)]

![Sequencing link]

![Box with nodes: Next Rule, Next objects in model]

**Figure 8.11** Ordering rules and links

![Diagram of a starting point with links and rules]

**Figure 8.12** Example ordering graph
To the extent that an ordering rule describes the controller’s behavior at a given moment, the ordering graph can describe the steps to be undertaken, i.e. the entire ordering mechanism for the controller. This mechanism, more precisely its graphical representation, is part of the controller’s characteristics. The graph has a special rule, the ‘graph entry’, which acts as starting point when running the controller. From this rule, all the rules in the graph must be traversable.

The RuleOrdering class
The RuleOrdering class models the state and the behavior of the ordering rules described above. Since a rule has three phases, an instance of class RuleOrdering must have at least three attributes.

First, the activation test (variable textActivation) is defined by a Boolean expression whose evaluation must return a Boolean object. The evaluation of this test can authorize the activation of the current rule. The Boolean expression is implemented as a BlockContext with two arguments. The first is linked at evaluation to the active controller, the second to the object. Only the two variables of this block are linked; there are no other free variables or pseudo-variables. This restriction means that BlockContext can be seen as a λ-expression whose closure is reduced to the bound variables.

Second is the activation of the point of view (variable activityPV). It is the message that should be sent to the point of view so that the specific semantic activity can take place. As for the activation test, this is done as a BlockContext with two arguments, the first argument being the point of view and the second the object.

Third are the next steps for the controller’s traversal (variable nexts). The attribute references an ordered collection of instances of class LinkOrdering, defined in the next subsection. The order of these instances determines the order in which the ordering links should be interpreted.

Last is the rule’s name (variable name), which means that the rule can be referenced textually when the controller’s ordering graph is being defined.

Here is the definition of the RuleOrdering class.

Object subclass: #RuleOrdering
    instanceVariableNames: ['name textActivation
        activityPV nexts']
    classVariableNames: ['
    poolDictionaries: ['
    category: 'MPVC System']!

These variables must, of course, be initialized as soon as the first rule is created. This instantiation-initialization requires one class method and one instance method. They have the same selector, i.e. they are both called named:if:do:. The message takes three arguments: the name of the receiving rule, the activation test expression and the transmission describing the activation of the point of view.
!RuleOrdering class methodsFor: 'instantiation-initialization'
named: aSymbol if: blockTest do: blockPV
"Instantiation"
  self new named: aSymbol if: blockTest do: blockPV! !

!RuleOrdering methodsFor: 'initialization'
named: aSymbol if: blockTest do: blockPV
"Initialization of instance variables at instantiation"
  name <- aSymbol.
  testActivation <- Compiler evaluate: blockTest.
  activityPV <- Compiler evaluate: blockPV.
  next <- OrderedCollection new! !

This technique of selector homonymy is a programming para-technique that simplifies rereading and creation of objects in the class. The class method takes the parameters and creates a new RuleOrdering.

RuleOrdering named: itsName if: aBlock do: another
If this rule did not exist, the user would have to create a RuleOrdering instance explicitly, with new, then send the appropriate messages to the new instance to effect the necessary initializations.

RuleOrdering new named: itsName if: aBlock do: another

The doubling of the selector, made explicit by the homonymy of the two methods, shows that they have the same rôle: initialize the ordering rule, where the method class begins with an instantiation.

This homonymy is often used in the Smalltalk base environment; it greatly simplifies the reading and the understanding of the different classes that make it up. The selector #origin:corner:, used in the Rectangle class, is a good example.

The class method creates a new Rectangle, then calls its homonym, the instance method, which initializes the attributes origin and corner by assigning to each a Point, so that the two points form one of the diagonals of the receiving rectangle. Hence, creating the Rectangle in Figure 8.13 requires the following message.

Rectangle origin: 5@8 corner: 9@10

The Smalltalk developer will notice that the two last arguments are character strings (instances of class String) representing Smalltalk blocks (instances of class BlockContext). This implementation allows the compilation of argument blocks in an empty context. A Smalltalk block, even if it contains no free variable, is always attached to the context in which it was compiled, which causes the system to crash if a block is used in a process (instance of class Process) in another context than the one in which it was compiled.

Five consultation methods are needed to manipulate the variables in an instance of class RuleOrdering.
Figure 8.13 Creating a rectangle

```smalltalk
!RuleOrdering methodsFor: 'access'
  name
    ^ name!
  test
    ^ testActivation!
  activity
    ^ activityPV!
  next
    ^ next!
  nextRule: aSymbol for: blockNext
    next add: (LinkOrdering rule: aSymbol for: blockNext)!
```

Method #nextRule:for: allows the creation of a new ordering link whose origin is the receiving rule of the message. It uses the class method #rule:for: of class LinkOrdering, defined in the next subsection.

**The LinkOrdering class**

This class models directed edges, here called ordering links, that connect two rules in an ordering graph. Two attributes are needed. The name of the rule at the end of the edge, the next rule to be applied, is called ruleNext. The blockNext variable, a BlockNext expression with two arguments (the controller and the object), computes the next object to be traversed.

```smalltalk
Object subclass: #LinkOrdering
  instanceVariableNames: 'nextBlock ruleNext'
  classVariableNames: '.
  poolDictionaries: '.
  category: 'MPVC system'!
```

The first method to be defined is the #rule:for: method. It is used in the instance method nextRule:for: of the RuleOrdering class; it creates and initializes a new ordering link.

```smalltalk
!LinkOrdering class methodsFor: 'instantiation'
  rule: aSymbol for: blockNext
    "Instantiation"
    ^ self new rule: aSymbol for: blockNext! !
```
!LinkOrdering methodsFor: 'initialization'!
rule: aSymbol for: blockNext
"Initialization of instance variables at instantiation"
ruleNext <- aSymbol.
blockNext <- Compiler evaluate: blockNext! !

The selector homonymy para-technique is used again.
Two consultation methods are needed as well.

!LinkOrdering methodsFor: 'access'!
block
^ blockNext!
rule
^ ruleNext!

Modeling the ordering graph
An ordering graph is characterized by its rules and its links. To ensure that it is deterministic, each rule must be unique. The same graph cannot have two rules with the same name.

A graph is modeled by a Smalltalk dictionary (class Dictionary). An instance of Dictionary is an indexed collection of pairs \(\{key \to value\}\) (instances of class Association). Each pair in the ordering graph dictionary references one of the rules in the modeled graph. The key of a pair, index into the dictionary, is the name of the referenced rule and the associated value is the instance of that rule.
Modeling ordering graphs in this manner means that no additional attributes or behaviors, apart from those already in the Dictionary class, are necessary; for this reason it is not necessary to create a new class GraphOrdering. A special class could well have been created to cover exactly the characteristics and the behaviors needed for our system. However, since nothing extra would be provided, it is better to use the implementation of Smalltalk dictionaries, which offer a sufficient environment to model ordering graphs.

8.2.2 The controller

The formal definition of the controller's purpose at the beginning of Section 8.2 states that the controller is the factorization of the instructions that form the model's traversal mechanism. This factorization is explained below.

Vertical factorization
The first section of this chapter already implemented and specified a kind of factorization. It was shown previously that exciting and inhibiting dendrites had similar characteristics and behaviors. The factorization consisted of finding the common traits and putting them into an abstract class called Dendrite.
Definition 10. Let $C_A = (\mu_A, \nu_A)$ be a class with behaviors $\mu_A$ and attributes $\nu_A$ and let $C_B = (\mu_B, \nu_B)$. Suppose that $\mu_a = \mu \cup \mu$ and $\mu_b = \mu \cup \mu\beta$, and that $\nu_a = \nu \cup \nu$ and $\nu_b = \nu \cup \nu\beta$. Then the behaviors $\mu$ and the attributes $\nu$ can be factored out and placed in an abstract class $C$, superclass of classes $C_A$ and $C_B$. The resulting inheritance graph is shown in Figure 8.14.

\[ C = (\mu, \nu) \]
\[ C = (\mu_a, \nu_a) \]
\[ C = (\mu_b, \nu_b) \]

**Figure 8.14** Vertical factorization

This *vertical* factorization is called *abstraction*: A new level of abstraction in the modeling of concepts has been added, through the addition of a new abstract superclass $C$.

Factoring uses the inheritance mechanisms of object-oriented languages. Such mechanisms often find subsumption relations between modeled concepts [21, 32]. In fact, the inheritance relation that exists between the Dendrite class and the DendriteExciting class can be interpreted as 'in general, dendrites subsume exciting dendrites' or as 'an exciting dendrite is a kind of dendrite, which is itself a sort of impulsive object, which is itself a kind of object'.

The factorization of the model's traversal mechanism is of another kind. First, it is based on the *conceptualization* (forming and organizing concepts) of different aspects of an algorithm. This decomposition of independent concepts is standard in object-oriented programming, but the decomposition of an algorithm is not. The resulting trilogy is a horizontal decomposition of the algorithm (see Figure 8.15).

\[ \text{algorithm} \]
\[ \text{decomposition} \]
\[ \text{controller} \]
\[ \text{model} \]
\[ \text{pt. of view} \]

**Figure 8.15** Horizontal decomposition of the algorithm

Factoring the traversal mechanism consists of computing the intersection between several decompositions of algorithms. We call the factorization horizontal because it is the result of a horizontal decomposition (see Figure 8.16).

The relations between the different parts of an MPVC trilogy are not subsumption relations. A given point of view is not a kind of controller handling its own activation, nor is a controller a kind of point of view. However, in the section on
meta-controllers (see Section 9.4), certain aspects of controllers are subsumed by the point-of-view concept.

The ControllerMPVC class
This abstract class includes the basic characteristics and behaviors shared by all controller classes, used for traversal mechanisms. This class is the root of the controller inheritance graph and is derived from the functions defined above.

A controller can be defined by its ordering graph, which formalizes the idea of traversal mechanism; by the model that it traverses, as well as the model’s object, starting point for the traversal; and by the point of view that must be activated at each traversal step.

Controllers are also the result of a horizontal factorization and should not inherit from any class other than the one that is currently being defined. Hence the ControllerMPVC class is defined as a subclass of the Object class.

Object subclass: #ControllerMPVC
instanceVariableNames: 'pointOfView model graphOrdering startConcept'
classVariableNames: '
poolDictionaries:

category: 'MPVC system'!

The instance variable pointOfView indicates the point of view to be invoked at each step; model designates the current model being traversed; graphOrdering describes the controller’s ordering rules; startConcept designates the first object in the model that should be traversed.

Here is the method for initializing the instance variables.

!ControllerMPVC class
methodsFor: 'instantiation-initialization'!
model: aModel pv: aPV startConcept: aConcept
"Instantiation"
self new model: aModel pv: aPV startConcept: aConcept! !
ControllerMPVC methodsFor: 'initialization'!
model: aModel pv: aPV startConcept: aConcept
"Initialization of model, pointOfView and startConcept"
model <- aModel.
pointOfView <- aPV.
startConcept <- aConcept!!

The previous method does not allow the initialization of graphOrdering; this is
done by the initialize method, which is invoked through the local redefinition
of the new class method.

ControllerMPVC class
methodsFor: 'instantiation-initialization'!
new
"Instantiation"
~ super new initialize!!

The initialize method is first defined at the ControllerMPVC class level.

ControllerMPVC methodsFor: 'initialization'!
initialize
"Initialization of the ordering graph"
"is done at the subclass level"
sel subclassResponsibility!!

Therefore the initialize method is a default method for the basic behavior of
all controllers. Since each controller should be able to respond to that message, it
was defined at the level of the abstract class ControllerMPVC.

However, the method should be redefined for each of the controller classes, all
subclasses of ControllerMPVC. In other words, the ordering rules for a controller
are the responsibility of its class. This requirement is formalized with the single
transmission in the definition of the initialize method.

self subclassResponsibility

The evaluation of that transmission interrupts the current computation and ‘notifies’ (class NotifyView) about the irregularity. Should the initialize message
defined in the ControllerMPVC class be invoked, then the newly created controller
does not have access to its own ordering rules, since these are not defined in the
initialize method local to its class.

The subclassResponsibility message is often used in the Smalltalk-80 base
environment. It is used whenever it is necessary to define that a certain class of
objects can respond to a particular message but that the definition of the correct
behavior depends on the subclass of the object at hand.

Hence the method associated with the #< (less than) selector is first defined in
the abstract class Magnitude, which incorporates the common behaviors for all
magnitudes, thereby stating that any magnitude must respond to that message.
Magnitude methodsFor: 'comparing'
< aMagnitude
  "Answer whether the receiver is less than the argument"
  ^ self subclassResponsibility!

The < method is redefined in the Character, Date and Number classes, all subclasses of the Magnitude class. In each of Magnitude’s subclasses, the < method describes the specific behavior to be adopted by the object receiving the message in order to determine if it is less than the message’s argument.

For example, in the Character subclass, the method states that for a character to be smaller than another, then the two associated ASCII codes should be compared.

Character methodsFor: 'comparing'
< aCharacter
  "Answer whether the receiver’s value"
  "is less than aCharacter’s value"
  ^ self asciiValue < aCharacter asciiValue!

Comparing two instances of class Date consists of comparing the day and the year in the two dates.

Date methodsFor: 'comparing'
< aDate
  "Answer whether the argument aDate"
  "precedes the date of the receiver"
  ^ year = aDate year
  ifTrue: [day < aDate day]
  ifFalse: [year < aDate year]!

Methods using the subclassResponsibility message should be considered as operational assertions, which can be handled directly by the Smalltalk-80 system’s interpreter. This development aid, which can be used by any Smalltalk designer, makes of Smalltalk an incremental programming environment and a method for designing and programming integrated-software management systems.

To complete the ControllerMPVC definition, consultation methods, including the method for initializing the graphOrdering instance variable, must be added.

ControllerMPVC methodsFor: 'access'
model
  ^ model!

pv
  ^ pointOfView!

graph
  ^ graphOrdering!

start
  ^ startConcept!
addRule: aRule
"Add a new rule in the ordering graph"

graphOrdering isNil
  ifTrue: [graphOrdering <- Dictionary new.
    graphOrdering at: #open
    put: (RuleOrdering named: #open
      if: '[[:ctrl :c | false]'
    do: '[[:pv :c | ]'])].

graphOrdering at: aRule name put: aRule!

The last method, addRule:, is used to define the ordering graph of the receiving controller. This method states that the ordering graph is initialized by a rule called #open, which should be the first rule to be invoked during the controller’s activation, i.e. it should be the entry to the ordering graph. This rule defines a default behavior, in particular that the following phases should in no way be invoked. This is done through the activation test block [:ctrl :c | false], whose evaluation systematically returns the value false. Therefore, by default, none of the other rules in the graph will be activated, since the only entry point is in fact a terminal rule.

So, the designer must redefine this rule so that it starts up the traversal of the ordering graph. This redefinition consists of reassigning to the graph a rule with the same name #open, thereby redefining the default rule.

8.2.3 Controller for a neural net (version 1)

There is now sufficient information to define a first traversal mechanism for the neural net model. It was shown above that the essential phase in building a controller is the definition of its ordering graph. Below is the formalization of the ordering of the different steps in the traversal of a neural net.

**Intuitive description**

Traversing a neural net begins with the activation of the evaluation semantics attached to the net’s neuro-emitters. In the object representation of the XOR neural net example, this implies the activation of the current point of view for the neuro-emitters marked by arrows in Figure 8.17. Once the evaluation semantics for a neuro-emitter has ended, then the evaluation semantics for the dendrites with that neuro-emitter as end should be activated. As a result, should neuro-emitter A be traversed, then the two dendrites marked by arrows in Figure 8.18 should have their current point of view activated. Should neuro-emitter B be traversed, then the two other dendrites (see Figure 8.19) should have their current point of view invoked. Once the evaluation semantics for a dendrite is finished, the evaluation semantics for the neuron or the neuro-receptor containing that dendrite should be activated.
So, depending on the last evaluated dendrite, one of the three neurons or the neuro-receptor can be activated (see Figure 8.20).

The evaluation semantics for a neuron can only be activated if all its input dendrites have been previously activated. Once that has been done, the evaluation semantics of the dendrites connected to the current neuron's axon branches can be activated. In this example, one of the three dendrites marked by arrows in Figure 8.21 is to be activated, depending on which neuron was actually activated.

The end of a neuro-receptor's evaluation semantics is not followed by other activations. It is a final step in the net's traversal.

**Formalization**

This informal description suffices to define the rules of the ordering graph for the controller describing the traversal mechanism for the neural net model.
The #open rule is the entry point to the ordering graph.

(RuleOrdering named: #open
  if: ‘[:ctrl :c | true]’  "Activation test block"
  do: ‘[:pv :c | ]’  "Point of view activation block"
  nextRule: #neuroEmitter  "Next rule"
  for: ‘[:ctrl :c | c inputs]’
)

The activation test block, which simply evaluates the true pseudo-variable, states that the following phases of this rule must always be activated. As for the point of view activation block, it contains no transmission, which means that the point of view is not to be invoked by this step. The #open rule is only used as a unique entry point on the ordering graph for the controller that is currently being defined; hence no evaluation semantics is given for this stage.

This rule has a unique ordering link that states that the next rule to be activated is called #neuroEmitter and that

[:ctrl :c | c inputs]

is the expression to be evaluated to obtain the next elements to be traversed. In this case, they are the net's input neuro-emitters. The previous expression makes the assumption that the initial object to be traversed is an instance of class NetNeural that contains the basic components of the net to be traversed.

The #neuroEmitter rule defines the controller's behavior when it is traversing a neuro-emitter. Recall that a neuro-emitter contains all the signals that it must emit.
(RuleOrdering named: #neuroEmitter
   if: ‘[:ctrl :c | c impulses isEmpty not]’
   do: ‘[:pv :c | pv activateNeuroEmitter: c]’
   nextRule: #dendrite
   for: ‘[:ctrl :c | c output]’
)

The activation test block

[ :ctrl :c | c impulses isEmpty not ]

indicates that the following parts of the rule are only to be invoked if the NeuroEmitter being traversed is non-empty. It is supposed that when the list of signals is empty, no point of view is to be activated. In other words, a neuro-emitter can only be analyzed if it has signals to be transmitted. This arbitrary restriction can be modified later if need be. Furthermore, given the way that this rule was written, nothing prevents the signals to be transmitted from being calculated during the neuro-emitter's activation.

The point of view activation block describes the message that should activate the associated point of view if the activation is authorized by the test block; here the message is activateNeuroEmitter: c. The argument c is the neuro-emitter currently being traversed.

This rule has but one ordering link. It states that the following rule to be activated is the rule called #dendrite. Furthermore, this rule should be applied to each of the elements resulting from the expression [:ctrl :c | c output], which returns the dendrites that have the current neuro-emitter as end.

The #dendrite rule describes the behavior that the controller should follow when traversing a dendrite.

(RuleOrdering named: #dendrite
   if: ‘[:ctrl :c | true]’
   do: ‘[:pv :c | pv activateDendrite: c]’
   nextRule: #neuron
   for: ‘[:ctrl :c | Array with: c origin]’;
   nextRule: #neuroReceptor
   for: ‘[:ctrl :c | Array with: c origin]’
)

The activation test block is the same as for the #open rule. It means that all the phases of the current rule are activated. This choice is safe, since the selection of dendrites to be activated is already made in the #neuroEmitter or #neuron rules.

The point of view activation block states that the activateDendrite: c message should be sent to the associated point of view. The argument is the dendrite currently being traversed.

This rule has two ordering rules, which in this particular case correspond to two choices. At this stage of the traversal, it is impossible to know whether the dendrite being traversed belongs to a neuron or to a neuro-receptor. The two cases
are therefore taken into account; the ordering of the links has no effect in this situation.

Each of the two links states which is the following rule to be activated (#neuron or #neuroReceptor), and that that rule will be applied to the unique element resulting from evaluating the expression:

\[
[:\text{ctrl} :c \mid \text{Array with: c origin}]\]

The choice will be resolved in the #neuron and #neuroReceptor rules, which authorize their activation only if the object being traversed is of the right class.

The #neuron rule defines the controller's behavior when traversing a neuron.

\[
\begin{align*}
\text{(RuleOrdering named: #neuron} & \\
\text{if:} & \quad \text{[[:\text{ctrl} :c \mid (c \text{ isKindOf: Neuron) and:}]}
\quad [c \text{ entriesWereActivated}]]' & \\
\text{do:} & \quad [[:\text{pv} :c \mid \text{pv activateNeuron: c}']] & \\
\text{nextRule: #dendrite} & \\
\text{for:} & \quad [[:\text{ctrl} :c \mid c \text{ output}]]
\end{align*}
\]

The activation test

\[
[:\text{ctrl} :c \mid (c \text{ isKindOf: Neuron) and: } [c \text{ entriesWereActivated}]]\]

states that the object being traversed during the rule's activation must be an instance of class Neuron (to resolve the conditional created in the #dendrite rule) and allows the activation only if the neuron's dendrites are all activated. This test is made by the inputsWereActivated method, defined in class Neuron.

!Neuron methodsFor: 'testing'!
inputsWereActivated
"Return true if each input dendrite was activated,"
"false otherwise"
~ (dendrites detect: [:d | d wasActivated not]
   ifNone: [nil])
   isNil!!

The inputsWereActivated method uses the wasActivated method, which informs whether the dendrite receiving the message was activated by the controller. This behavior is defined for all ObjectImpulsive instances.

!ObjectImpulsive methodsFor: 'testing'!
wasActivated
"Indicate if the receptor was active"
~ state notNil!!
The point of view activation block states that the activateNeuron: c message must be sent to the associated point of view. The argument c is the neuron being traversed.

This rule contains a unique ordering link, which states that the next rule to be activated is called #dendrite and that it is to be applied to the elements resulting from the evaluation of the expression

```
[:ctrl :c | c output]
```

which returns the dendrites connected to the branches of the current neuron’s axon.

The #neuroReceptor rule describes the controller’s behavior when traversing a neuro-receptor.

```
(RuleOrdering named: #neuroReceptor
         if: '[:ctrl :c | c iskindOf: NeuroReceptor ]'
         do: '[:pv :c | pv activateNeuroReceptor: c]')
```

The activation test block

```
[:ctrl :c | c iskindOf: NeuroReceptor]
```

simply checks that the object being traversed is in fact a NeuroReceptor, in order to resolve the conditional created by the #dendrite rule.

The point of view activation block states that the activateNeuroReceptor: c transmission should be sent to the current point of view. The argument is the current neuro-receptor.

This rule has no ordering links, so it is a leaf of the ordering graph. Hence a neuro-receptor finishes, in all cases, traversal of a neural net.

A leaf of the ordering graph is necessarily terminal. A rule that is not a leaf can still be terminal if the activation test forbids the rule’s activation.

The `ControllerNetNeural` class

Now that the ordering steps of a neural net’s traversal have been defined, the class `ControllerNetNeural` can be created, along with its traversal mechanism. This mechanism subsumes the general concept of controller (`ControllerMPVC`).

```
ControllerMPVC subclass: #ControllerNetNeural
    instanceVariableNames: '
    classVariableNames: '
    poolDictionaries: '
    category: 'MPVC Neuron!'
```

The `initialize` method is redefined locally: it describes the specific ordering for this controller. The description consists of all the rules defined above.
ControllerNetNeural methodsFor: 'initialization'
initialize
"Initialization of the dictionary of ordering rules"
self
  addRule:
    (RuleOrdering named: #open
      if: '\[:ctrl :c | true]\'
      do: '\[:pv :c | ]\')
    nextRule: #neuroEmitter
    for: '\[:ctrl :c | c inputs]\'
  addRule:
    (RuleOrdering named: #neuroEmitter
      if: '\[:ctrl :c | c impulses isEmpty not]\'
      do: '\[:pv :c | pv activateNeuroEmitter: c]\')
    nextRule: #dendrite
    for: '\[:ctrl :c | c output]\'
  addRule:
    (RuleOrdering named: #dendrite
      if: '\[:ctrl :c | true]\'
      do: '\[:pv :c | pv activateDendrite: c]\')
    nextRule: #neuron
    for: '\[:ctrl :c | Array with: c origin]\'
    nextRule: #neuroReceptor
    for: '\[:ctrl :c | Array with: c origin]\'
  addRule:
    (RuleOrdering named: #neuron
      if: '\[:ctrl :c
        (c isKindOf: Neuron)
        and: [c entriesWereActivated]\'
      do: '\[:pv :c | pv activateNeuron: c]\')
    nextRule: #dendrite
    for: '\[:ctrl :c | c output]\'
  addRule:
    (RuleOrdering named: #neuroReceptor
      if: '\[:ctrl :c | c isKindOf: NeuroReceptor ]'
      do: '\[:pv :c | pv activateNeuroReceptor: c]\')

Ordering graph editor
The Smalltalk development environment was augmented by adding an ordering graph editor. It can be used to enter, graphically, the ordering graph of a controller and thereby to generate the associated controller, as well as the initialize method specific to that class.
When the editor is started up, the name of the controller whose ordering graph is to be described must be given. Should the controller already exist, the editor decompiles the initialize instance method of that class and recomposes the graphical representation of the graph described by that method.

Otherwise a new graph is initialized with an open rule (see Figure 8.22). Therefore, the editor can be used for creating new controllers or for reading and modifying already existing controllers. The designer then begins by describing the ordering graph. This is done using a popup menu, offering choices of (1) creating a new rule; (2) creating a link between two rules; (3) compiling the graph into a class and a compile; (4) decompiling an already existing controller class; (5) refreshing the graph; or (6) renaming the controller being described.

![Figure 8.22 Editing a new controller](image)

The first phase consists of creating and naming the ordering rules that make up the graph (see Figure 8.23). The designer then describes the order between the different rules by creating the appropriate links (see Figure 8.24). Each link is numbered by its position in the rule’s ordered collection of links.

Once the graph is graphically represented, the functional specifications, i.e. the expressions used in the mechanism’s description, must be given. This is done through a menu option allowing the inspection of the instance variables of the rules and links (see Figure 8.25).

The designer can take the default expressions and modify them as need be (see Figure 8.26).

By pointing with the mouse at an element in the graph and by pressing a mouse button, the user can look at the expressions attached to the rules and links (see Figure 8.27).

The last step in this graphical description consists of compiling the different specifications, graphical and functional (see Figure 8.28). This compilation creates a new class, subclass of ControllerMNPC, and defines its initialize method.
The kernel of a Model-Point of View-Controller system

Figure 8.23 Adding a rule

Figure 8.24 Adding a link
Figure 8.25 Examining attributes

Figure 8.26 Modifying a block
Figure 8.27 Examining expressions

Figure 8.28 Compiling the graph
8.3 Defining a point of view

Defining a point of view constitutes the third step in defining an MPVC system. This step describes the specific part of the decomposed algorithm, i.e. the part that cannot be factored out during the horizontal factorization of several algorithms traversing a model in the same manner (see Figure 8.29).

![Figure 8.29 Horizontal decomposition](image)

This specific part is called the evaluation semantics of the algorithm. It augments the traversal of the model described by the controller, giving it a particular semantics, in the same way as the semantic analysis augments the syntactic analysis of a semiotic function over component tokens.

8.3.1 Point of view

A point of view is split up into local evaluation semantics, defined as behaviors to be adopted by different steps of the traversal of the model by the controller.

These behaviors have already been named. During the description of the controller's ordering graph, the second part of each rule, the point of view's activation, explicitly states what message must be sent to the point of view by the current controller in order for the evaluation semantics for that step to be executed.

Consequently, defining a point of view consists of grouping together and defining methods for the messages that will be sent to the point of view by the controller. These methods describe a particular interpretation of the model's traversal.

Each new point of view, defined according to a controller, has its own set of methods. In order to resolve the resulting overloading of operators that occurs for the selectors in the point of view activation messages when several points of views are defined, each set of methods is grouped into a new class. The latter only contains these methods and those methods that they need.

A point of view models a particular interpretation. As a result, an object – in the object-oriented programming meaning – is not just 'an abstraction of an element of the real world' [157], but can also be the model of a reasoning process, of a computation algorithm or of an activity.
Creating a new point of view consists of creating a new point of view class, then defining the methods corresponding to the methods described by the ordering graph of the controller that activated the point of view.

The `PointFOfViewMPVC` class
This abstract class includes the basic characteristics and behaviors that must be in a class for an evaluation semantics, i.e. any point of view class.

Figure 8.30, which describes the different kinds of link in the MPVC trilogy, shows that there exists a consultation kind of link between the point of view and its model.

![Figure 8.30 MPVC trilogy](image)

This link is in fact the only characteristic that is common to all points of view. The point of view part of an algorithm is the most specific and the most specialized part in the MPVC formalization. It is intimately tied to the activating controller and to the underlying semantics.

The `PointFOfViewMPVC` class, root of the inheritance graph for points of view, is created as follows.

```
Object subclass: #PointFOfViewMPVC
    instanceVariableNames: 'model'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'System-MPVC'
```

Upon initialization of the point of view, the instance variable model becomes the model traversed by the point of view's controller.

```
!PointFOfViewMPVC class
    methodsFor: 'instantiation-initialization'
        model: aModel
        ^self new model: aModel !

!PointFOfViewMPVC methodsFor: 'initialization'
```
model: aModel
"Initialization method for the model"
"linked to the receptor point of view"
model <- aModel !

The EvaluationByInhibition class
The concept of point of view is illustrated by giving the evaluation semantics of an
analysis algorithm for a neural net. The algorithm computes the successive states
of a net's neurons and dendrites.

To begin, a new point of view class, EvaluationByInhibition, is created. It
should inherit the attribute shared by all points of view, so it is a subclass of
PointOfViewMPVC.

PointOfViewMPVC subclass: #EvaluationByInhibition
  instanceVariableNames: '
  classVariableNames: '
  poolDictionaries: '!
  category: 'MPVC-Neuron'!

Next are the methods for the messages used in the activating controller's ordering
graph, here for the controller ControllerNetNeural. Its ordering rules include the
following four messages.

  activateNeuroEmitter:   rule #neuroEmitter
  activateDendrite:      rule #dendrite
  activateNeuron:        rule #neuron
  activateNeuroReceptor: rule #neuroReceptor

The activateNeuroEmitter: method defines what should be done when traver-
sing a neuro-emitter. For this point of view, it simply computes the next state.

  !EvaluationByInhibition methodsFor: 'activation'!
  activateNeuroEmitter: aNeuroEmitter
    "Behavior for the #neuroEmitter stage,"
    "reads the next state in the buffer"
  aNeuroEmitter state: aNeuroEmitter nextImpulse !

This state is obtained by extracting the next impulse from the ordered collection
of impulses to be transmitted by the neuro-emitter.

The activateDendrite: method does the same for a dendrite.

  !EvaluationByInhibition methodsFor: 'activation'!
  activateDendrite: aDendrite
    "Behavior for the #dendrite stage,"
    "reads the current state of Neuron or NeuroEmitter"
  aDendrite state: aDendrite end state !
The next state for a dendrite, be it inhibiting or exciting, is obtained by assigning the state variable the value of the state variable of the neuron or neuro-emitter at the dendrite's source end.

The `activateNeuron` method computes the next state for a neuron, using the inhibiting algorithm, whose definition is recalled here. Let \( N \) be a neuron, \( s \) its threshold, \( e_t \) its state at instant \( t \); if \( N \) emits an impulse, then \( e_t = 1 \), otherwise \( e_t = 0 \). Suppose that \( N \) has \( p \) exciting dendrites \( D\text{Ex}_{i_t} \), \( i = 1 \ldots p \), and \( q \) inhibiting dendrites \( D\text{In}_{i_t} \). Then

\[
e_{t+1} = 1 \iff \sum_{i=1}^{p} D\text{Ex}_{i_t} \geq s \quad \text{and} \quad \sum_{j=1}^{q} D\text{In}_{i_t} = 0,
\]

\[
e_{t+1} = 0 \quad \text{otherwise.}
\]

The algorithm is implemented as follows.

```smalltalk
!EvaluationByInhibition methodsFor: 'activation'
activateNeuron: aNeuron
    "Behavior for the #neuron stage,"
    "reads the current state of the neuron receptor"
    ((aNeuron dendritesInhibiting size = 0)
        add: [aNeuron dendritesExciting size
            >= aNeuron threshold])
    ifTrue: [aNeuron state: 1]
    ifFalse: [aNeuron state: 0].
aNeuron input do: [:d | d resetState] ! !
```

This method uses two new instance methods for the `Neuron` class.

```smalltalk
!Neuron methodsFor: 'access'
dendritesInhibiting
    "Returns the inhibiting dendrites of the receptor neuron"
    ^ dendrites select: [:d | d isInhibiting]!
dendritesExciting
    "Returns the exciting dendrites of the receptor neuron"
    ^ dendrites select: [:d | d isExciting] ! !
```

The `activateNeuroReceptor` method computes the next state from the state of the final dendrite that is input to the neuro-receptor.

```smalltalk
!EvaluationByInhibition methodsFor: 'activation'
activateNeuroReceptor: aNeuroReceptor
    "Behavior for the #neuroReceptor stage,"
    "reads the current state of the input dendrite"
aNeuroReceptor
collectImpulse: aNeuroReceptor input state! !
```
8.3.2 Motor for an MPVC system (version 1)

The pieces must now all be put together. The motor for an MPVC system combines the specifications for a model, a point of view and an ordering graph.

Defining such a mechanism consists of formulating an interpreter for the ordering graph. But this is precisely what the controller was designed for, i.e. the controller is the motor. It should launch and activate the specifications.

The interpreter is defined using two instance methods of class ControllerMPVC. First is the main loop of the interpreter.

```smalltalk
!ControllerMPVC methodsFor: 'activation'
activate: aRule with: anObject
  "Motor for sequential graph traversal.
  anObject -- the current traversed object
  aRule -- the current rule to be applied"
  objectsNext |
  "Phase 1: evaluation of the test block"
  (aRule test value: self value: anObject)
  ifTrue:
    [ "Phase 2: activation of the point of view"
      aRule activity value: pointOfView value: anObject.
      "Phase 3: each of the ordering links is traversed"
      aRule nexts do:
        "Evaluation of the next block"
        [:link | (objectsNexts <-
          link block value: self value: anObject)
          notNil
          ifTrue:
            ['There is at least one object that can be traversed with the following rule"
              objectsNext do:
                [:c | self
                  activate: (graphOrdering at: link rule)
                  with: c]]]]]
```

Argument `aRule` is the ordering rule to be interpreted while argument `anObject` is the element in the model to which the rule should be applied.

The `activate:with:` method defines how an ordering rule is to be interpreted. It includes the three phases of a controller's behavior at a given step in the traversal of the model.

In the first phase, to ensure if the two following stages should be activated, the test block is evaluated, with the current controller and the element being traversed as arguments.

```smalltalk
aRule test value: self value: anObject
```
The second and third phases are placed inside a block, the true branch of a conditional that is sent to the result of the above transmission.

The second phase, which activates the point of view, consists of evaluating the point of view block, with the point of view (instance variable `pointOfView` of the controller) and the current element (anObject) as arguments.

```
aRule activity value: pointOfView value: anObject
```

The last phase activates the rules for the other elements in the model, using two nested loops.

The main loop traverses the ordering links with the current rule as origin.

```
aRule nexts do: [:link | ...]
```

Each of these links (link) consists of a block. Evaluating this block computes the next objects in the model for which the link's rule should be applied. The block is evaluated with the receiving controller (self) and the current object (anObject) as arguments.

```
objectsNexts <- link block value: self value: anObject
```

Should the `objectsNext` collection be non-empty, i.e. there is at least one element for which the link's rule should be activated, then the list is traversed and the interpreter is called recursively, with the rule and the current object as arguments.

```
objectsNexts do:
[:c | self
 activate: (graphOrdering at: link rule)
 with: c]]]!!!
```

The first call to the interpreter, with the initial rule and the starting element, is made by the `open` method.

```
!ControllerMPVC methodsFor: 'activation'!
open
"Method for starting up the ordering graph traversal linked to the receptor controller"
self activate: (graphOrdering at: #open)
with: startConcept!!!
```

So the interpreter is started up with initial rule `#open`, which is supposed to be the entry point for the ordering graph of any controller; the starting element is the contents of instance variable `startConcept`. 
8.3.3 Implementation of a first MPVC system

These mechanisms are illustrated by implementing the MPVC system for a neural net model. The model part is the previously defined XOR neural net (see Figure 8.31).

Once again, here is how the net is created.

```lisp
| nn ne1 ne2 nr n1 n2 n3 n4 |  
"Create the new net"
    nn <- NetNeural new.
"Create the input buffers"
    nn addInput: (ne1 <- NeuroEmitter new).
    nn addInput: (ne2 <- NeuroEmitter new).
"Create four neurons"
    nn addNeuron: (n1 <- Neuron new threshold: 1).
    nn addNeuron: (n2 <- Neuron new threshold: 1).
    nn addNeuron: (n3 <- Neuron new threshold: 1).
    nn addNeuron: (n4 <- Neuron new threshold: 1).
"Create the output buffer"
    nn addOutput: (nr <- NeuroReceptor new).

"Create the dendrites and interconnections
"between neurons and buffers"
    nn addDendrite: "d1"
        (DendriteExciting new end: ne1 origin: n1).
    nn addDendrite: "d2"
        (DendriteInhibiting new end: ne1 origin: n2).
    nn addDendrite: "d3"
        (DendriteExciting new end: ne2 origin: n1).
    nn addDendrite: "d4"
        (DendriteInhibiting new end: ne2 origin: n1).
    nn addDendrite: "d5"
```
(DendriteExciting new end: n1 origin: n3).
nn addDendrite: "d6"
(DendriteExciting new end: n2 origin: n3).
nn addDendrite: "d7"
(DendriteExciting new end: n3 origin: nr).

The point of view here is an instance of class EvaluationByInhibition. It has one instance variable model, inherited from class PointOfViewMPVC, which contains the XOR neural net in the temporary variable nn.

\[ pv <- \text{EvaluationByInhibition model: nn.} \]

In Smalltalk-80, temporary variables, only accessible to a list of instructions, are declared by putting their names between vertical bars; the declaration must precede the actual list of instructions.

The system's controller is an instance of ControllerNetNeural, currently the only way to traverse a neural net. Its initialization requires the model (the neural net nn), the point of view to be activated and the starting point of the model's traversal (here, once again, the model itself).

\[ ctrl <- \text{ControllerNetNeural model: nn pv: pv startConcept: nn.} \]

The traversal of the XOR net computes the successive states of the net's neurons and dendrites.

Of particular interest are the different signals collected by the net's neuro-receptor as a function of the initial values of the net's neuro-emitters. The neuro-emitters are assigned the different possible values for a truth-table with two inputs; after the traversal, the neuro-receptor's buffer should have the successive values of XOR's truth-table.

Here is how to initialize the neuro-emitters with the values for a truth table with two inputs.

\[ ne1 <- \text{impulses: #(0 0 1 1) asOrderedCollection.} \]
\[ ne2 <- \text{impulses: #(0 1 0 1) asOrderedCollection.} \]

The controller in temporary variable ctrl is activated by sending it the message open.

\[ ctrl \text{ open. "Activation of controller"} \]
\[ \text{Transcript show: (nn buffer asArray printString);} \]
\[ \text{cr. ->(0) "Result"} \]

The trace of the activations of the ordering rules for this controller is shown in Figure 8.32. It shows the different

\{ Rule \rightarrow TraversedObject \}

pairs interpreted during the traversal.
1. \{#open -> nn\}
2. \{#neuroEmitter -> ne1\}
3. \{#dendrite -> n1\}
4. \{#neuron -> n1\}
5. \{#neuroReceptor -> n1\}
6. \{#dendrite -> n2\}
7. \{#neuron -> n2\}
8. \{#neuroReceptor -> n2\}
9. \{#neuroEmitter -> ne2\}
10. \{#dendrite -> d4\}
11. \{#neuron -> n1\}
12. \{#dendrite -> d5\}
13. \{#neuron -> n3\}
14. \{#neuroReceptor -> n3\}
15. \{#neuroReceptor -> n1\}
16. \{#dendrite -> d3\}
17. \{#neuron -> n2\}
18. \{#dendrite -> d6\}
19. \{#neuron -> n3\}
20. \{#dendrite -> d7\}
21. \{#neuron -> nn\}
22. \{#neuroReceptor -> nn\}
23. \{#neuroReceptor -> n3\}
24. \{#neuroReceptor -> n2\}

Figure 8.32 Sweep trace of the ordering graph for an XOR net

1. The first rule invoked by the interpreter is rule #open of the activated controller’s ordering graph. This rule is applied to the object referenced by the controller’s startConcept instance variable, namely the XOR net (instance of NetNeural). This evaluation triggers the evaluation of rule #neuroEmitter (2 and 9) for the two neuro-emitters (ne1 and ne2) of the net.
2. Rule #neuroEmitter is applied to the neuro-emitter ne1. Its interpretation activates the associated point of view, instance of EvaluationByInhibition, referenced by the instance variable pointOfView of the controller. This activation takes place by sending the message ActivateNeuroEmitter: to the point of view. Evaluating this point of view extracts the first signal (last element in the ordered collection) contained in the neuro-emitter’s instance variable impulse, in this case, the digit 1, and assigns it to the instance variable state. Once this evaluation is finished, the ordering link provokes the application of the #dendrite rule (3 and 6) for the two dendrites (d1 and d2) connected as output to the neuro-emitter.
3. Rule #dendrite is applied to dendrite d1. It in turn activates the point of view with the activateDendrite: message. As a result, the dendrite’s instance variable state is assigned the current state of the neuro-emitter at its end. The state instance variable therefore contains the value 1, which is the current state of the neuro-emitter. Then the ordering link provokes the interpretation of rules #neuron (4) and #neuroReceptor (5) for the element that holds the dendrite. In this case, the element is the neuron n1, an instance of class Neuron.

4. The #neuron rule is applied to neuron n1. Its activation is immediately halted since the activation test fails: not all the input dendrites have been activated.

5. The #neuroReceptor rule is also applied to neuron n1. Its activation is also immediately halted, but for a different reason: the traversed object is not an instance of the NeuroReceptor class.

6. The #dendrite rule is applied this time to the second dendrite connected to the ne1 neuro-emitter. Activating the point of view is similar to (3): the state variable of dendrite d2 is assigned the current state of neuro-emitter ne1, in this case the value 1. Interpreting the link triggers the application of rules #neuron (7) and #neuroReceptor (8) to neuron n2.

7. For the same reason as in (4), the interpretation of rule #neuron is halted.

8. For the same reason as in (5), the interpretation of rule #neuroEmitter is halted.

9. The rule #neuroEmitter is applied to the second neuro-emitter (ne2). Activating the point of view provokes a transfer similar to the one in (2). The neuro-emitter’s state variable therefore contains, at the end of the point of view’s evaluation, the digit 1, the first signal transmitted by neuro-emitter ne2. Interpreting the ordering rule triggers the application of rule #dendrite (10 and 16) on the two dendrites (d4 and d3) connected to the neuro-emitter's output.

10. Interpreting the #dendrite rule implies copying the tip neuro-emitter’s current state to the current dendrite’s state variable (d4) and applying the rules #neuron (11) and #neuroEmitter (15) to the neuron (n1) containing that dendrite.

11. The #neuron rule is applied in the same context as in (4), i.e. to neuron n1. However, this time the full interpretation can take place, since the two input dendrites have been activated: in (3) for d1 and in (10) for d4. Evaluating the point of view starts up the inhibiting algorithm on the current neuron. The result is that the neuron does not emit any impulse, hence that the neuron’s state variable contains the value 0. The link triggers rule #dendrite for the only dendrite (d5) connected to neuron n1.

12. As for any #dendrite rule, its interpretation copies the current state of the element at the tip of the dendrite. Here the dendrite is d5 and the element is the neuron n1. So the state variable takes value 0. The link triggers the application of rules #neuron (13) and #neuroReceptor (14) to neuron n3.
13. For the same reason as in (4), the interpretation of rule #neuron is halted.
14. For the same reason as in (5), the interpretation of rule #neuroEmitter is halted.
15. This is the dual rule to rule (11) and, for the same reason as for (14), its interpretation is halted.
16. The #dendrite rule is applied to the second dendrite d3, connected to the second neuro-emitter in the net. Evaluating the point of view assigns the value 0 to the dendrite’s state variable. Interpreting the links starts up the rules #neuron (17) and #neuroReceptor (24) with neuron n2.
17. The #neuron rule is applied in the same context as in (7), i.e. to neuron n2. But this time the complete interpretation can take place since both input dendrites have been activated: in (6) for d2 and in (16) for d3. The result of the point of view’s evaluation is that the neuron does not emit an impulse, so its state variable takes the value 0. The ordering link triggers rule #dendrite on the single dendrite d6 connected to the axon of neuron d2.
18. The #dendrite rule is applied to dendrite d6. It copies the current state of the tip neuron to the dendrite’s state variable and applies rules #neuron and #neuroEmitter to neuron n3, which contains this dendrite.
19. Rule #neuron is applied in the same context as in (13), i.e. is applied to neuron n3. But this time, the full interpretation can take place since both input dendrites have been activated: in (12) for d5 and in (18) for d8. The neuron does not emit an impulse and its state variable takes the value 0. Rule #dendrite is applied to dendrite d7.
20. Interpreting the #dendrite rule means copying the tip neuron’s current state into the state variable of the current dendrite d7 (value 0) and applying the rules #neuron and #neuroEmitter to the neuro-receptor nr containing the dendrite.
21. Rule #neuron is applied to the neuro-receptor nr. Its interpretation is halted since the traversed object is not an instance of class Neuron.
22. The rule #neuroReceptor is applied to nr, the only neuro-receptor in the net. Unlike in situations (5), (8), (14) and (15), its interpretation is not halted, since the object being traversed is an instance of class NeuroReceptor. The point of view is activated by the activateNeuroReceptor: message. As a result, the signal transmitted by its input dendrite (d7) is collected, here the value 0.
23. This is the dual rule for rule (19) and, for the same reason as in (14), the interpretation is halted.
24. This is the dual rule for rule (17) and, for the same reason as in (23), the interpretation is halted.

The traversal gives the result of the evaluation of the neural net xor for the last impulse of the two neuro-emitters. The result is contained in the buffer of neuro-receptor nr, the signal 0.
Transcript
show: (ne1 impulses asArray printString); cr.
-> (0 0 1) "ne1's buffer"

Transcript
show: (ne2 impulses asArray printString); cr.
-> (0 0 1) "ne2's buffer"

The controller must be activated three more times for all of the neuro-emitters’ impulses to be transmitted.

"Activation of the controller"
ctrl open.
Transcript
show: (nr buffer asArray printString); cr.
-> (1 0) "Result"

ctrl open.
Transcript
show: (nn buffer asArray printString); cr.
-> (1 1 0) "Result"

ctrl open.
Transcript
tshow: (nn buffer asArray printString); cr.
-> (0 1 1 0) "Result"

The initialization of the neuro-emitters as above and the collection of the signals in the neuro-receptor after the four traversals is shown in Figure 8.33. It is in fact the truth-table for the XOR function.

![Figure 8.33 The XOR truth-table](image)
8.3.4 Controller for a neural net (version 2)

The current implementation of the MPVC is still rather laborious. The system's controller must be invoked as many times as there are signals to be transmitted by the net's neuro-emitters, since it only effects one traversal of the net per activation, hence one traversal of each of the neuro-emitters.

This single traversal is more easily seen in the graphical representation (see Figure 8.34) of the controller's ordering graph: the #open rule invokes the neuro-emitters in the model net only once.

![Diagram of controller net neural](image)

**Figure 8.34** The original ordering graph

This is overcome by modifying the controller in such a way that it retraverses the neuro-emitters of the model so long as they have signals to transmit to the dendrites that are connected to them.

The modification of the controller's ordering graph is done from its graphical representation. This consists of making sure that the #open rule, which provokes a single traversal, is reactivated so long as the net's neuro-emitters still have signals to transmit.

A reflexive link, with origin and end the #open rule, is added (see Figure 8.35). Now the reactivation of rule #open must take place in the same context, i.e. for the same net, the first object traversed by the controller. The next block of the reflexive link must be defined so that it returns the object being traversed by the #open rule. Here is how it is done.

```
[:ctrl :c | Array with: c]
```

This expression turns out, in fact, to be the default expression for the next block when a new link is initialized (see Figure 8.36), so the blockNext attribute need not be changed.
Figure 8.35 Adding a reflexive link

Figure 8.36 The default method
The created loop needs a way of stopping. This is done by redefining the activation test for the #open rule so that it only activates the rule if there exists at least one neuro-emitter with one or more signals to transmit (see Figure 8.37).

Now that the modifications are finished, the graph must be recompiled (see Figure 8.38) in order to update initialize in class ControllerNetNeural. Remember that the initialize method describes the transmissions for creating the ordering graph at the controller's creation time.

Here is the resulting initialize method.

```ruby
initialize

"Initialization of the ordering rule dictionary"
self
addRule:
  ((RuleOrdering named: #open
    if: ‘[:ctrl :c | c isActivable]’
    do: ‘[:pv :c | ]’)
  nextRule: #neuroEmitter
  for: ‘[:ctrl :c | c inputs]’
  nextRule: #open
  for: ‘[:ctrl :c | Array with: c]’)
addRule:
  ((RuleOrdering named: #neuroEmitter
    if: ‘[:ctrl :c | c impulses isEmpty not]’
    do: ‘[:pv :c | pv activateNeuroEmitter: c]’)
  nextRule: #dendrite
  for: ‘[:ctrl :c | c output]’
addRule:
  ((RuleOrdering named: #dendrite
    if: ‘[:ctrl :c | true]’
    do: ‘[:pv :c | pv activateDendrite: c]’)
  nextRule: #neuron
  for: ‘[:ctrl :c | Array with: c origin]’;
  nextRule: #neuroReceptor
  for: ‘[:ctrl :c | Array with: c origin]’)
addRule:
  ((RuleOrdering named: #neuron
    if: ‘[:ctrl :c | (c isKindOf: Neuron)
      and: [c entriesWereActivated]’
    do: ‘[:pv :c | pv activateNeuron: c]’)
  nextRule: #dendrite
  for: ‘[:ctrl :c | c output]’)
```
Figure 8.37 Modifying the activation test

Figure 8.38 Recompiling the graph
addRule:
  (RuleOrdering named: #neuroReceptor
   if: '[:ctrl :c | c isKindOf: NeuroReceptor ]'
   do: '[:pv :c | pv activateNeuroReceptor: c]').

The isActivable method, used in rule #open, returns the Boolean value true if the neural net receiving the message has at least one neuro-emitter with at least one signal to transmit.

!NetNeural methodsFor: 'testing'!

isActivable
  "Test if the message receptor has at least one neuro-emitter with at least one signal to transmit"
  self inputs
  do: [:ne | ne impulses isEmpty ifFalse: [^ true]].
  false!

And now for the implementation of the test. In the discussion below, the temporary variable nn continues to refer to the XOR neural net.

Creating and initializing the point of view consists of evaluating the following transmission.

pv <- EvaluationByInhibition model: nn.

The controller is still created in the same manner.

ctrl <- ControllerNetNeural

The two neuro-emitters of the net must be reinitialized to the truth-table inputs.

"Initialization of the NeuroEmitters"

ne1 impulses: #(0 0 1 1) asOrderedCollection.
ne2 impulses: #(0 1 0 1) asOrderedCollection.

The activation of the new controller is started by the ctrl open transmission. After the first activation, the contents of the net's neuro-receptor have received all the values.

Transcript
  show: (nr buffer asArray printString); cr.
  -> (0 1 1 0) "Result"

and the impulse variables of the neuro-emitters have transmitted all their signals.

Transcript
  show: (ne1 buffer asArray printString); cr.
  -> () "ne1's result"
The result is still an XOR truth-table, but this time the controller had to be invoked only once.

8.3.5 Evaluation by subtraction

The preceding subsection showed how to modify the controller part of an MPVC system. Implementing these modifications required no changes to the point of view, since the modifications in no way put into question the traversal mechanism of a neural net, nor the evaluation semantics of the traversal.

Here the MPVC system is modified by augmenting the example with a second point of view. The EvaluationByInhibition point of view defines an evaluation semantics computing the successive states of the net's neurons and dendrites, using the inhibiting algorithm, which formalizes the fact that an inhibiting dendrite that emits an impulse systematically inhibits the emission of impulses by the containing neuron, no matter the latter's threshold and the states of the other dendrites. This absolute inhibition is only one form presented by Minsky [130]. The other, the subtractive inhibition, offers another point of view for the computation of the neuron's next state.

**Definition 11.** A neuron transmits an impulse at instant \( t + 1 \) if and only if at instant \( t \) the difference between the number of exciting dendrites transmitting an impulse and the number of inhibiting dendrites transmitting an impulse is greater than or equal to the threshold.

Subtractive inhibition was used by McCulloch and Pitts [125] to model neural nets where knowledge was protected against certain kinds of flux in the neuronal thresholds; the basic question was how a machine can remain trustworthy despite these fluctuations.

Here is the subtractive algorithm. Let \( N \) be a neuron, \( s \) its threshold, \( e_t \) its state at instant \( t \); if \( N \) emits an impulse, then \( e_t = 1 \), otherwise \( e_t = 0 \). Suppose that \( N \) has \( p \) exciting dendrites \( DEx_i, i = 1 \ldots p \), and \( q \) inhibiting dendrites \( DNi_j \). Then

\[
e_{t+1} = 1 \quad \text{iff} \quad \left( \sum_{i=1 \ldots p} DEx_i - \sum_{j=1 \ldots q} DNi_j \right) \geq s,
\]

\[
e_{t+1} = 0 \quad \text{otherwise}.
\]

Implementing this new point of view implies implementing a new point of view class. Like any point of view of the neural net model, any instance of this class must be able to respond to the messages `activateNeuroEmitter`, `activateDendrite`,

Transcript
show: (ne2 buffer asArray printString); cr.
-> () "ne2's result"
activateNeuron: and activateNeuroReceptor: that can be sent by the activating controller.

For this particular point of view, only the behavior for the activateNeuron: message differs from the behaviors specified for the EvaluationByInhibition class. Consequently, a vertical factorization of the two points of view is made.

The abstract class EvaluationNetNeural factors out the common behaviors of the two points of view; it is a subclass of PointOfViewMPVC and superclass of the two points of view.

```
PointSizeOfViewMPVC subclass: #EvaluationNetNeural
  instanceVariableNames: ";
  classVariableNames: ";
  poolDictionaries: ";
  category: 'MPVC-Neuron'
```

It provides definitions for the three methods that are common to the two points of view.

```
!EvaluationNetNeural methodsFor: 'activation'!
activateNeuroEmitter: aNeuroEmitter
  "Behavior for the #neuroEmitter step,
   reads the next state in the buffer"
  aNeuroEmitter state: aNeuroEmitter nextImpulse!
activateDendrite: aDendrite
  "Behavior for the #dendrite step,
   reads the current state in Neuron or NeuroEmitter"
  aDendrite state: aDendrite end state!
activateNeuroReceptor: aNeuroReceptor
  "Behavior for the #neuroReceptor step,
   collects the current state for the input dendrite"
  aNeuroReceptor collectImpulse: aNeuroReceptor input state! !
```

The activateNeuron: method is also defined at this level, but all it does is declare that each subclass must define it.

```
!EvaluationNetNeural methodsFor: 'activation'!
activateNeuron: aNeuron
  "Behavior for the #neuron step,
   to be defined by the subclasses"
  self subclassResponsibility! !
```

Once this factorization has taken place, EvaluationByInhibition is recompiled as a subclass of the abstract class EvaluationNetNeural.
The kernel of a Model-Point of View-Controller system

EvaluationNetNeural subclass: #EvaluationByInhibition
  instanceVariableNames: '
  classVariableNames: '
  poolDictionaries: '
  category: 'MPVC-Neuron'

The only method that is local to EvaluationByInhibition is activateNeuron:, which implements the inhibiting algorithm.

!EvaluationByInhibition methodsFor: 'activation'!
activateNeuron: aNeuron
  "Behavior for the #neuron step, computes the next state of the receiving neuron"
  (aNeuron dendritesInhibiting size = 0) and:
  [aNeuron dendritesExciting size >= aNeuron threshold])
  ifTrue: [aNeuron state: 1]
  ifFalse: [aNeuron state: 0].
  aNeuron state do: [:d | d resetState]! !

The EvaluationBySubtraction class is the second factored class.

EvaluationNetNeural subclass: #EvaluationBySubtraction
  instanceVariableNames: '
  classVariableNames: '
  poolDictionaries: '
  category: 'MPVC-Neuron'

Its activateNeuron: implements the subtracting algorithm.

!EvaluationBySubtraction methodsFor: 'activation'!
activateNeuron: aNeuron
  "Behavior for the #neuron step, computes the next state of the receiving neuron"
  (((aNeuron dendritesExciting size) -
    (aNeuron dendritesInhibiting size))
  >= aNeuron threshold)
  ifTrue: [aNeuron state: 1]
  ifFalse: [aNeuron state: 0].
  aNeuron state do: [:d | d resetState]!!

This new class enriches the example system with a new point of view for the traversal of a neural net by the controller defined in ControllerNetNeural. Apart from the vertical factorization that was necessary in this case, implementing a new new point of view simply means creating a new class regrouping the methods that are specific to the new algorithm.

Activating this point of view on the XOR neural net example gives the same results as for the EvaluationByInhibition point of view, i.e. the exclusive or
truth-table. This is because the neuronal configuration in Figure 8.39 gives the same result for the two algorithms.

This partial equivalence can be summarized. Let \( N \) be a neuron, \( s \) its threshold. Suppose that \( N \) has \( p \) exciting dendrites \( \text{DExi}_i \), \( i = 1 \ldots p \), and \( q \) inhibiting dendrites \( \text{DIni}_j \). Then the subtracting inhibition applied to neuron \( N \) is equivalent to the absolute inhibition applied to the same neuron if and only if

\[
e_{t+1} = 1 \quad \text{iff} \quad \sum_{i=1}^{p} \text{DExi}_i = s \quad \text{and} \quad \sum_{j=1}^{q} \text{DIni}_j = 1.
\]

Figure 8.40 shows two neuronal configurations where the two points of view differ.

### EvaluationBySubtraction

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0001111</td>
</tr>
<tr>
<td>0101</td>
<td>00110011</td>
</tr>
<tr>
<td></td>
<td>01010101</td>
</tr>
</tbody>
</table>

### EvaluationByInhibition

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0001111</td>
</tr>
<tr>
<td>0101</td>
<td>00110011</td>
</tr>
<tr>
<td></td>
<td>01010101</td>
</tr>
</tbody>
</table>

Figure 8.40 Different configurations

### 8.3.6 Graphical animation

This subsection presents a new evaluation semantics; it allows the graphical animation of the graphical representation of the neural net.

The Smalltalk development environment has been augmented with a neural net graphical editor. Creating these nets resembles the three steps involved in creating an ordering graph for a controller. First is the creation of the nodes (neurons, neuro-emitters and neuro-receptors) in the net (see Figure 8.41). Second is the creation of the arcs (inhibiting and exciting dendrites – see Figure 8.42).

Activating the new point of view is the third step. Once the graphical representation has been finished, its graphical animation can be invoked. It visualizes the activation of the nodes and arcs traversed by the controller (see Figure 8.43).
The kernel of a Model-Point of View-Controller system

**Figure 8.41 Adding a node**

**Figure 8.42 Adding a dendrite**

**Figure 8.43 Graphical animation**
Here is the definition of the AnimationNetNeural class.

```
PointSizeMPVC subclass: #AnimationNetNeural
    instanceVariableNames: '
        classVariableNames: '
    poolDictionaries: '
    category: 'MPVC-Neuron'
```

The four following methods define local evaluation semantics, all specific to this point of view.

```
!AnimationNetNeural methodsFor: 'activation'
activateNeuroEmitter: aNeuroEmitter
    "Animation of a neuro-emitter"
    aNeuroEmitter flash!
activateDendrite: aDendrite
    "Animation of a dendrite"
    aDendrite path!
activateNeuron: aNeuron
    "Animation of a neuron"
    aNeuron flash!
activateNeuroReceptor: aNeuroReceptor
    "Animation of a neuro-receptor"
    aNeuroReceptor flash! !
```

These methods use two graphical animation messages. The flash message quickly inverts, several times, each of the pixels of the icon for a neuron, a neuro-emitter or a neuro-receptor receiving a message. The path message makes an icon travel along the receiving dendrite. The icon represents the state, more precisely the type of signal (0 or 1).

Executing this new point of view triggers the graphical animation of the path taken by the controller as it traverses the neural net.

The description of the kernel of the object-oriented environment for MPVC systems is now complete. The next chapter describes the different properties of and extensions for this kernel.
According to Ginguay and Lauret [73], an algorithm is ‘a set of operating rules defining a finite sequence of operations to be undertaken to obtain the solution to a problem’. These operating rules are organized in a hierarchical manner, based on the algorithm’s functional decomposition.

The MPVC model perceives algorithms in another context. Its horizontal decomposition provides a global understanding. Before undergoing a particular functional decomposition, the algorithm is understood to be the symbiosis between a mechanism of traversing the data, an evaluation semantics and a given structural model.

By this change of reference [186], the MPVC system’s designer works at a much higher level of abstraction than the algorithm itself. The algorithm becomes an object in the same manner as the objects that it manipulates.

This chapter describes a number of properties, related to and induced by the MPVC decomposition, which can be understood as extensions of the previously defined kernel. Explaining their implementation will present a more synthetic view of the software prototype being developed.

9.1 Multiple points of view

9.1.1 Definition

Multiple points of view correspond to the need to effect several interpretations in parallel, without having having to retrace the model for each of them. An MPVC system, as described up to now, requires a different traversal for each interpretation. This restriction is tied to the fact that the motor for an MPVC controller only calls upon a single point of view when it is active, namely the value of the pointOfView instance variable.

Factorizing the traversal allows the execution of several interpretations during
the traversal. This is done through the encapsulation of the points of view for the
different interpretations in a global point of view, called a multiple point of view.

**Definition 12.** Let \( P_1, \ldots, P_n \) be \( n \) evaluation semantics for a model \( M \) and let \( C \)
be a mechanism for a traversal of model \( M \). Then the encapsulation of \( P_1, \ldots, P_n \)
is the **multiple point of view** \( \Pi \):

\[
\{C, \Pi(P_1, \ldots, P_n), M\}.
\]

This formulation shows that the main aspects of a multiple point of view are that (1) it is a point of view, like those that it encapsulates and (2) it is activated by the controller of the points of view that it encapsulates.

### 9.1.2 Modeling

Modeling multiple points of view relies on the two above aspects.

**The MultiPV class**

Since a multiple point of view is itself a point of view, it means that the class MultiPV that is going to model this kind of encapsulation must inherit the characteristics shared by all points of view, hence must be a subclass of PointOfViewMPVC, the root of the inheritance graph of points of view. Furthermore, a multiple point of view consists of the collection of points of view that it encapsulates.

```powershell
PointOfViewMPVC subclass: #MultiPV
    instanceVariableNames: 'pvEnvelopes'
    classVariableNames: '
    poolDictionaries: ''
    category: 'system-MPVC'
```

The pvEnvelopes variable is initialized to the collection of instances of points of view that it encapsulates. Here is the initializing method.

```powershell
!MultiPV class methodsFor: 'instantiation-initialization'
    model: aModel for: aCollectionOfPV
    ^ (self model: aModel)
    pvEnvelopes: aCollectionOfPV!!
```

It uses the class method model:, defined in the superclass PointOfViewMPVC, to initialize the model for the point of view, and the pvEnvelopes:, which initializes the pvEnvelopes instance variable.

```powershell
!MultiPV methodsFor: 'access'
    pvEnvelopes: aCollectionOfPV
    "Initialization of the collection"
of encapsulated pvs"

pvEnvelopes <- aCollectionOfPV.

pvEnvelopes do: [:pv | pv model: model]!!

In addition to initializing the instance variable, this method gives to each of the
encapsulated points of view the same model, the same one as for the multiple
points of view.

Consequently, creating an MPVC trilogy containing a multiple point of view as
point of view differs slightly from the previously described creations. Only the
transmission corresponding to the point of view is different: it now contains a
second argument describing the points of view to be encapsulated.

In fact, this transmission uses the class method model:for: of class MultiPV.
The two arguments are the model of the created trilogy and the collection of
encapsulated points of view.

Hence creating a multiple point of view encapsulating an **evaluation by inhibition**
point of view and a **graphical animation** point of view would be done as follows.

MultiPV model: nn
for: (Array with: EvaluationByInhibition new
  with: AnimationNetNeural new)

**Point of view ordering**

Since the multiple point of view is activated by the same controller as the individual
points of view, and since it is a point of view in the MPVC trilogy, it receives all
the point of view activation messages defined in the controller’s ordering graph.

Consequently, the multiple point of view must be capable of responding to all
of the messages, i.e. it must be able to trigger, upon reception of a message, all of
the behaviors that each of the encapsulated points of view would have had.

But for the moment an instance of class MultiPV does not know how to respond
to any of these messages. This fact, relevant to the implementation of multiple
points of view, is used to allow multiple interpretations. The messages sent by the
controller to the multiple point of view are redirected using a standard Smalltalk
technique, by catching the exception doesNotUnderstand:, predefined at the class
level [20, 63, 143].

When an object does not know how to handle a message, Smalltalk sends it the
transmission doesNotUnderstand: aMessage, indicating the message that triggered
the exception. The default method for message doesNotUnderstand:, defined in
the Object class, is to open a NotifyView window over the context in which the
error took place. Here is a simplified method.

!Object methodsFor: ‘error handling’!

doesNotUnderstand: aMessage
  NotifierView
  openContext: thisContext
  label: ‘Message not understood:’, aMessage selector
contents: thisContext shortStack.
"Try the message again if the programmer
decides to proceed"
~ self perform: aMessage selector
  withArguments: aMessage arguments!!

This method can be redefined locally in a class in order to define a particular behavior for a class.

Catching the doesNotUnderstand: exception is an inappropriate tool for modifying the standard Smalltalk transmission mechanism, to better control its behavior and to offer new control possibilities in the language. This trap mechanism is often used in prototyping certain aspects of parallel programming [19, 20, 37, 143].

In this particular case, for the implementation of multiple points of view, the doesNotUnderstand: mechanism is the only way to model the behavior of a multiple point of view upon the reception of the messages that its controller sends it.

A multiple point of view does not understand any of the messages sent by the controller, but it does know which are the recipients. Therefore every time that an instance of class MultiPV is invoked by the controller, it is sent the message doesNotUnderstand:; the message is intercepted at the MultiPV class level by a redefinition of the doesNotUnderstand: method, which redirects the message to the encapsulated points of view.

!MultiPV methodsFor: 'redirection'
doesNotUnderstand: aMessage
  "Interception of message and redirection
to encapsulated points of view"
  pvEnvelopes
  do: [:aPV | aPV perform: aMessage selector
    withArguments: aMessage arguments]!!

The resulting MPVC trilogy is shown in Figure 9.1. The advantage of the multiple points of view extension is that its implementation requires no modification of existing points of view, nor of the initial system.

Furthermore, in a traditional approach to programming, if several points of view are to be executed, one of the algorithms must be modified to include the points of view that it does not include. The multiple points of view avoids these insertions and modifications of code, which are generally sources of error.

9.1.3 Restrictions in usage

There are two major restrictions pertaining to the use of multiple points of view.
Figure 9.1 MPVC trilogy (with multiple points of view)

Point of view ordering
The order in which the instances of points of view are encapsulated is of importance. The multiple point of view does not distribute the point of view activation message simultaneously to the different encapsulated points of view but, rather, it activates them sequentially, respecting the order in which they were transmitted during the initialization of the instance variable pvEnvelopes.

Hence, the activation of a multiple point of view having as a first encapsulated point of view an instance of class EvaluationByInhibition and as a second point of view an instance of class AnimationNetNeural does not have the same evaluation semantics as in the case where the instance of class AnimationNetNeural had been placed first. The first case allows a graphical animation of the evaluation of the net. The second case provides a graphical animation of the initial state of the net, since the animation cannot take advantage of the information resulting from the evaluation by inhibition.

Exclusive points of view
The second constraint comes from the fact that the co-existence of encapsulated points of view is not always possible.

Hence, the activation of a multiple point of view encapsulating the two evaluation points of view of the net (by inhibition and by subtraction) would provide inconsistent results. Activating the first point of view will modify the common model, in particular, the state of one of the elements of the traversed model. Activating the second will in turn alter the same state. In the next activation, the first point of view will take the contents of the state, assuming that it was the last to modify the state, when it is not in fact the case. The second point of view prevents the first from working properly, and vice versa.
More generally, algorithms are called exclusive if their respective points of view write into the same structures of their common model.

Of course, this restriction can be lifted by indexing the modifications made to the model and by taking as index keys the actual point of view instances making the modifications. Hence each point of view could have exclusive access, without any ambiguity, to its own modifications.

The subsequent sections often use the concept of multiple points of view. Its ease of use encourages the decomposition of a problem into multiple evaluation semantics and to regroup the latter once they have been perfected.

9.2 Multiple controllers

The independence, in their implementation, between a graph traversal mechanism and an evaluation semantics means that different traversals of the same model can be elaborated and tested for efficiency, by combining them with different points of view. This result, induced by the MPVC decomposition, shows that the traversal mechanism is one of an algorithm's parameters. The controller is a behavioral parameter of the algorithm.

This concept can be found, for example, in algorithms for the analysis of syntax trees; these can be built using different traversal mechanisms, such as bottom-up or top-down, implementing the same evaluation semantics for each of the nodes in a tree.

This concept is illustrated by the implementation of a new controller for the example system.

9.2.1 Cycles in a neural net

Class ControllerNetNeural defines a depth-first traversal of nets. Each controller traverses a net's branch from the neuro-emitters to the dendrites, from the dendrites to the neurons, etc., until encountering either a neuron, none of whose dendrites have been activated, or a neuro-receptor, which stops the traversal.

Such a controller cannot traverse a net with a cycle, i.e. when the state of a neuron depends on the state of a dendrite whose state itself depends on the same neuron. This problem can be seen in Figure 9.2.

The adder is never completely traversed by the neural net controller, version 2, since the activation of nodes $N_1$, $N_2$ and $N_3$ depends on the activation of neuron $N_2$. This dependency, shown in bold in Figure 9.2, comes from the dendrite contained by each of the neurons, connected to the axon of neuron $N_2$. Because of that link, the activation test will systematically be false, since it tests that each of the dendrites of the current neuron has already been activated, which will never be the case for dendrites forming a cycle in the net. The implementation of a new
controller is given below; it uses a traversal mechanism that can properly handle cycles in a neural net.

### 9.2.2 Synchronization and traversal

The new controller assumes the synchronization of activities in the mathematical model for a neural net: 'Each neuron is considered to be a finite automaton operating at discrete instants. These instants are synchronized for all the neurons in a net. At each instant $t$ a neuron emits, or does not, an impulse along its axon.'

The ControllerNetNeural attempts to model one aspect of this synchronization by checking, before the neuron’s activation, that each of its dendrites has been activated. This test is in fact not adequate for nets with cycles. To better understand the idea of synchronization, the traversal must be studied from a temporal point of view. As an example, consider the XOR neural net being traversed during a clock’s ticks (see Figure 9.3).

Each interval of time $\Delta t$ corresponds to a clock tick. Such a traversal gives the following activity sequence:

- at instant $t_0$, the two neuro-emitters in the net are activated;
- at instant $t_0 + \Delta t$, the first two neurons are activated;
- at instant $t_0 + 2\Delta t$, the third neuron is in turn activated;
- at instant $t_0 + 3\Delta t$, the traversal terminates with the activation of the neuro-receptor.

During this traversal, each of the net’s neuro-emitters transmits only once, which is not in accordance with a neuro-emitter’s definition: they are supposed to transmit so long as they still have signals. Hence, at instant $t_0 + \Delta t$, in addition to the two neurons, the two neuro-emitters should be reactivated. Then, at instant $t_0 + 2\Delta t$, in addition to the third neuron being activated, the two neuro-emitters and the two other neurons should all be reactivated. The activation of the two
neurons follows from the activation of the neuro-emitters at instant $t_0 + \Delta t$. And so on.

To simplify the text, it is supposed for the moment that the neuro-emitters transmit only once per traversal. Later it is shown that generalizing to one transmission per impulse in no way modifies the reasoning.

In order to reference the different activation instants of the dendrites of the net, it is supposed that their activation takes place at an instant $\delta t < \Delta t$ after the activation of the neuro-emitter or of the neuron that is connected to its extremity. It follows that

- at instant $t_0 + \delta t$, the four dendrites connected to the two neuro-emitters in the net are activated;
- at instant $t_0 + \Delta t + \delta t$, the two exciting dendrites connected to the axons of the first two neurons are activated;
- at instant $t_0 + 2\Delta t + \delta t$, it is the final dendrite's turn, as it is connected to the final axon in the net.

This temporal decomposition shows that the mathematical model for a neural net does not need to effect any tests to allow a neuron's traversal. The synchronization mechanism is sufficient to traverse the different elements of the net, so that at each moment of their activation, the elements that should already have been activated are activated.

The activation of the first two neurons can take place at instant $t_0 + \Delta t$, since at instant $t_0 + \delta t$ all the dendrites that should have been activated for these neurons are activated. The same holds true for the activation of the last neuron: this can also take place, since at instant $t_0 + \Delta t + \delta t$ the two dendrites of the neuron have been activated.

Consider a similar decomposition for a net with a cycle: the net with memory (see Figure 9.4), described in [130, p. 39]. It controls the flow of impulses circulating
between the input neuro-emitter and the output neuro-receptor. This control takes place by sending an initial signal on start or stop. In the first case, due to the memorization dendrite, the unique signal authorizes the passage of impulses coming from input, so long as no signal is sent from stop. Once this takes place, due to the memorization dendrite, the unique signal inhibits each impulse coming from input.

![Diagram](image)

**Figure 9.4** Firing net with memory

The net is traversed as follows:

- at instant $t$, the three neuro-emitters are activated;
- at instant $t_0 + \Delta t$, it is up to the neurons labeled $n_1$ and $n_2$; neuron $n_1$ considers, during its activation, that the memorization dendrite transmits the signal 0;
- at instant $t_0 + 2\Delta t$, neuron $n_3$ is activated, as is neuron $n_1$; the reactivation of neuron $n_1$ is due to the memorization dendrite, which transmits the state in which neuron $n_1$ found itself after its activation at instant $t_0 + \Delta t$;
- at instant $t_0 + 3\Delta t$, neuro-receptor output is activated and neuron $n_3$ is reactivated, since at the previous instant neuron $n_1$ was activated and it is once again activated;
- at instant $t_0 + 4\Delta t$, all the elements that were activated in the previous instant are reactivated, including neurons $n_1$ and $n_3$ and the neuro-receptor output; the net can loop like this indefinitely.

Using temporal decomposition, efficient traversals can be handled for a net with one or more cycles. Consequently, a controller modeling the above behavior is described below.
9.2.3 Neural net controller (version 3)

Formalization
To model the traversal mechanism for temporal decomposition, the activities that take place between two clock ticks, i.e. during a period $\Delta t$, are formalized below.

Each period consists of two phases. First is the activation of the nodes (neurons, neuro-emitters and neuro-receptors) of the net: all neuro-emitters having at least one signal to transmit are systematically activated; only the neurons and neuro-receptors with dendrites activated during the previous period are activated. Second is the activation of the arcs (exciting and inhibiting dendrites) connected to the previously activated nodes.

Instant $t_0$ begins with the activation of the neuro-emitters in the net.

Ordering graph
The model for the traversal is given by the ordering graph in Figure 9.5. It is a new controller, radically different from previous ones. The periodic activation cycle, described above, is modeled by two ordering links between the rules nodes and arcs. Furthermore, the rules neuroEmitter, neuron and dendrite have become terminal rules: scheduling the traversal has been left to the nodes and arcs rules.

The #open rule is the entry point for the ordering graph.

(RuleOrdering named: #open
  if: ‘[:ctrl :c | true]’ do: ‘[:pv :c | ]’)
nextRule: #nodes
  for: ‘[:ctrl :c | Array with: c inputs]’

The activation test is always verified, so it always allows the rule's activation. The point of view activation block is empty, so no reference to the point of view is made. Finally, there is only one ordering link, to the #nodes rule, which is applied to all the neuro-emitters in the net.
for: `[:ctrl :c | Array with: c inputs]`

The #nodes rules is applied to a collection of nodes in the traversed net.

(RuleOrdering named: #nodes
 if: `[:ctrl :c | c isEmpty not]` do: `[:pv :c | ]`
 nextRule: #neuroEmitter for: `[ctrl :c | c]`; "Links"
 nextRule: #neuron for: `[ctrl :c | c]`
 nextRule: #neuroReceptor for: `[ctrl :c | c]`
 nextRule: #arcs
 for: `[:ctrl :c | Array with: (ctrl outputOf: c)]`

The activation test allows the rule to be activated if the collection is non-empty. The point of view activation block does not have any transmissions either. Neither it nor the #arcs rule call the point of view. These rules serve only as schedulers for the traversal mechanism, and no evaluation semantics takes place during this management process.

The rule has four ordering links. The first three formalize the three kinds of nodes that can be found in the input collection. The fourth link defines the activation period for the traversal mechanism. It refers to the #arcs rule, which is applied to the collection of dendrites that are output to the nodes for the current #nodes rule. This set is computed by evaluating

`ctrl outputOf: c`

where `ctrl` is the current controller, `c` is the current rule and method `outputOf:` is defined at the ControllerNNSynchro class level.

The #arcs rule resembles the #nodes rule. It is activated with a collection of dendrites.

(RuleOrdering named: #arcs
 if: `[:ctrl :c | c isEmpty not]` do: `[:pv :c | ]`
 nextRule: #dendrite for: `[ctrl :c | c]`
 nextRule: #nodes
 for: `[ctrl :c | Array with: ((ctrl originOf: c) asOrderedCollection
 (ctrl model inputs asOrderedCollection
 select: [:ne | ne impulses isEmpty not]))]`

The activation test allows the rule's activation if the transmitted collection is non-empty. The point of view activation block contains no transmissions. The first ordering link refers to rule #dendrite, which activates the evaluation semantics for each dendrite in the collection. The second link loops back to the #nodes rule, which is applied to all the nodes with dendrites resulting from evaluating

`ctrl originOf: c`
and to those neuro-emitters in the net with signals to transmit.

\[
\text{ctrl model inputs} \\
\quad \text{select: } [:\text{ne} \mid \text{ne impulses isEmpty not}] 
\]

The next four rules are all terminal. The \#neuroEmitter rule defines the controller's behavior when traversing a neuro-emitter. The activation test ensures that the object is in fact a neuro-emitter and the point of view activation block gives the message triggering the neuro-emitter's evaluation semantics. The same holds true for rules \#neuron, \#neuroReceptor, and \#dendrite.

(RuleOrdering named: \#neuroEmitter 
  if: ‘[:ctrl :c \mid c isKindOf: NeuroEmitter]’ 
  do: ‘[:pv :c \mid pv activateNeuroEmitter: c]’)

(RuleOrdering named: \#neuron 
  if: ‘[:ctrl :c \mid c isKindOf: Neuron]’ 
  do: ‘[:pv :c \mid pv activateNeuron: c]’)

(RuleOrdering named: \#neuroReceptor 
  if: ‘[:ctrl :c \mid c isKindOf: neuroReceptor]’ 
  do: ‘[:pv :c \mid pv activateNeuroReceptor: c]’)

(RuleOrdering named: \#dendrite 
  if: ‘[:ctrl :c \mid true]’ 
  do: ‘[:pv :c \mid pv activateDendrite: c]’)

The \texttt{ControllerNNSynchro} class

Compiling the ordering graph creates class ControllerNNSynchro as a subclass of ControllerMPVC.

ControllerMPVC subclass: \#ControllerNNSynchro 
  instanceVariableNames: ‘’ 
  classVariableNames: ‘’ 
  poolDictionaries: ‘’ 
  category: ‘MPVC-Neuron’

It also defines method \texttt{initialize}, which describes the ordering graph.

!ControllerNNSynchro methodsFor: ‘initialize’!
  initialize
    "Initialization of the dictionary of ordering rules"
    self 
    addRule: ((RuleOrdering named: \#open 
        if: ‘[:ctrl :c \mid true]’ do: ‘[:pv :c \mid ]’))
nextRule: #nodes
  for: ‘[:ctrl :c | Array with: c inputs]’;
addRule:
  ((RuleOrdering named: #nodes
    if: ‘[:ctrl :c | c isEmpty not]’ do: ‘[:pv :c | ]’)
nextRule: #neuroEmitter
  for: ‘[:ctrl :c | c ]’;
nextRule: #neuron
  for: ‘[:ctrl :c | c ]’;
nextRule: #neuroReceptor
  for: ‘[:ctrl :c | c ]’;
nextRule: #arcs
  for: ‘[:ctrl :c | Array with: (ctrl outputOf: c)]’);
addRule:
  ((RuleOrdering named: #arcs
    if: ‘[:ctrl :c | c isEmpty not]’ do: ‘[:pv :c | ]’)
nextRule: #dendrite for: ‘[ctrl :c | c ]’;
nextRule: #nodes
  for: ‘[:ctrl :c | Array with:
    ((ctrl originOf: c) asOrderedCollection,
    (ctrl model inputs asOrderedCollection
    select: [:ne | ne impulses isEmpty not]))]’);
addRule:
  ((RuleOrdering named: #neuroEmitter
    if: ‘[:ctrl :c | c isKindOf: NeuroEmitter]’
    do: ‘[:pv :c | pv activateNeuroEmitter: c]’);
addRule:
  ((RuleOrdering named: #neuron
    if: ‘[:ctrl :c | c isKindOf: Neuron]’
    do: ‘[:pv :c | pv activateNeuron: c]’);
addRule:
  ((RuleOrdering named: #neuroReceptor
    if: ‘[:ctrl :c | c isKindOf: NeuroReceptor]’
    do: ‘[:pv :c | pv activateNeuroReceptor: c]’);
addRule:
  ((RuleOrdering named: #dendrite
    if: ‘[:ctrl :c | true]’
    do: ‘[:pv :c | pv activateDendrite: c]’))

The #outputOf: method, used by rule #node, returns the collection of all the
dendrites that are output of the nodes in the collection passed as argument.

!ControllerNNSynchro methodsFor: ‘appendix’!
outputOf: aCollectionOfNodes
    | dendrites |
    dendrites <- OrderedCollection new.
    aCollectionOfNodes
do: [:n | (n isKindOf: NeuroReceptor)
    ifFalse: [dendrites <- dendrites, n output]].
    ^ dendrites! !

The method #originFrom:, used by rule #arcs, returns all the nodes containing
the dendrites in the collection passed as argument.

!ControllerWNSynchro methodsFor: 'appendix'
originOf: aCollectionOfNodes
    | nodes |
    nodes <- Set new.
    aCollectionOfDendrites do: [:d | nodes add: d origin].
    ^ nodes! !

The example system has been augmented by a new ordering mechanism described
by class ControllerWNSynchro. Implementing this mechanism did not require any
modification of existing points of view, which is what makes it interesting. Hence
the controller becomes a parameter of the MPVC trilogy, just as are the point of
view and the model.

The point of view is an evaluatory parameter of a trilogy, the model is a structural
parameter and the controller is a behavioral parameter. The only aspect of a
trilogy that is not variable is the motor. In fact, Section 9.3.2, presenting the
meta-controller, shows that even the motor itself can be a parameter, just as are
the components of the MPVC trilogy.

9.3 Parallelism and MPVC

In the preceding sections, the MPVC decomposition was presented as a trilogy, and
each of the different aspects of a tripartite decomposition was addressed. It is now
time to examine the different implementation aspects of each of these parts.

The implementation of the example system shows that the MPVC decomposition
defines a methodology for implementing algorithms. In addition to the model–
point of view–controller decomposition, the MPVC methodology encourages the
decomposition of a point of view into multiple local and autonomous evaluation
semantics and of a controller into multiple ordering rules and links.

Making such decompositions allows one to indentify and to isolate the parts of
an algorithm represented by an MPVC trilogy that must be executed sequentially
and the parts that can be executed in parallel.
9.3.1 Parallel and sequential traversal

As defined in Section 8.2.1, an ordering rule defines the behavior that a controller should have at a given step in the traversal of its model. Each behavior consists of three phases: the activation test, the point of view activation and the ordering of the next activities, called ordering links (see Figure 9.6).

![Sequencing rule]

A set of ordering rules, forming an ordering graph, formalizes the general ordering scheme of the different steps for a given point of view. In reality, the execution of an MPVC trilogy treats two kinds of orderings, orthogonal with respect to each other.

The actual order of the ordering links in a rule determines the unchangeable order in which the rules at the ends of the links must be executed. This first level therefore defines the general sequentiality of the different steps in a model's traversal. This is the generic level of the traversal.

Locally, an ordering link indicates that the rule should be applied to each of the elements in the collection computed during the traversal itself, by evaluating blockNext. This is the specific or individual level of the traversal. The order in which the next rule is applied to the different elements is immaterial, so it can be applied simultaneously, i.e. in parallel, to each of the elements in the collection.

Consider the XOR neural net in Figure 9.7. The concept of orthogonality can be illustrated by graphically representing the trace of the net's traversal by controller ControllerNetNeural (Figure 9.8).

It is therefore easy to deduce the parts of an algorithm that can be started up as independent processes and those that must be executed sequentially. Due to the sequentiality property, it is possible to define a graphical animation point of view that allows the simulation of a parallel traversal of a data structure.
Figure 9.7 Object representation of an XOR neural net

Figure 9.8 Rules and ordering links
9.3.2 Motor for an MPVC system (version 2)

The motor of an MPVC system can be enriched so that it takes into account the above concept of sequentiality. However, to ensure that the user has the choice of running the interpretation in sequential or parallel mode, the ordering mode is treated below as one of the motor's parameters.

Processors
In order to implement the ordering mode as a parameter of the motor, a new concept, the processor, is introduced. There are two kinds of processors: single-tasking or sequential processors, and multi-tasking or parallel processors.

The term processor was chosen in order to allude to the difference between existing computer architectures [160]. The assumed parallel architecture is a SIMD (Single Instruction Multiple Data) processor [80, 81, 160].

The role of the processors is to apply a particular treatment, here the execution of a rule, to different data. For a sequential processor, the algorithm is applied successively to each of the data, while for a parallel processor, the treatment of the data can take place simultaneously. This role is purely functional; modeling a processor makes no reference to any structural attributes. The Processor abstract class, which models the attributes common to the two kinds of processor, is a subclass of Object.

```
Object subclass: #Processor
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'System-MPVC'
```

The exec:for: method, which applies a block of instructions to the elements in a data collection, is first defined at this class level. Each of the subclasses must define the method.

```
!Processor methodsFor: 'execution'
exec: aBlock for: aCollection
"Behavior is defined by subclasses"
self subclassResponsibility!
```

The same holds true for the exec: value: value: method, which executes a block with two arguments.

```
!Processor methodsFor: 'execution'
exec: aBlock value: arg1 value: arg2
"Behavior is defined by subclasses"
self subclassResponsibility!
```
The Processor Sequential class

The Processor Sequential class models the behavior of a single-tasking processor. It is a subclass of Processor.

Processor subclass: #ProcessorSequential
  instanceVariableNames: '"
  classVariableNames: '"
  poolDictionaries: '"
  category: 'System-MPVC'

For single-tasking, the processor can execute only one instruction at a time. Modeling the behavior of such a processor consists of enumerating the data in the collection argument and of successively evaluating the argument block for each of them. The methods exec:for: and exec:value:value: are redefined.

"ProcessorSequential methodsFor: 'execution'!
exec: aBlock for: aCollection
  "Behavior of a single-tasking processor"
  aCollection do: [:data | aBlock value: data]!
exec: aBlock value: arg1 value: arg2
  "Behavior is defined by subclasses"
  aBlock value: arg1 value: arg2! !

The Semaphore Counter class

Implementing the class modeling the behavior of a parallel processor requires a particular kind of semaphore.

Smalltalk-80 provides classes allowing basic multi-tasking programming: class Process models independent processes and class Semaphore allows the synchronization of processes [76, 127]. A Smalltalk process is simply a sequence of instructions that execute a given task. This process can suspend its activity if it wishes to wait for a signal from another process that is running simultaneously. To do this a wait message is sent to a semaphore instance. The latter interrupts the active process, i.e. the process that sent it the wait message, once it receives a signal sent by the other process.

A new class of semaphores is created, in which the instances reactivate the interrupted process only when a precise number (rather than one for instances of Semaphore) of signals has been received. The class SemaphoreCounter is created as a subclass of Semaphore in order to inherit its basic characteristics.

Semaphore subclass: #SemaphoreCounter
  instanceVariableNames: 'counter'
  classVariableNames: '"
  poolDictionaries: '"
  category: 'System-MPVC'
In addition to the inherited characteristics, the new class defines an instance variable counter. The latter is initialized upon creation to a positive integer value, the number of signals that the semaphore must receive before reactivating the interrupted process. *Selector homonymy* is used to define the instantiation-initialization methods for the instance variable counter.

``` Smalltalk
SemaphoreCounter class
    methodsFor: 'instantiation-initialization'
    forSignal: anInteger
        "Instantiation"
        ^ self new forSignal: anInteger!

SemaphoreCounter methodsFor: 'initialization'
    forSignal: anInteger
        "Initialization of a signal counter"
        counter <- anInteger!

To define a behavior that is specific to this kind of semaphore, the methods `wait` and `signal` are redefined locally.

``` Smalltalk
!SemaphoreCounter methodsFor: 'communication'
    wait
        "Interrupt the current process"
        counter > 0 ifTrue: [super wait]!
    signal
        "Reactivation"
        counter <- counter + 1.
        counter <= 0 ifTrue: [super signal]!
```

The `wait` method only interrupts the current process if the number of signals that the receiving semaphore must wait for is greater than zero. The actual interruption is effected by sending the `wait` message to the pseudo-variable `super`, which invokes the `wait` method of the superclass `Semaphore`.

The pseudo-variable `super` refers to the receiver itself, just like the pseudo-variable `self`. However, the strategy is not the same: when a message is sent to `super`, the search strategy begins not in the receiver's class, but in its superclass.

The `signal` method, which formalizes a semaphore's behavior when sending a `signal` to a process, decrements the receiving semaphore's `counter` and reactivates the interrupted process if the counter reaches zero. Reactivating the process is done with the `signal` method of superclass `Semaphore`.

*The ProcessorParallel class*

The `ProcessorParallel` class models the behavior of a multi-tasking processor, which treats the elements in a data collection simultaneously.
Processor subclass: #ProcessorParallel
instanceVariableNames: '"
classVariableNames: '"
poolDictionaries: '"
category: 'System-MPVC'

Since the Smalltalk-80 virtual machine [76] has only one processor, parallelism is simulated in Smalltalk using the Process class. Each evaluation of the argument block over an element in the data collection is considered to be a separate process, instance of the Process. There are therefore as many processes as there are elements in the collection.

!ProcessorParallel methodsFor: 'execution'!
  exec: aBlock for: aCollection
    "Behavior of a multi-tasking processor"
    | semaphore |
    semaphore <- SemaphoreCounter forSignal: aCollection size.
    (aCollection
      collect: [:data |
        [:bl :dt | bl value: dt. semaphore signal]
        copy newProcessWith:
          (Array with: aBlock copy with: data)]
      do: [:process | process resume].
    semaphore wait!
  exec: aBlock value: arg1 value: arg2
    "Behavior of a multi-tasking processor"
    aBlock copy value: arg1 value: arg2! !

Notice that each handling block

[bl value: dt. semaphore signal]
is copied before being linked to the process that must execute it. This kind of behavior can be found in SIMD architectures [160], where computation units are replicated. This is done since a Smalltalk block (BlockContext) has a unique context, including the block ordinal counter, the method containing the block, and the method receiver, which is determined when the block is compiled. However, it is necessary to have a context per process, in particular to have an ordinal counter per process.

Like its dual for class ProcessorSequential, the exec:for: method only returns control once all the calculations have been done. The synchronization of a number of processes and of their parent process takes place through a SemaphoreCounter, which waits for a signal message, sent at the end of each treatment, from each of the child processes.
**The motor (version 2)**

The kind of processor used by a controller for interpreting an ordering rule is a behavioral characteristic of the controller, as is the point of view that it activates. Consequently, the ControllerMPVC class must be recompiled with a new argument.

```
Object subclass: #ControllerMPVC
    instanceVariableNames: 'pointOfView model startConcept
                           graphOrdering processor'
    classVariableNames: '"
    poolDictionaries: '"
    category: 'System-MPVC'
```

The new instance variable `processor` indicates the kind of processor to be used for interpreting the ordering rules in the current controller ordering graph.

In fact, the concept of processor is only temporary. It is replaced in Section 9.4, which presents the meta-controller concept, by two meta-points of view.

The main loop of the interpreter of the ordering rules must be redefined as well. The evaluations of the different blocks composing an ordering rule are left to the associated processor; during a parallel traversal, the blocks must be duplicated in order to avoid different processes from evaluating the same block at the same time.

In addition, when the ordering rules are being interpreted, it is possible to choose between a sequential and a parallel traversal of the model. The selection of traversal takes place by an indirection in the interpretation of rule

```
[:c | self activate: ruleNext with: c]
```

for each of the `objectsNext` to be traversed. This indirection is made via the processor referenced by the processor instance variable.

```
!ControllerMPVC methodsFor: 'activation'
    activate: aRule with: anObject
    "Motor for traversing the ordering graph.
    anObject -- current traversed object
    aRule -- the current rule to be applied"
    | objectsNext ruleNext |
    "Phase 1: evaluation of the test block"
    (processor exec: aRule test value: self value: anObject)
    ifTrue:
    [ "Phase 2: activation of the point of view"
        processor exec:
        aRule activity value: pointOfView value: anObject.
        "Phase 3: each of the ordering links is traversed"
        aRule nexts do:
        "Evaluation of the next block"
        ]
```

[:ls | (objectsNext <- processor exec:
ls block value: self value: anObject)
notNil
ifTrue:
[ "There is at least one object that
can be traversed with the following rule"
ruleNext <- (graphOrdering at: ls rule).
processor exec:
[[:c | self
activate: (graphOrdering at: ls rule)
with: c]
for: objectsNext]]]!!

The method that starts up the interpreter takes a processor as argument.

!ControllerMPVC methodsFor: 'activation'!
openWith: aProcessor
 "Startup the ordering graph traversal
 linked to the receptor controller"
processor <- aProcessor.
self activate: (graphOrdering at: #open)
with: startConcept!!

From now on, activating the controller requires that the kind of processor used
for interpreting the ordering graph be specified. Hence if one wishes to run a
graphical animation of an evaluation by inhibition of a neural net, and to have
a synchronous view by simulating a real-time mechanism of these points of view,
then creating and activating the resulting MPVC trilogy is specified as follows.

"Create a multi-point of view"
pv <- MultiPV model: nn
   for: (Array with: EvaluationByInhibition new
          with: AnimationNetNeural new).

"Create the controller"
ctrl <- ControllerNNSynchro
"Initialize the NeuroEmitters of the Neural Net Model"
ne1 impulses: #(0 0 1 1) asOrderedCollection.
ne2 impulses: #(0 1 0 1) asOrderedCollection.
"Activation of the trilogy with a parallel processor"
ctrl openWith: ProcessorParallel new
9.4 Reflexivity and MPVC

Reflexivity is the ability of a system to model itself by describing its own representation, in order to intervene on the representation's representation and its execution mechanism, and on the reasoning's reasoning.

Reflexivity provides a meta-view of the system that it models. Smith [169] and Pitrat [147] present reflexivity as a means for producing intelligent programs capable of reasoning about their own behavior and that can, therefore, modify it.

Minsky [132, p. 59] describes reflexivity as the examination of brain A's behavior by brain B, where A and B can be parts of the same real brain (see Figure 9.9). Minsky considers that brain B must learn to play the role of advisor or psychologist, who can judge the mental strategy of a client without handling all the details of his or her profession.

![Figure 9.9 Minsky's reflexivity](image.png)

This kind of abstract and symbolic understanding of an analysis method, for example of a computer program, can be found in programming environments. The latter normally include meta-programs, which are capable of describing their own activity, and meta-evaluations (symbolic evaluations), which evaluate programs by replacing actual values with abstract or symbolic values [77, 190].

Reflexive systems follow from the three following principles [70].

- There is a two-level language (the base or object level, and the meta-level) and there are two functions (denotation and reification) which associate elements in one language level to elements in the other.
- There is a link between the two levels, normally called the causality link, such that each operation undertaken at the meta-level can have an effect at the object level.
- The two previous principles can be generalized: for each language level $L$, there exists a meta-level $L'$, meta with respect to $L$. This principle shows the infinite aspect of reflexivity, including the ability to always pass to a meta-level from a given level.
A meta-representation of the MPVC is defined below. With it, the processor outgrowth of the system’s motor disappears. Furthermore, it becomes easier to define a specialized point of view, in order to better finetune MPVC systems.

9.4.1 MPVC meta-trilogy

The controller meta-description consists of applying the MPVC approach to the MPVC systems themselves. This is actually possible since the motor of these systems can be understood as a special interpretation of the traversed ordering graph.

Consider the MPVC decomposition \( A(M) \rightarrow \{C_A, P_A, M\} \). Let \( G_A \) be the ordering graph associated with controller \( C_A \) and let \( \mu \) be the motor of the MPVC system \( \{C_A, P_A, M\} \) such that \( \mu(\{C_A, P_A, M\}) = A(M) \). Let \( \mu_\mu \) be the traversal mechanism for \( G_A \) by \( \mu \) and \( P_\mu \) be the evaluation semantics for \( \mu \). Then the transformation

\[
\mu(\{C_A, P_A, M\}) \rightarrow \{\mu, P_\mu, \{C_A, P_A, M\}\}
\]

holds, thereby ensuring that

\[
\{C_\mu, P_\mu, \{C_A, P_A, M\}\} = \{C_A, P_A, M\}.
\]

So, \( C_\mu \) is the MPVC motor’s controller and the meta-controller of algorithm \( A(M) \); \( P_\mu \) is the motor’s point of view and the meta-point of view of algorithm \( A(M) \); and \( \{C_A, P_A, M\} \) is the motor’s model and the meta-model of algorithm \( A(M) \).

From these remarks, one can conclude that an MPVC trilogy is a meta-model of the algorithm that it represents. A meta-trilogy can be represented graphically (see Figure 9.10).

![Figure 9.10 The MPVC meta-trilogy](image)

In the discussion below, the terms model, point of view and controller refer to their meta-equivalents.

The meta-trilogy yields not just an experimental implementation of the MPVC approach – since the MPVC methodology is applied to the MPVC systems themselves – but also offers, for the system’s development, the prototyping tools for the
system itself. Hence the different meta-points of view that are built below can be used to manage, manipulate, test, observe and experiment with the behavior of a MPVC trilogy.

9.4.2 Implementation of the meta-model

The meta-controller should be able to work at the generic and specific levels as freely as possible. To simplify this process, new classes encapsulating data from the two levels are defined. The objects in these classes are the (causality) links between the two levels.

The MetaConcept class
From now on, before applying an ordering rule or link to an object that is an element of the model, the object is encapsulated in an instance of the MetaConcept class. A meta-concept is characterized by the object that it encapsulates (variable object); by the result of the evaluation of the rule test block within the encapsulated object (variable test); and by the collection resulting from the next link block within the encapsulated object (variable next).

Object subclass: #MetaConcept
instanceVariableNames: 'concept test next'
classVariableNames: '

poolDictionaries: 'category: 'Meta-MPVC''

When a MetaConcept is created, the object to be encapsulated must be designated, so the instantiation-initialization method is called concept:.

!MetaConcept class
    methodsFor: 'instantiation-initialization'
concept: aConcept
    ^ self new concept: aConcept!!

!MetaConcept methodsFor: 'access'
concept: aConcept
    "Initialization of encapsulated method.
    concept <- aConcept!!

There are six consultation methods.

!MetaConcept methodsFor: 'access'
concept
    ^ concept!
next
    ^ next!
The MetaRule class
An ordering rule is encapsulated by an instance of class MetaRule. A meta-rule is characterized by the ordering rule that it encapsulates (variable rule) and by the collection of meta-concepts (variable concepts) to which the rule should be applied.

```
Object subclass: #MetaRule
  instanceVariableNames: 'rule concepts'
  classVariableNames: '
  poolDictionaries: '
  category: 'Meta-MPVC'!
```

The instantiation-initialization is made by the rule:concepts: method.

```
!MetaRule methodFor: 'instantiation-initialization'!
  rule: aRule
case: aCollection
  self new rule: aRule; concepts: aCollection!!
```

There are four consultation methods.

```
!MetaRule methodFor: 'access'!
  rule
  rule!
  rule: aRule
  rule <- aRule!
  concepts
  concepts!
  concepts: aCollection
  concepts <- aCollection!!
```

The MetaLink class
The MetaLink class is dual to class MetaRule. Its instances encapsulate an ordering link (variable rule) and the collection of meta-concepts (variable concepts).

```
Object subclass: #MetaLink
  instanceVariableNames: 'rule concepts'
  ```
classVariableNames: ','
poolDictionaries:  ','
category: 'Meta-MPVC'!

The instantiation-initialization is made by the link:concepts: method.

!MetaLink class
  methodsFor: 'instantiation-initialization'!
link: aLink concepts: aCollection
  ^ self new link: aLink; concepts: aCollection! !

There are four consultation methods.

!MetaLink methodsFor: 'access'!
link
  ^ link!
link: aLink
  link <- aLink!
concepts
  ^ concepts!
concepts: aCollection
  concepts <- aCollection! !

9.4.3 Implementation of the meta-controller

The traversal mechanism defined by a meta-controller is based on the traversal of an ordering graph composed of ordering rules and links.

In Section 9.3, dealing with the concepts of sequential and parallel traversals, it was shown that there are two levels in the traversal of an ordering graph. First is the generic level, which describes the general ordering of the different steps in the traversal of the model. Second is the specific, individual aspect of the traversal. It is at this level that one chooses between a sequential and a parallel traversal of the model. This choice is shown below simply to be a choice of activating the meta-point of view for sequential execution or the meta-point of view for parallel execution.

The meta-controller must, therefore, manage the ordering of the different steps of the generic level. The meta-point of view that can be used by a meta-controller can have a meta-model that is either an ordering graph or an MPVC trilogy.

The meta-controller is the same in both situations, i.e. it has the same steps, ordered in the same manner, and only the points of view differ. For example, a meta-point of view for compiling a controller’s ordering graph undergoes the same evaluation steps as does an animation meta-point of view for interpreting the ordering graph and activating the point of view in the trilogy’s meta-model.

A compilation point of view is a meta-point of view that translates the model ordering graph into another formalism. The MPVC environment is based on three
of these meta-points of view: two create the controller class for the graph and define, optimally (MetaPVC compilOptim) or not (MetaPVC compilation), the instance method initialize that describes the graph in question; one saves an ordering graph (both MPVC characteristics and graphs) in a textual form.

An activation point of view actually interprets the ordering graph, i.e. interprets the rules and links and activates the associated point of view.

There is a difference between the two different kinds of point of view. For compilation points of view, an ordering rule should be traversed and compiled only once (this is the compilation traversal). For activation ones, each rule is traversed and interpreted as many times are are necessary to traverse the entire model (this is the execution traversal). The two cases are treated in parallel in the discussion below.

Below, the meta-controller defining the ordering in a graph is implemented using the ordering graph editor (see Figure 9.11). Traversal of this graph always starts with the #open rule, encapsulated in a meta-rule. After traversing a meta-rule, the links, encapsulated in a meta-link of the current encapsulated rule, are activated. Activation of a meta-link, in turn, activates the rule encapsulated in a meta-rule, at the end of the encapsulated link.

![Diagram](image)

Figure 9.11 MetaController

Here is the complete description of the rules in the graph.

The #open rule
The #open rule is the entry point.

```
(RuleOrdering named: #open
  if: '[[:ctrl :c | true]'
  do: '[[:pv :c | pv openWith: c]'])
nextRule: #metaRule
for: '[[:ctrl :c | Array with: c]]
nextRule: #close
```
for: ‘[:ctrl :c | Array with: c]’

Note that the point of view activation of rule #open now has a transmission: the message openWith: c is sent to the meta-point of view during the rule’s interpretation. This transmission, as well as the one defined below for rule #close, allow a point of view to be enclosed by an initial action as well as by a final action.

In this implementation, only the compilation meta-points of view redefine these enclosing methods locally. Before beginning the compilation of an ordering graph, it is necessary to check if the controller class containing the initialize method actually exists and, if not, to create it. Similarly, the file saving the compiled graph must be opened at the beginning and must be closed at the end.

The first ordering link triggers the real traversal of the ordering graph. It begins by traversing the #open rule of the model ordering graph. This rule is encapsulated in an instance of class MetaRule, which is the first object in the traversed meta-model. The MetaRule is assigned to the startConcept instance variable of the MetaController. Furthermore, the meta-rule encapsulates the first object to be traversed by the meta-model's controller, this object being itself encapsulated in an instance of class MetaConcept.

So, let aCtrl be a local variable referencing the controller instance whose ordering graph is to be traversed. The MetaRule is created by evaluating the following expression.

```
MetaRule
    "Encapsulation of the rule"
    "Rule to be traversed by the meta-controller"
    rule: (aCtrl graph at: #open)
    concepts: (OrderedCollection with:
                  (MetaConcept
                   "Encapsulation of the object"
                   "Object to be traversed by the ctrl"
                   concept: aCtrl startConcept)
```

The second ordering link refers to rule #close, which finishes the meta-model's traversal.

The #metaRule rule
The #metaRule rule describes the behavior that a meta-controller must adopt when traversing a meta-rule.

```
(RuleOrdering named: #metaRule
   if: ‘[:ctrl :c | ctrl isActivableMetaRule: c]’
   do: ‘[:pv :c | pv activateMetaRule: c]’)
nextRule: #metaLinks
for: ‘[:ctrl :c | ctrl nextMetaLinkFor: c]’
```
The test block asks the current controller if it is possible to activate the current meta-rule, by sending the `isActivatableMetaRule:` message to the controller, with the meta-rule as argument. The definition of the associated method is different for a compilation traversal or an execution traversal.

To take account of the two kinds of traversal, two subclasses of `MetaController` are created. However, before the subclasses can be created, the `MetaController` superclass must be created.

```ruby
ControllerMPVC subclass: #MetaController
  instanceVariableNames: '
    
  classVariableNames: '
    
  poolDictionaries: 
    category: 'MPVC-MPVC'!
```

The `isActivatableMetaRule:` method is defined as the responsibility of the subclasses of `MetaController`.

```ruby
!MetaController methodsFor: 'testing'

isActivatableMetaRule: aMetaRule
  ^ self subclassResponsibility!
```

The `MetaCtrlForExecution` subclass handles an execution traversal.

```ruby
MetaController subclass: #MetaCtrlForExecution
  instanceVariableNames: '
    
  classVariableNames: '
    
  poolDictionaries: 
    category: 'MPVC-MPVC'!
```

The `MetaCtrlForCompilation` subclass handles a compilation traversal.

```ruby
MetaController subclass: #MetaCtrlForCompilation
  instanceVariableNames: 'wasActive'
  classVariableNames: '
    
  poolDictionaries: 
    category: 'MPVC-MPVC'!
```

This last class defines an instance variable `wasActive` that references a set (instance of class `Set`) containing the rules that have already been traversed by the meta-controller. It is initialized during the controller's creation, by the local redefinition of the class method `new`.

```ruby
!MetaCtrlForCompilation class methodsFor: 'instantiation'
new
  ^ super new initActivation!

!MetaCtrlForCompilation methodsFor: 'initialization'
initActivation
  wasActive <- Set new!
```
The `isActivableMetaRule` method can now be redefined locally. For a compilation traversal, a meta-rule can only be traversed if its encapsulating rule has not yet been traversed.

```
!MetaCtrlForCompilation methodsFor: 'testing'
   isActivableMetaRule: aMetaRule
      (wasActive includes: aMetaRule rule) not! !
```

For an execution traversal, a meta-rule can only be traversed if its collection of concepts is non-empty.

```
!MetaCtrlForExecution methodsFor: 'testing'
   isActivableMetaRule: aMetaRule
      aMetaRule concepts isEmpty! !
```

The activation point of view block specifies that the `activateMetaRule: c` message must be sent to the associated meta-point of view.

The ordering link states that the `#metaLinks` rule must be applied to the meta-links resulting from the evaluation of the expression

```
ctrl nextMetaLinkFor: c
```

These metalinkes are obtained from the meta-controller. This result of this computation depends on the kind of traversal (compilation or execution). The method `nextMetaLinkFor:` is first defined at the `MetaController` class level, as a subclass responsibility.

```
!MetaController methodsFor: 'accessing'
   nextMetaLinkFor: aMetaRule
      self subclassResponsibility! !
```

The method is redefined for `MetaCtrlForCompilation`.

```
!MetaCtrlForCompilation methodsFor: 'accessing'
   nextMetaLinkFor: aMetaRule
      wasActive add: aMetaRule rule.
         Array with: aMetaRule nexts! !
```

It states that the meta-controller should first memorize the rule that was just traversed, using the transmission.

```
   wasActive add: aMetaRule rule.
```

It then returns an array with a single element, the result of message `nexts` sent to the traversed meta-rule. The method associated with message `nexts` returns a collection of `MetaLink` encapsulating both the links to be traversed and the meta-concepts encapsulating the objects to which the next block should be applied. Here is the method’s definition.
MetaRule methodsFor: 'access'

nexts
  rule nexts
  collect: [:link | MetaLink link: link
    concepts: concepts]]

The nextMetaLinkFor: method is redefined for class MetaCtrlForExecution.

MetaCtrlForExecution methodsFor: 'accessing'

nextMetaLinkFor: aMetaRule
  Array with: aMetaRule purge next

This method returns an array containing a single element, the result of messages purge and nexts being sent to the traversed meta-rule. The method associated with message purge allows the meta-concepts to be extracted from the meta-rule whose test instance variable is false (test is set by the meta-points of view for the execution of an MPVC trilogy). In other words, this method extracts the meta-concepts not satisfying the test block of the encapsulated rule.

MetaRule methodsFor: 'access'

purge
  concepts <- concepts select: [:c | c testValue]

The #metaLinks rule

Rule #metaLinks is applied to a collection of MetaLink. This intermediate rule allows one to fix the order in which the different ordering links are traversed. By definition, the ordering links must be activated in a well-defined order.

If the #metaLinks rule is omitted from the ordering graph (see Figure 9.12), then that means that with a parallel processor the ordering links would be traversed simultaneously.

![Diagram](image)

**Figure 9.12 MetaController 2**
(RuleOrdering named: #metaLinks
   if: '[:ctrl :c | c isEmpty not]'
   do: '[:pv :c | ]'
   nextRule: #metaLink
   for: '[:ctrl :c | Array with: c removeFirst]';
   nextRule: #metaLinks
   for: '[:ctrl :c | Array with: c]'
)

The ordering of the links is set by a reflexive link attached to the #metaLinks rule. This rule allows looping over the links.

The test block

'[:ctrl :c | c isEmpty not]

defines the test that halts the loop once all the links have been traversed.

The first link indicates that the #metaLink must be applied to the first element in the traversed collection, at the same time removing it.

Array with: c removeFirst

The #metaLink rule

Rule #metaLink is the dual rule for rule #metaRule. It describes the behavior that the meta-controller must have during the meta-link’s traversal.

(RuleOrdering named: #metaLink
   if: '[:ctrl :c | ctrl isActivatableMetaLink: c]'
   do: '[:pv :c | pv activateMetaLink: c]'
   nextRule: #metaRule
   for: '[:ctrl :c | ctrl nextMetaLinkFor: c]'
)

The block test asks the current controller if it is possible to activate the current meta-link. The test is made by sending the isActivatableMetaLink: message to the controller with the traversed meta-link as argument. As for the isActivatableMetaRule: message, the method differs for a compilation traversal and for an execution traversal.

The initial definition is made at the MetaController level.

!MetaController methodsFor: 'testing'
   isActivatableMetaLink: aMetaLink
   ~ self subclassResponsibility! !

For a compilation traversal, it is not necessary to test if the encapsulated link has already been traversed. Since an ordering link belongs to a single rule, it can only be activated once. Hence the definition for MetaCtrlForCompilation.

!MetaCtrlForCompilation methodsFor: 'testing'
   isActivatableMetaLink: aMetaLink
   ~ true! !
For an execution traversal, a meta-link is only traversed if its collection of concepts is non-empty.

```plaintext
!MetaCtrlForExecution methodsFor: 'testing'
   isActiveivableMetaLink: aMetaLink
      aMetaLink concepts isEmpty not!

The point of view activation block specifies that the message

activateMetaLink: c

must be sent to the associated meta-point of view.

The only ordering link indicates that the #metaRule should be applied to the sole meta-rule resulting from the evaluation of the expression

```plaintext
ctrl nextMetaLinkFor: c

It is the meta-controller that decides which meta-rule should be traversed. For the #metaRule rule, this computation depends on the kind of traversal (compilation or execution). In the other methods, the nextMetaLinkFor: method is first defined at the MetaController level.

```plaintext
!MetaController methodsFor: 'accessing'
   nextMetaRuleFor: aMetaLink
      self subclassResponsibility!

The method is redefined at the MetaCtrlForCompilation level.

```plaintext
!MetaCtrlForExecution methodsFor: 'accessing'
   nextMetaRuleFor: aMetaLink
      Array with: aMetaLink rule!

It returns an array containing a sole element, a result of the rule message sent to the traversed meta-link. The method associated with message rule returns a MetaRule encapsulating the next rule to be traversed and the meta-concepts that should be applied.

```plaintext
!MetaLink methodsFor: 'access'
   rule
      newConcepts
      newConcepts <- OrderedCollection new.
      concepts
      do: [:concept |
         concept next notNil
         ifTrue: [concept next
            do: [:c |
               newConcepts
               add: (MetaConcept concept: c)]]]]].
      MetaRule rule: link rule concepts: newConcepts!
```
The method is also redefined at the MetaCtrlForExecution level.

```lisp
!MetaCtrlForExecution methodsFor: 'accessing'
nextMetaRuleFor: aMetaLink
   ^ Array with: aMetaLink purge rule!!
```

This method returns an array containing a single element, the result of messages purge and rule being sent to the traversed meta-link. The method associated with the purge message extracts the meta-concepts encapsulated in the meta-link whose next instance variable (set by the meta-points of view for the execution of an MPVC trilogy) contains no element.

```lisp
!MetaLink methodsFor: 'access'!!
   purge
   concepts <- concepts select: [:c | c next notNil]]!!
```

The **#close** rule

The **#close** rule describes the behavior to adopt once the ordering graph traversal is finished.

```lisp
(RuleOrdering named: #close
   if: '[:ctrl :c | true]
   do: '[:pv :c | pv closeWith: c]')
```

The point of view activation states that the **closeWith: c** message should be sent to the current point of view.

The **compiled graph**

Compiling the graph yields the following initialization method.

```lisp
!MetaController methodsFor: 'initialization'!!
   initialize
      self
      addRule:
        ((RuleOrdering named: #open
           if: '[:ctrl :c | true]
           do: '[:pv :c | pv openWith: c]')
        nextRule: #metaRule
          for: '[:ctrl :c | Array with: c]';
        nextRule: #close
          for: '[:ctrl :c | Array with: c]');
        addRule:
          ((RuleOrdering named: #metaRule
             if: '[:ctrl :c | ctrl isActivableMetaRule: c]
             do: '[:pv :c | pv activateMetaRule: c]')
          nextRule: #metaLinks
```
for: `[:[ctrl :c | ctrl nextMetaLinkFor: c]]`
addRule:
  ((RuleOrdering named: #metaLinks
    if: `:[:[ctrl :c | c isEmpty not]]`
    do: `:[:[pv :c | []]]`
  )
nextRule: #metaLink
for: `:[:[ctrl :c | Array with: c removeFirst]]`
nextRule: #metaLinks
for: `:[:[ctrl :c | Array with: c]]`
addRule:
  ((RuleOrdering named: #metaLink
    if: `:[:[ctrl :c | ctrl isActivableMetaLink: c]]`
    do: `:[:[pv :c | pv activateMetaLink: c]]`
  )
nextRule: #metaRule
for: `:[:[ctrl :c | ctrl nextMetaLinkFor: c]]`
addRule:
  ((RuleOrdering named: #close
    if: `:[:[ctrl :c | true]]`
    do: `:[:[pv :c | pv closeWith: c]]`
  )
A meta-controller is created by instantiating either MetaCtrlForCompilation, if one wishes to compile the ordering graph, or MetaCtrlForExecution, if one wishes to interpret the ordering graph.

Before actually implementing different meta-points of view, note that the collections of objects to which rules are applied by the meta-controller are all singletons. Rule #metaRule is applied either to the singleton resulting from evaluating the expression

`Array with: c`
in rule #open or to the singleton resulting from evaluating the expression

`ctrl nextMetaRuleFor: c`
in rule #metaLink.

Rule #metaLinks is applied to the singleton generated by the #metaRule rule, the result of the expression

`ctrl nextMetaLinkFor: c;`

Rule #metaLink is applied to the singleton generated by the #metaLinks rule, the result of the expression

`Array with: c removeFirst.`
Hence the meta-controller will never have to simulate several objects in parallel. It is therefore useless to provide a parallel processor upon activation of a meta-controller since it will have the same effect as a sequential processor.

This remark is in keeping with the generic level definition of the traversal of an ordering graph. The choice between a sequential and a parallel traversal of the model must be made at the specific level of traversal. In the next subsection, the specific level is handled by the meta-points of view of the meta-trilogy and, consequently, it is upon creation of the meta-points of view that the choice takes place between a sequential and a parallel traversal of the object model.

### 9.4.4 Implementing meta-points of view

Every meta-point of view activated by a meta-controller has four local evaluation semantics, whose names are (in the meta-controller definition) openWith:, activateMetaRule:, closeWith: and activateMetaLink:. These messages are sent to the meta-point of view from the current meta-trilogy, respectively, at the beginning of the graph traversal, when a MetaRule is being applied, when a MetaLink is being applied, and at the end of the traversal.

Below are the meta-points of view needed to interpret the meta-model trilogy in sequential mode or in parallel mode, as well as the meta-point of view needed for the graphical animation of the ordering graph traversal.

**The MetaPVSquential meta-point of view**

The MetaPVSquential class allows a sequential execution of the point of view of the meta-model. Its methods define the behaviors to be put into effect at different steps of the traversal by the meta-controller.

```ruby
PointOfViewMPVC subclass: #MetaPVSquential
  instanceVariableNames: ";
  classVariableNames: ";
  poolDictionaries: ";
  category: 'Meta-MPVC'!
```

This class contains the four methods that respond to the messages openWith:, activateMetaRule:, activateMetaLink: and closeWith:.

Method openWith: describes the behavior to be adopted when traversing the ordering graph. For this meta-point of view, no initial behavior is necessary.

```ruby
!MetaPVSquential methodsFor: 'activation'!
openWith: aMetaRule
  self !
```

Method activateMetaRule: defines the behavior for applying a MetaRule. For this meta-point of view, it states how to evaluate the encapsulated rule; the evaluation is made for each of the meta-concepts encapsulated by the MetaRule.
MetaPVSequential methodsFor: 'activation'!
activateMetaRule: aMetaRule
   aMetaRule concepts
do: [:mconcept |
   mconcept test: (aMetaRule rule test
      value: model
      value: mconcept concept).
   mconcept testValue
   ifTrue: [aMetaRule rule activity
      value: model pv
      value: mconcept concept]] !

This method evaluates, for each of the encapsulated meta-concepts, the test block of the encapsulated rule. The result of each evaluation is assigned to the test instance variable of the meta-concept; if the test succeeds, the point of view activation block is evaluated in turn.

The activateMetaLink: method defines the behavior for following a MetaLink. For this point of view, it states how to evaluate the encapsulated link for each of the meta-concepts encapsulated by the MetaLink.

MetaPVSequential methodsFor: 'activation'!
activateMetaLink: aMetaLink
   aMetaLink concepts
do: [:mconcept |
   mconcept next: (aMetaLink link block
      value: model
      value: mconcept concept)] !

This method evaluates, for each of the encapsulated meta-concepts, the block of the encapsulated link. The result of each evaluation is assigned to the next variable of the encapsulated meta-concept.

The closeWith: method describes the behavior after the traversal of the ordering graph. For this meta-point of view, no terminal behavior is necessary. Consequently, the closeWith: has no behavior.

MetaPVSequential methodsFor: 'activation'!
closeWith: aMetaRule
   ~ self !!

Note that this meta-point of view makes no reference to a processor. The same holds true for the meta-point of view that interprets a parallel point of view. The outgrowth of MPVC systems that was added to deal with parallelism is no longer needed.

It is now possible to define the first meta-trilogy, which sequentially evaluates any given trilogy. To do this, the meta-model, i.e. an MPVC trilogy, must already be defined. Consider, for example, the evaluation of a neural net.
pv <- EvaluationByInhibition model: nn.
ctrl <- ControllerNetNeural
   model: nn pv: pv startConcept: nn

The controller instance, referenced by variable ctrl, plays the rôle of meta-model, as it regroups the three components in an MPVC trilogy. So here is the meta-trilogy definition.

metaPV <- MetaPVSequential model: ctrl.
metaCtrl <- MetaController
   model: ctrl
   pv : metaPV
   startConcept:
      (MetaRule rule: (ctrl graph at: #open)
         concepts: (OrderedCollection
            with: (MetaConcept concept:
               ctrl startConcept))).

Activating this meta-trilogy is no different from any other: the (meta-)controller should be sent the message open.

metaCtrl open

Of course, this activation is equivalent to the direct activation of the object controller (ctrl open) because of the equality explained in the definition of a meta-trilogy:

\[
\{C_\mu , P_\mu , \{C_A, P_A, M\}\} = \{C_A, P_A, M\}.
\]

The MetaPVParallel meta-point of view
The MetaPVParallel meta-point of view is for executing in parallel the point of view of a meta-model.

PointOfViewMPVC subclass: #MetaPVParallel
   instanceVariableNames: '\'
   classVariableNames: '\'
   poolDictionaries: '\'
   category: 'Meta-MPVC'!

The definitions of methods activateMetaRule: and activateMetaLink: differ little from the definitions of their homonyms in MetaPVSequential. The same transmissions are encapsulated in the processes (instances of Process).

   !MetaPVParallel methodsFor: 'activation'!
openWith: aMetaRule
   ^ self !
activateMetaRule: aMetaRule
| semaphore |
semaphore <- SemaphoreCounter
  forSignal: aMetaRule concepts size.
(aMetaRule concepts collect: [:mconcept | :
  [:mc | mc test: (aMetaRule rule test copy
      value: model
      value: mc concept).
    mc testValue
    ifTrue: [aMetaRule rule activity copy
      value: model pv
      value: mc concept].
    semaphore signal]
    newProcessWith: (Array with: mconcept)])
  do: [:process | process resume].
semaphore wait! !

!MetaPVParallel methodsFor: 'activation'!
activateMetaLink: aMetaLink
  | semaphore |
semaphore <- SemaphoreCounter
  forSignal: aMetaLink concepts size.
(aMetaLink concepts collect: [:mconcept | :
  [:mc | mc test: (aMetaLink link block copy
      value: model
      value: mc concept).
    semaphore signal]
    newProcessWith: (Array with: mconcept)])
  do: [:process | process resume].
semaphore wait!
closeWith: aMetaRule
  ~ self! !

This meta-point of view does not need the concept of processor. Consequently, there is no longer any need to have a special processor for a controller's activation: this outgrowth has been replaced by the meta-vision – in keeping with the meta-point of view – that a meta-trilogy has of its model trilogy.

Hence, starting up the parallel execution of the neural net mn takes place by evaluating the neural net trilogy with an instance of MetaPVParallel as the meta-point of view, giving the following transmissions.

"Create an object--point of view"
Properties and extensions of the MPVC environment

 pv <- MultiPV model: nn
     for: (Array with: EvaluationByInhibition new
           with: AnimationNetNeural new).
"Create the controller"
ctrl <- ControllerNetNeural
"Create the meta-point of view"
metaPV <- MetaPVPParallel model: ctrl.
"Creation of the meta-controller"
metaCtrl <- MetaController
    model: ctrl pv: metaPV startConcept: nil.
"Activation of the meta-trilogy"
metaCtrl open.

The MetaPVAanimation meta-point of view
By keeping in mind the graphical animation point of view for neural nets, a new
meta-point of view (class MetaPVAanimation) is defined, allowing the graphical
animation of the ordering graph's traversal.

PointOfViewMPVC subclass: #MetaPVAanimation
    instanceVariableNames: ',
    classVariableNames: ',
    poolDictionaries: ',
    category: 'Meta-MPVC'!

Here are the four methods describing the evaluation semantics specific to this
point of view.

!MetaPVAanimation methodsFor: 'activation'!
openWith: aMetaRule
    ^ self!
activateMetaRule: aMetaRule
    aMetaRule rule flash!
activateMetaLink: aMetaLink
    aMetaLink link path!
closeWith: aMetaRule
    ^ self!

These methods use the messages flash and path, already used during the definition
of point of view AnimationNetNeural. These messages here reference the same
methods: the flash method in class NodeGraphic and the path method in class
ArcGraphic.

Encapsulating in a multiple point of view both AnimationNetNeural and one of
MetaPVPParallel or MetaPVSquential is sufficient to effect a video trace of the
order of application of the links and rules of the meta-model's controller.
"Create a meta-point of view"
metaPV <- MultiPV model: ctrl
  for: (Array with: MetaPVParallel new
       with: MetaPVAnimation new).

"Create the meta-controller"
metaCtrl <- MetaController
  model: ctrl pv: metaPV startConcept: nil.

"Activate the meta-trilogy"
metaCtrl <- open.

The point of view AnimationNetNeural is a finetuning tool for the specific level. It animates the ordering of the different elements in the traversed object-model.

As for the meta-point of view MetaPVAnimation, it is a finetuning tool for the generic or meta-level. It animates the ordering of the rules and links, i.e. it animates the ordering of the ordering.

These two tools can be combined, thereby allowing cascade animation. For example, Figure 9.13 shows the simultaneous graphical animation of

- the traversal of the ordering graph of a meta-controller that is traversing the controller's ordering graph;
- the traversal of the ordering graph of the controller that is traversing a neural net; and
- the traversal of a neural net.

9.5 An MPVC browser

For the controllers and points of view and their meta-equivalents in an MPVC system to be easily managed, a browser is presented here.

In Smalltalk, a browser is a view with several subviews, organizing information so that it can be consulted; the best-known browser is the System Browser, which provides access to the classes and methods of the current environment.

9.5.1 Description

The browser screen is shown in Figure 9.14. It has six subviews:

1. This view is a scrolling window (instance of class SelectionInListView). It shows the names of point of view classes that can be selected (at the object level) for the trilogy.
2. This scrolling window shows an ordered collection of the points of view that have been selected for the trilogy at the object level.
3. This scrolling window shows the names of controller classes that can be included in the trilogy at the object level.

4. This view is switchable: it is an instance of class BooleanView and can have two states, activated or deactivated; normally this is shown by the view's label color (white on black or black on white). Selecting this view triggers the creation of subviews in view 5 in order to build a meta-trilogy that can be included in the trilogy.

5. This view, initially gray, provides the space required for creating subviews used during the meta-trilogy's description.

6. When selected, this switchable view creates and activates the MPVC system described in the different subviews.
9.5.2 Creation

The MPVC browser is created as soon as one begins to build or activate an MPVC system. For example the neural net editor creates the MPVC browser with the neural net as model; the classes EvaluationByInhibition, EvaluationBySubtraction and AnimationNetNeural as points of view; and the classes ControllerNetNeural and ControllerNNSynchro as controllers. The resulting browser is shown in Figure 9.15.

Once the trilogy has been described, the user can access the meta-level and define the meta-trilogy that generates the object level trilogy. To do this, the META switchable view is selected; the subviews for building the meta-trilogy are created. The resulting browser is shown in Figure 9.16.

At the object level, the trilogy for the MPVC system shown in Figure 9.16 includes the points of view EvaluationByInhibition and AnimationNetNeural and the controller ControllerNNSynchro. At the meta-level, the meta-trilogy includes the meta-points of view MetaPVPParallel and MetaPVAnimation and the meta-controller ControllerNNSynchro.

9.5.3 Long-range reflexivity and the MPVC browser

The MPVC browser can handle an unbounded number of reflexive systems, where for each language level $L$, there is a level $L'$ that is meta-$L$. In other words, a meta-
### Figure 9.15 Browser after creation

<table>
<thead>
<tr>
<th>Points of View</th>
<th>Selected PV</th>
<th>Controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvaluationByInhibition</td>
<td>EvaluationByInhibition</td>
<td>ControllerNetNeural</td>
</tr>
<tr>
<td>EvaluationBySubtraction</td>
<td>AnimationNetNeural</td>
<td>ControllerNNSynchro</td>
</tr>
<tr>
<td>AnimationNetNeural</td>
<td></td>
<td>META</td>
</tr>
</tbody>
</table>

### Figure 9.16 Browser after creation of subviews

<table>
<thead>
<tr>
<th>Points of View</th>
<th>Selected PV</th>
<th>Controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaPVSequential</td>
<td>MetaPVParallel</td>
<td>MetaCtrlForCompilation</td>
</tr>
<tr>
<td>MetaPVParallel</td>
<td>MetaPVAnimation</td>
<td>MetaCtrlForExecution</td>
</tr>
<tr>
<td>MetaPVAnimation</td>
<td></td>
<td>META</td>
</tr>
</tbody>
</table>
trilogy is an MPVC trilogy that can itself be the model for a meta-meta-trilogy, and so on.

In the browser, this means that at each level, it is possible to define the meta-level of the current level simply by selecting the switchable view META of the current level.

For example, to get graphical animation of the meta-controller's ordering, the graphical animation point of view must be selected at the meta-meta-level (see Figure 9.17).

![MPVC Browser Table]

**Figure 9.17 Browser after animation**

Activating this system sets off a cascade animation, i.e. animation at three levels (see Figure 9.18):

- the ordering graph of the MetaCtrlForExecution controller, i.e. the ordering graph for MetaController, which traverses the ControllerNNSynchro controller;
- the ordering graph of the ControllerNNSynchro controller traversing the neural net described by the editor;
- the traversal of the neural net by the ControllerNNSynchro controller.

This kind of tool in no way modifies the existing points of view and, consequently, the algorithm that was first defined. This non-interference property, a result of the multiple points of view implementation, allows the MPVC system designer to elaborate as many different tools as are necessary and to combine them in different ways. For example, pre- and post-evaluation assertion points of view could be defined to check the data upon entering and exiting each step in a traversal. The designer builds a personalized assistant, a specialized programming environment.
Figure 9.18 Screen snapshot during a cascade animation
Chapter 10

Conclusion

10.1 Summary

This book has presented a prototyping system based on a three-pronged approach to program construction. Focus was placed on the design of tools facilitating the implementation of multiple representation models during the specification, design and validation of software.

Once the concepts pertaining to prototyping were presented, the structuralist approach, as it is implemented in object languages, was presented. As a result, object-oriented environments were shown to be modeling and programming environments well suited to prototyping.

The MPVC approach was then presented as an object-oriented design method that favors the implementation of multiple software tools for the manipulation, interpretation and finetuning of representation models.

This approach can be considered to be an experimental prototyping method and the resulting MPVC environment as a design tool for interpreters of multiple representation models.

The Model–View–Controller scheme, also known as the MVC methodology, was presented. In an interactive application, this scheme calls for the clear distinction between the internal representation of an application and its input and output interfaces. This approach simplifies and systematizes the writing of such applications.

The MPVC approach is similar: clearly distinguish the traversal mechanism for a representation model from the multiple associated interpretations. Hence, the writing of interpreters of representation models is facilitated and systematized.

Implementing this approach in the MPVC environment gives the designer the chance of

- designing, using graphics, multiple traversals of the same model;
- designing multiple points of view;
• experimenting with the combination of each of the traversals associated with
  one or more points of view, using the MPVC system;
• building multiple tools for personalized finetuning, as in video tracers, steppers, and pre- and post-tests;
• studying the behavior of its systems by observing the path of the model in a sequential or parallel manner;
• finetuning the tools in the base environment or augmenting the latter with new tools – meta-points of view – that can examine or even modify the general behavior of these systems.

These characteristics definitely favor a speculative attitude by the user-designer, with respect to the studied model; the designer simultaneously builds the interpreters and the tools needed for experimentation and validation. The resulting environment can then be seen as a platform for mechanical schemata – the controllers – and semantic schemata – the points of view – of traversals of representation models.

Three immediate extensions to the MPVC system are proposed here. We conclude by presenting the different perspectives offered by such an environment.

10.2 Extensions to the MPVC environment

10.2.1 Trilogy interrupt mechanisms

The first extension that could be added to the MPVC environment would be to implement a basic mechanism allowing a point of view to interrupt its execution, i.e. to ask its controller to be activated no longer.

The points of view presented up to now do not need this mechanism. They never find themselves in a critical situation requiring their suspension. This kind of situation might come about because of evaluations that cannot continue, either because of lack of information or because an error has taken place during a computation; for example, a point of view could encounter an error when reading from a disk.

The current MPVC environment does not know how to handle this kind of information. The guiding principle was that a point of view has no access to its controller and cannot directly influence its activity. Consequently, it cannot ask its controller to suspend it.

10.2.2 Automatic generation of points of view

A second extension consists of enriching the MPVC environment with a new compilation meta-point of view, which would generate an abstract class, subclass of PointOfViewMPVC, that would incorporate the methods used in the messages sent
by the controller during a graph traversal. These would serve as default meth-
ods for the subclasses describing a particular point of view for which some of the
methods invoked by the controller have no effect.

For example, the abstract class MetaPVAbs tract, generated during the traversal
of the MetaController controller’s ordering graph, would be defined as follows.

```
PointOfView subclass: #MetaPVAbs tract
    instanceVariableNames: '\''
    classVariableNames: '``
    poolDictionaries: '``
    category: 'Meta-MPVC'!

    !MetaPVAbs tract methodsFor: 'activation'!
    openWith: aMetaRule
      ^ self !
    activateMetaRule: aMetaRule
      ^ self !
    activateMetaLink: aMetaLink
      ^ self !
    closeWith: aMetaLink
      ^ self ! !
```

With this abstract class, the user can ignore, at the point of view level, the
methods that do not describe anything in particular. For example, the openWith:
method of class MetaPVSequential includes no transmission, since that point of
view does not require any initial behavior. The same holds true for its final behav-
ior: closeWith: includes no transmission. By creating class MetaPVSequential
as a subclass of class MetaPVAbs tract, these empty behaviors need no longer be
written down. As a result, class MetaPVSequential should define only non-empty
behaviors, here methods activateMetaRule: and activateMetaLink:.

This automatic generation of default points of view means that the controller’s
behavior can be checked experimentally right from its initial design; the user can
immediately test the controller by providing it with an instance of the generated
point of view. The generation also documents the MPVC system that is being
developed, as it summarizes the messages that each new point of view must be
able to respond to.

### 10.2.3 Transformation meta-point of view

The last extension is to implement a second compilation meta-point of view, which
transforms the traversed meta-model trilogy into a program that runs indepen-
dently of the MPVC system. Once the designer has tested and validated several
points of view and controllers, he or she can use this meta-point of view to pass
directly from the prototyping to coding.
A particularly interesting kind of prototyping uses two similar languages, $L_1$ for building of the prototype and $L_2$ for the operational program, thereby improving the construction manual aspects of prototyping [17].

This transformation step (analogous to transformational programming [10, 46, 142]) is more efficient if the target language is similar to the source language, here Smalltalk-80. This step should generate code that is close to the final implementation, while still preserving the structural and functional characteristics specified in the MPVC trilogy [142].

Implementing such a meta-point of view enlarges the scope of the MPVC approach, since it allows tools for a software engineering environment to be included in an MPVC system, thereby covering much of the life cycle of an application.

10.3 Overview

10.3.1 MPVC as a programming environment

The MPVC environment is completely integrated into the Smalltalk-80 environment [75, 76]. It therefore has access to the language’s entire environment: classes, instances and methods — and it is possible to build MPVC systems for this environment.

For example, a controller could be built to traverse the inheritance graph. This controller could be tied to a point of view that would provide the methods that use a given selector. Taking the Smalltalk-80 inheritance graph as model for the trilogy would give an already existing utility, activated by the transmission

```
Smalltalk browserAllCallsOn: #msg
```

which opens a window on all methods using the selector #msg.

A second point of view of this traversal could reconstruct the text file containing all the user method definitions. This file could be rebuilt by keeping only the last version of each definition. As for the preceding point of view, there is a Smalltalk-80 utility that does just that.

```
Smalltalk condenseChanges
```

Implementing these two utilities as MPVC points of view means that they could be triggered simultaneously. The environment could even be enriched by a third point of view that visualizes, over a graphical representation of the inheritance graph, the class currently being traversed by the system, thereby informing the user of the system’s current state.

Another example is the formalization as an MPVC system of the Smalltalk debugger [75], which simulates and visualizes, step by step, an expression’s interpretation. The controller of such a system would describe how to traverse a Smalltalk
expression and the points of view would be the different kinds of interpretation: evaluation, video trace, symbolic evaluation, compilation, step by step, etc.

Nevertheless, the MPVC environment does not really depend on the Smalltalk environment. The implemented concepts use only general mechanisms that can be found in most object languages. Porting such an environment to a language such as Flavors [134], ObjVLisp [49], CLOS [23], LORE [15] or Eiffel [128] would be no problem, consisting essentially of the translation of the basic classes and mechanisms of MPVC systems. We adapted in less than three hours the entire MPVC system to a Smalltalk-80 system that did not have multiple inheritance.

A new project would then be the integration of our MPVC system into an object-oriented language with a high-level, uncompiled internal representation (for example, the object-oriented languages based on LISP [124], such as Flavors, ObjVLisp or CLOS, represent their methods as lists of lambda-expressions [191]). We would then have a programming environment with all the functionality of an MPVC environment. Such an environment would allow the easy implementation of multiple assistance tools [192] while a program is running, thereby allowing, simultaneously, many different kinds of information about the program, including assertions about the input or output of methods, version histories and documentation [192].

10.3.2 MPVC as a tool for understanding and managing interpretative strategies

Up to now, the implementation of reflexivity in the MPVC program representation model has served as an example of meta-describable systems and of how to rapidly build tools for finetuning MPVC systems.

This concept can be more fully developed by using existing ideas about reflexivity [132, 147, 169]: reflexivity is a means to produce intelligent programs capable of reasoning about their own behavior, hence to assist and modify them.

Minsky defined reflexivity as the examination of brain A by brain B (see Figure 10.1). He cites as examples some of the functions that a reflexive system should be able to do. Below, in the MPVC system, the world is a model, brain A is an MPVC trilogy and brain B is a meta-trilogy interpreting a meta-model trilogy.

Now A can see and act upon what happens in the outside world – while B can 'see' and influence what happens inside A. What uses could there for such a B? Here are some A-activities that B might learn to recognize and influence.

1. A seems disordered and confused. Inhibit that activity.
2. A appears to be repeating itself. Make A stop. Do something else.
3. A does something B considers good. Make A remember it.
4. A is occupied with too much detail. Make A take a higher-level view.
5. A is not being specific enough. Focus A on lower-level details.
Figure 10.1 Minsky’s reflexivity

... brain $B$ could learn to play a rôle somewhat like that of a counselor, psychologist or management consultant, who can access a client’s strategy without having to understand all the details of that client’s profession [132, p. 59].

In the MPVC approach, a controller is a particular strategy for traversing a model and a point of view is a particular interpretation strategy. We have shown that it is possible to design and implement multiple controllers, i.e. multiple traversal strategies, and multiple points of view, i.e. multiple interpretation strategies (see Figure 10.2).

Figure 10.2 MPVC meta-trilogy

Hence it is possible, using the meta-trilogy concept, to design meta-points of view that reason about the behavior of a meta-model trilogy, i.e. a meta-trilogy reasoning about a meta-model trilogy.

For example, it is possible to design activation meta-points of view that are capable of changing, if need be, the controller of the meta-model being traversed, i.e. of changing the strategy for traversing the model in the meta-model. Additional activation meta-points of view, capable of changing the interpretation strategies of the model in the meta-model, can also be designed, by changing the points of...
view associated with the traversal of the model in the meta-model. Some examples follow, using the numbering for examples in the above quote from Minsky.

1. A seems disordered and confused. Inhibit that activity.

A meta-point of view could handle exceptional events, of different kinds, generated by the meta-model's controller or points of view.

**Domain exceptions.** An input assertion failed upon calling an operation, as when a method does not receive an argument of the right type.

**Scope exceptions.** An output assertion failed or cannot be verified upon exiting an operation, as when the set of elements to be traversed next by the controller is computed and turns out not to be a collection.

**Programmed exceptions.** The exception is not necessarily an error, since it allows, for example, a point of view to interrupt itself by triggering an exception.

By definition, an exceptional situation cannot be resolved in a local manner by the procedure – here the point of view or the controller. More precisely, it can be resolved locally, but the decision whether it should be resolved locally is made elsewhere. The meta-point of view that must handle the exception must be capable of finding the handler for the given exception [63]: halting the computation, restarting the computation at the instruction after the one that provoked the signal, or triggering another exception (normally of a higher conceptual level, here the encompassing meta-trilogy, if it exists).

2. A appears to be repeating itself. Make A stop. Do something else.

A meta-point of view could check that a controller is not looping over and over on the same element in the model; if such is the case, it could signal the problem, thereby allowing the interruption of the computation without actually reinitializing the entire process.

3. A does something B considers good. Make A remember it.

A third meta-point of view, 'seeing' that a rule in the ordering graph has an empty point of view activation block, could inform the meta-model's controller to attempt to evaluate this empty block no longer.

4. A is occupied with too much detail. Make A take a higher-level view.

Another meta-point of view could note that the following block of an ordering link systematically generates a singleton, i.e. that the collection of objects to which the next rule is to be applied contains a single element. This occurs, for example, with the MetaController controller, which manipulates only singletons. (The meta-point of view can notice that the transmission in the next block is of the form Array with: anExpression.) The meta-point of view could then modify the controller so that it directly applies the next rule to the single element without starting up an iterative process.
5. A is not being specific enough. Focus A on lower-level details.

A final meta-point of view could, for example, automatically add an instance of a symbolic evaluation point of view to the meta-model trilogy, even if the user requests a more detailed animation. The reflexive aspect of the MPVC environment therefore favors an elegant implementation of multiple functions – meta-points of view – that a reflexive system must implement.

10.4 To conclude

The current MPVC environment is an experimental environment for problem solving. It offers multiple levels of abstraction, in terms of both the representation models that can be conceived and the reasoning models that can be implemented. The different aspects of problem statement and of rapid prototype development are all covered.

The environment has evolved since the writing of this book. It was first developed for Smalltalk-80, version 2.1. Since then it has been ported to version 2.5 and is being ported to release 4. Therefore it runs on all platforms that support this language (Unix, PC, Mac, etc.).

It has been used for several industrial projects. The first was a management software engineering environment for a large insurance company. The different MPVC trilogies that were developed control the consistency of the specifications written using the AXIAL methodology (derived from MERISE), simulate the logical behavior of the described applications, optimally handle memory allocation, edit different forms of documentation (e.g., requirements, or complete or partial specifications), and handle updates of the document base, taking into account the different versions.

The second was a prototype design toolkit for ergonomists. It allows non-computer scientists to describe control and reasoning mechanisms that should be implemented in the final product. The ergonomist can test the different mechanisms and validate them with the help of the user and the developer. Once the prototype has been completely specified, the design toolkit can generate a clear and precise programming specification.

The different perspectives that the MPVC approach offers show that a prototyping and programming environment is, above all, a problem-solving one and that it must allow many representation and reasoning models to be built and manipulated. We are convinced that future software engineering environments will all have these characteristics and that they will ensure a more efficient and more intelligent interface between humans and machines.
Bibliography


[76] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, Massachusetts, 1983.


Index

abstraction, 28
actigram, 89
actor, 17
Ada packages, 91
agent, 17
algorithm, 160
    inhibiting, 102
    subtracting, 156
analogy, 4, 6, 73
analysis
    semantic, 137
    syntactic, 137
animation
    cascade, 201, 205, 206
    graphical, 96, 97, 157, 162, 181, 200
application
    generic, 19
    interactive, 81
application skeleton, 50
approach
    analytical, 17
    interactionist, 17
    MPVC, 99
    object-oriented, 18, 19, 81
    structural, 17
    three-pronged, 207
aspect
    evaluation, 93
    mechanical, 93, 116
    semantic, 93
    assertion, 17, 211
    operational, 125
    assistant
        personalized, 205
    automatic generation, 209
    AXIAL, 214
    axon, 101
        final, 110
    behavior, 24
        default, 107
        intrinsic, 98
    Boolean algebra, 102
    break, 20
    browser, 201
    MPVC, 201
    calculator
        infix, 59, 62
        postfix, 32, 58
    canvas, 6
    category, 103
    cell, 101
    change
        of context, 70
        of reference, 74, 160
        of representation, 73
        of state, 69
        type-1, 69
characteristic
behavioral, 22
functional, 22
generic, 25
structural, 22

CHN, 91
class, 25
abstract, 103, 122
AnimationNetNeural, 157, 164
Boolean, 107
Calculator, 60
CalculatorInfix, 60
CalculatorPostfix, 33, 53, 60
CalculatorView, 53
Character, 125
Collection, 105
Controller, 47
ControllerMPWC, 123, 141, 180
ControllerNetNeural, 131, 151, 165
ControllerNNSynchro, 171
Date, 125
Dendrite, 105
DendriteExciting, 107
DendriteInhibiting, 107
Dictionary, 121
EvaluationByInhibition, 139, 155, 164
EvaluationBySubtraction, 156
EvaluationNetNeural, 155
False, 107
House, 23
Key, 36
KeyCmdImmediate, 41
KeyCmdSwitchable, 41, 56
KeyFuncBinary, 37
KeyFuncSpecial, 37
KeyFunction, 36
KeyFuncUnary, 37
KeyNum, 36
LinkOrdering, 120
Magnitude, 124
MetaConcept, 184
MetaController, 189
MetaCtrlForCompilation, 189
MetaCtrlForExecution, 189
MetaLink, 185
MetaPVAAbstract, 209
MetaPVAnimation, 200
MetaPVParallel, 198
MetaPVSequential, 196, 209
MetaRule, 185
Model, 43
MultiPV, 161
NetNeural, 112
NeuroEmitter, 110
Neuron, 104
NeuroReceptor, 111
Number, 125
Object, 43
ObjectImpulsive, 103
PointOfViewMPVC, 138
Process, 177, 179
Processor, 176
ProcessorParallel, 178
ProcessorSequential, 177
Rectangle, 27, 119
RectangleGraphic, 29
RuleOrdering, 118
Screen, 34
Semaphore, 177
SemaphoreCounter, 177
StandardSystemController, 50
StandardSystemView, 50
SwitchView, 52
TextView, 51
True, 107
UndefinedObject, 109
View, 47
coding, 12
cognitive atom, 17
cognitive structure, 17
cognitive unit, 68
command
immediate, 40
switchable, 40
completeness, 96
finite state automaton, 100
flow
  control, 89
  data, 89
  mechanism, 89
focal point, 4
frame, 17, 87
functional arguments, 51, 52
generator
  application, 18, 44, 81
  graphical interface, 81
generic, 16
global understanding, 160
graph
  inheritance, 30, 77, 113, 210
  ordering, 117, 121, 174, 183
  graphical workstation, 81
hidden function, 58
hierarchical aggregation, 89
impulsive state, 103
indirection, 180
industrial circles, 75
inheritance, 29
  multiple, 30
inhibition
  absolute, 157
  subtracting, 157
input–output configuration, 89
instance, 25
intelligent program, 182
interface, 25
  user, 82
interpretation layer, 100
interpreter, 141
KL-ONE, 30, 87
language
  class, 17
  natural, 17
  object-oriented, 21
  prototyping, 16
level
  abstraction, 214
  derived, 87
  meta-, 182, 203
  meta-meta-, 205
  model, 8
  object, 182, 203
  primitive, 87
  prototype, 8
link
  activation, 116
  causality, 182, 184
  consultation, 116
  dependency, 40
  ordering, 117, 120, 185
  reflexive, 149
management system
  integrated software, 125
  project, 76
mechanism
  dependency, 40
  inheritance, 29, 122
  programming, 108
  traversal, 99, 116, 165
memory
  intermediate, 64
  long-term, 64
  object, 22
  sensory, 64
  short-term, 64
MERISE, 214
message, 23
message passing, 21
meta-
  concept, 184
  controller, 173, 183, 186
  description, 183
  evaluation, 182
  link, 185, 187
  model, 183, 184
  point of view, 183, 186, 196, 212
  activation, 187, 212
  compilation, 186, 208, 209
transformation, 209
program, 182
rule, 185, 187
trilogy, 183, 199, 203, 212
vision, 199
method, 22, 23, 25
class, 53, 104
default, 107, 109, 124
design, 21
experimental prototyping, 207
programming, 21
methodology
MVC, 207
model, 3, 8, 210
anticipatory, 6
computational, 5
coupling, 5
empirical, 7
experimental, 8, 25
exponential, 4
figurative, 4, 5
analogy, 6
imitator, 6
objective reference, 5
simplifier, 6
final, 3
hypothetical, 7
life-size, 8
mathematical, 4
meta-, 183
MPVC, 99, 100, 183
MVC, 44
original, 3
partial, 8
reasoning, 214
representation, 214
simplified, 8
software, 9, 66
standard, 8
switchable, 52
theoretical, 4, 7
validated, 8
modeling, 7
motor

inference, 17
MPVC system, 141, 176, 180, 183
multi-evaluated, 100
multiple controller, 165
multiple points of view, 160, 205
multiprocessor, 89

net
Petri, 76, 77
neural, 100, 112
with memory, 167

network, 89
concept, 17
semantic, 17, 30, 87
neuro-emitter, 110, 126, 166
neuro-receptor, 110, 111, 127
neuron, 101, 104, 126

object, 21
composed, 77
primitive, 77

object box, 19, 50
object mold, 25
object-oriented database, 75
objective reference, 5
observer, 70
operator overloading, 137
order, 174
ordering
activity, 89
ordering, 201
point of view, 162
orthogonality, 174
para-technique, 119
parallel, 174, 198
parallelism, 179
parameter
behavioral, 165, 173
evaluatory, 173
structural, 173

part
dynamic, 82
static, 82
viewing, 83
PAWS, 89
perceived reality, 3
phase
deductive, 7
descriptive, 7
inductive, 8
interpreting, 7
predictive, 7
platform
for mechanical schemata, 208
for semantic schemata, 208
point of view, 59, 74, 92, 137, 154, 183, 208, 209, 213
default, 209
evaluation, 98
exclusive, 164
MPVC, 99, 137
temporal, 166
polymorphism
application, 86
subtyping, 21
popup menu, 133
prediction, 7
predictive theory, 5
problem, 66
problem solving, 66, 68
problem statement, 214
process, 177
independent, 174
transformational, 14
processing power, 22
processor, 176, 197, 199
multi-tasking, 176
parallel, 176
sequential, 176
single-tasking, 176
programming
data-directed, 22
object-oriented, 19, 21, 25
project management, 76
project management net, 77
proof, 12
propagation of constraints, 40
protocol, 104
prototype, 3, 8, 210
programming, 32
software, 10, 12, 25, 66
prototyping, 10, 12, 16, 210
rapid, 16, 21
software, 6, 10
psycholinguistics, 63, 64
psychologist, 182
psychology, 63
cognitive, 63, 64
industrial, 63, 65
organizational, 63, 65
programming, 65
query, 23
RCS, 76
receiver, 23
redirection, 162
reflexivity, 182, 211
long-range, 203
reframing, 70
reification, 182
reify, 33
relation
ako, 30
inheritance, 29
isa, 30
subsumption, 88
representation
external, 45
figurative, 8
internal, 45, 87, 207
knowledge, 15, 21, 71, 75
problem, 71
representation, 182
symbolic, 8
reservoir of knowledge, 29
reuse, 44
rule
formal, 7
ordering, 116, 118, 174, 185
production, 82, 83
SADT, 89, 95
Index 231

traversal, 212
structural operator, 76
subclass, 29
subclassResponsibility, 124
subsume, 122
superclass, 29
synchronization, 166
synchronous view, 181
system
  communication, 89
database, 89
formal verification, 20
MPVC, 100, 143
MVC, 44
operating, 89
parallel, 90
prototyping, 207
real-time, 90
reflexive, 214
test
  activation, 117
testing, 12
tool, 75
  assistance, 211
  interpretation, 93
  interpreter design, 207
  manipulation, 93
  modeling, 4
  prototyping, 18, 183
  simulation, 93
  software engineering, 9
  specialized, 8, 10
toolbox, 19, 50
toolkit
  design, 214
tracer, 20
transformation step, 210
transition, 76
transitive closure, 17
traversal
  compilation, 187
depth-first, 165
  execution, 187

SCCS, 76
scenario, 12
scheme, 17
  Boehm, 11
  model-view-controller, 44
  MPVC, 32
prefabricated, 19, 50
simplifying, 6
science
  exact, 3
  experimental, 3
scrolling window, 201
selector, 23
selector homonymy, 119, 121, 178
semantics
  compositional, 17
  evaluation, 74, 92, 98, 137, 173,
  183, 196
semaphore, 177
sequential, 196
sequentiality, 174
serial binary adder, 102, 165
signal, 177
simulation, 5, 12
sink, 110
sketch, 66
Society of Mind, 18
SODOS, 76
software
  experimental, 9
  object-oriented, 24
software engineering, 9
software life cycle, 9, 11, 76, 210
specialization, 29
specific, 16
specification, 12
  informal, 68
speculative attitude, 208
speculative reasoning, 6
Statecharts, 90
statistics, 89
stepper, 20
strategy
  interpretation, 212
of neural net, 126
parallel, 180
sequential, 180
trilogy
MPVC, 92, 116, 173
MVC, 44
type
abstract, 21, 25
typing
dynamic, 26
static, 26
understanding
abstract, 182
symbolic, 182
user, 9

variable
class, 103
instance, 25, 103
shared, 104
version history, 211
video trace, 200
view
MVC, 45, 46
switchable, 202
view hierarchy, 46, 52
virtual copy, 29
WQN, 89, 95
XOR, 114
Prototyping with Objects presents an in-depth study of rapid software prototyping, highlighting the role of object-oriented languages and their associated software methodology in designing and implementing a prototype.

Key features include:
- a series of ready-to-use Smalltalk classes used throughout the book to implement Minsky's neural nets
- clear presentation of object-programming theory to improve the reader's understanding of how to create, use, combine and modify objects
- explanation of the cognitive framework for prototyping
- study of state-of-the art programming environments

The first part of the book, Prototyping and Object-Oriented Languages, looks at the epistemological, scientific and industrial contexts for prototyping. The concepts of model and prototype are formalized and the role of prototyping in the software life cycle is made clear. The second part looks at the Model–Point of View–Controller (MPVC) approach, which is presented as an object method that simplifies the implementation of tools for interpreting, manipulating and animating representation models.

Prototyping with Objects will be essential reading for the Smalltalk programmer, object-oriented programmer and the software engineer.

Philippe Krief is Technical Director of ACKIA Expert Systems in Boulogne.