Predictive Software*

JOSÉ HERNÁNDEZ-ORALLO jorallo@dsic.upv.es
Department of Information Systems and Computation, Universitat Politècnica de València, Valencia (Spain)

M. JOSÉ RAMÍREZ-QUINTANA mramirez@dsic.upv.es
Department of Information Systems and Computation, Universitat Politècnica de València, Valencia (Spain)

Abstract. We examine the adaptation of classical machine learning selection criteria to ensure or improve the predictiveness of specifications. Moreover, inspired in incremental learning, software construction is also seen as an incremental process which must generate and revise the specification with the main goal of being predictive to requirements evolution. The new goal is not necessarily to achieve the highest accuracy at the end of a first prototype or version, but to maximise the cumulative benefits obtained throughout the entire software life-cycle. This suggests a new software life-cycle, whose main characteristic is to move modifications earlier, by using more eager inductive techniques, and reducing overall modification probability. This new predictive software life-cycle is particularised for the case of (functional) logic programming, placing the deductive/inductive techniques necessary for each stage of the life-cycle. The maturity of each stage and the practical possibilities for a (semi-)automation of the cycle based on declarative techniques are also discussed.

Keywords. Software Development, Machine Learning (ML), Inductive (Logic) Programming (ILP), Predictive Modelling, Data-Mining, Software Life-cycles, Hypotheses Selection Criteria.

1. Introduction

The software process defines the way in which software development is organised, managed, measured, supported and improved. Nowadays it is widely accepted that the key to successful development lies in the effective management of the software process. This emphasis on process and the initiatives to measure their maturity (Humphrey 1990, Kuvaja 1994) reflects an evolution of the concept of software quality from the traditional *verification* and *validation* approach towards process-focused environments.

The development process is supported by the construction of explicit models. A process model is a special kind of process representation in a suitable notation or formalism. During the last three decades, the study of software production processes has led to the development of various software life-cycle models that have been employed to some extent in software engineering (stage-wise, waterfall, transformational, evolutionary and spiral models, Pressman 1997).

The main functions of a software life-cycle are to determine the dependencies between the stages involved in software development and evolution, and to establish the transition criteria for progressing from one stage to the next. These life-cycle models help engineers to determine the order of global activities in the production of software. However, in our opinion, the main drawbacks of these models (including those based on formal methods) are due to a misconception of the nature of software and its evolution. It is still not infrequent to read

* This work has been partially supported by CICYT under grant TIC 98-0445-C03-C1 and Generalitat Valenciana under grant GV00-092-14.

conceptions of software "from specification to final product", which do not take maintenance or generation of that specification into account.

Fortunately, there is an increasing interest in requirements elicitation and evolution as the most important topics in software engineering. The following words, written by Fred Brooks more than a decade ago, have been used elsewhere (Berry and Lawrence 1998) to illustrate the point: "*The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements ... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*"

Therefore, a more suitable approach to software development should emphasise the construction of a *good* requirements model. The term 'good' means a model which minimises software modification probability, particularly in the later stages. The idea is to 'predict' requirement evolution as much as possible in order to minimise the 'remake' of software as a trace of this evolution. It should be explicitly stated that this predictive character of the model must be preserved throughout the remainder of the life-cycle: the design must be conceived to maintain the generality of the model, validation must be made according to this general model, and, more importantly, future modifications must consist of coherent revisions, not extensional 'patches' to the model. The reason is clear: the later the modification the costlier it is.

It seems that specific models should be avoided, in order to decrease future changes and in order to favour reusability (software development is expensive, time-consuming and a highly repetitive process). The study and development of inductive techniques for generalising the requirements to make them more predictive looks highly suitable. Before designing new techniques and languages for further generalisation, an empirical and theoretical study on when this generalisation of the model is useful and how it should be done would be necessary. Predictive modelling and machine learning (ML), in particular, and the philosophy of science, in general, provide us with very useful and corroborated tools (and terminology) for selecting the most likely model or the most informative one.

In (Hernández and Ramírez 2000) we showed the benefits of adapting the paradigm of scientific theory construction to software. An analogy between software development and theory construction is presented. It suggests many equivalences to be worked on.

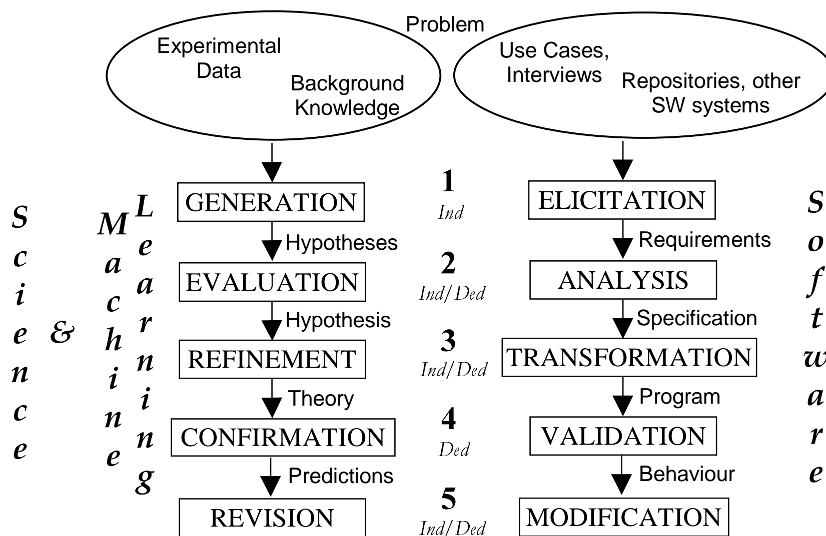

Figure 1.1. Main stages in scientific theories and software systems development

Many results from one field could be usefully applied to the other; in particular, when and how different reasoning techniques (induction, abduction[1], deduction) should be used.

Philosophy of science and software engineering are complementary in experience and techniques because they have focused primarily on different stages. Hence, the generation and evaluation of hypotheses (phases 1 and 2) have been traditionally addressed by the former, whereas design, coding, testing and debugging (phases 3, 4 and 5) have until now been priority phases for software development. Only recently have the generation and evaluation of hypotheses been included in the software construction paradigm, under the banner of "requirement engineering". However, it is usually not recognised in the literature that the techniques should be mainly inductive. The information transmitted from the user to the developer is incomplete and inexact and the developer must complete and explain it by inductive methods. The relevance of this inference process has recently given rise to a different approach to software engineering: inductive programming (Partridge 1997).

In this paper, we examine the adaptation of classical ML selection criteria to software systems to distinguish predictive models (or specifications) from non-predictive ones. Moreover, inspired by incremental learning, software construction is also seen as an incremental generation process. The new goal is to minimise the maintenance cost of a software system by selecting a predictive model of requirements, with a much more long-term view of the entire software process. This suggests a new software life-cycle, whose main characteristic is to move modification probability earlier, by using more eager inductive techniques. The induction of software specifications can be carried out for simple cases only in a declarative environment. In particular, we will show that it is suitable for classification problems. We must clarify that with regard to automation we show that the evaluation/selection stage can be automated for large-scale problems but that the "predictive software" claim is not exclusively an automated paradigm. It is a motivation for using *eager* induction methodologies (which can be automated) in software engineering.

The rest of the paper is organised as follows. First, in Section 2, we recall previous applications of ML to software engineering and highlight some other issues that still have not been applied, introducing some additional concepts and ML features. Next, in Section 3, we define predictive software and relate it to adaptive software and intelligent software, and discuss the convenience of either eager or lazy methods. Two examples are shown to illustrate the (automated) selection of the most predictive model. Section 4 defines the predictive software life-cycle and specialises it for declarative programming, discussing the stages which are mature for automation. Section 5 includes an example of the automation of the whole cycle and a discussion of partial automation of more complex problems. Section 6 presents other ML techniques which can be applied to other kinds of software systems. Finally, Section 7 closes the paper with an overall discussion of the benefits of accepting the "predictive software" paradigm, even though some stages would still be manual or semi-automated.

2. Machine Learning and Software Engineering

Several learning or inductive techniques have been applied in software engineering, either in CASE tools or in operative parts of full systems. From time to time, the idea has been revived,

[1] In what follows, abduction will be considered a special case of induction when the learning process is more related to the assumption of factual hypotheses than the construction of general ones. In some cases, revision can also be more abductive than inductive, when the revision consists of a simple patch or the assumption of extensional facts (see e.g. Flach and Kakas 2000).

but efforts on real applications run into the problem that induction is computationally much harder than deduction.

2.1. Previous work

At present, there is already an incipient and productive (although not explicit) transfer between ML and software engineering of *several* software systems. This has been done by comparing the construction of past theories or systems and using this experience. As an example, experienced software engineers use analogy as a very powerful method to translate previous situations, problems, decisions and results into a situation which is different but somewhat similar. The idea of automation dates back to the 'Programmer's Apprentice', an automated programming system designed by Rich and Shrobe (Rich and Shrobe 1978), where some analogical reasoning was used from previous programming experiences. Some more recent approaches have also attempted to use automated analogy for helping to reuse specifications (Tessem et al 1994).

However, the generation and evaluation of hypotheses in full programming languages from extensional or partially intensional data have been successfully undertaken in the field of Inductive Logic Programming (ILP). ILP is a framework of supervised learning in first order logic (Muggleton 1991, Lavrac and Dzeroski 1994, Muggleton and De Raedt 1994). ILP has been applied to different kinds of software engineering problems (Bratko and Dzeroski 1995): program development from high level specifications through data reification and the construction of invariants that can be used for proving correctness of procedural programs. With regard to the testing phase, ILP techniques can also be used to generate an adequate set of test data for a logic program.

The use of ILP for automatic program synthesis has been suggested elsewhere (Bergadano and Gunetti 1995) but it is only at the present time that it can be considered able to cope with non-toy problems. To apply the paradigm to complex problems, strong constraints on the clauses that could possibly be generated must first be established. Then, inductive inference selects a target (Prolog) program that is consistent with the constraints and the examples. In other words, the basic skeleton of the target program is given, and the induction system will fill in the missing details on the basis of the given examples (inductive program completion). One of the great advantages of this approach is that if the application schema does not change completely, the program can be revised for a new goal problem by modifying only the examples (software maintenance and reusability).

Nonetheless, the applicability of ILP techniques to automatically obtain correct programs from *complex* specifications consisting of a *few* examples of their input/output behaviour is questionable. (Flener and Yilmaz 1999) claim that at the current state of the research in this area, the above objective can only be achieved for very specific or small problems.

2.2. Issues from Machine Learning

As has been commented on in the previous subsection, in the last two decades many ML applications have emerged in the field of software engineering, adapting ML techniques to improve specific parts of the software life-cycle. However, most of the techniques, paradigms and theoretical results produced in ML have not been fully exploited to date for the whole process of software development.

In particular, we will review the most important learning paradigms which have appeared in the ML literature. We will centre on the problem of hypothesis selection in induction, and finally, we will highlight the distinction of character between eager and lazy methods. Since we only

briefly discuss those issues that will be used in subsequent sections, we refer to (Mitchell 1997) for a complete view of the field.

The issues discussed below will allow us to define a new software quality factor, predictiveness, according to the selection criterion that would minimise hypothesis (or program) changes. It will also allow the introduction of a clear characterisation between eager and lazy software systems and the introduction of a new life-cycle, which is inspired by ML paradigms.

2.2.1. Learning Paradigms

The first and most important learning paradigm was introduced by Gold in his seminal paper (Gold 1967). In an *incremental learning* session, where an infinite sample is gradually given example by example, the goal of the learner is to obtain a hypothesis at a certain point that will not change for future examples (i.e. will predict all of them correctly). Since this point of convergence is not pre-determined, the paradigm is called *identification in the limit*.

Initially, very strict requirements were assumed (exact identification, lack of interactivity and worst-case consideration). Consequently, the learnability results obtained were negative for many families of languages. In order to allow more positive results, other paradigms have been proposed. The *PAC-learnability model* (Valiant 1984) was a reaction to bringing ML to tractability by relaxing the exact character of the identification. In order to do this, the identification should not be exact but Probably Approximately Correct (PAC). The model has also been modified to consider the efficient learning of universal representations such as logic programs, as represented by the U-learnability model by (Muggleton and Page 1999).

Another approach is based on the interaction between the learner and the environment. This allows the introduction of new examples that the learner generates and are not present in the given evidence (sample). This paradigm is called *Query Learning* and was introduced by (Angluin 1987). The key issue of query learning is how the learner is able to make good questions in order to optimise the learning process.

However, it is not until recently that the paradigm of identification in the limit has been questioned. The new goal is not necessarily to achieve the highest possible accuracy at the end of a learning session, but rather to maximise the cumulative benefits obtained throughout the learning session (Abe 1997). Now, it is not only whether and when the final model or hypothesis (if it exists) is found which is important but how many times the model has been changed. Moreover, it is also important how much it has been changed. This has generated an *incremental-revisionist* trend in ML and knowledge acquisition (Katsuno and Mendelzon 1991). When new observations are received, three situations are possible: *prediction hit*, *novelty* and *anomaly*. The first situation, prediction hit, is the clearest one. The theory is more validated than before and no revision to it is necessary. The last situation, anomaly, is also quite clear. The new evidence is inconsistent with the theory[2]. Consequently, it must be revised. However, this revision can be done in many ways: by a simple patch or by a coherent (general) revision. Finally, in the same way, a novelty (i.e., an equation which is not covered by the hypothesis but it is consistent) can be incorporated into the theory by a simple extension (a patch) or by a coherent extension, which may motivate the revision of the theory.

The following subsections discuss the two factors which influence the frequency and character of revisions, the selection criteria (conservative or explanatory) and the inductive method (lazy or eager).

[2] For classification problems, we have not addressed the case of having different misclassification costs.

2.2.2. Selection Criteria

A fundamental question for the learning task is whether there is any way to know, a priori, when a given hypothesis will be satisfied by future experiences in a given context.

If we know the initial distribution of hypotheses in that context, the plausibility of the hypothesis can be obtained in a direct way under a Bayesian framework. Since this initial distribution is generally unknown, many different measures of quality of theories have been proposed in order to use them as criteria for theory selection. From these, there are two main trends: *descriptional or conservative induction*, which is usually related to the simplicity criterion or Occam's Razor, and, *explanatory induction*, which is more closely related to coherence, cohesion or 'consilience' criteria. Although they are not exclusive trends, the literature is full of discussions and support for each of them (Barker 1957, Solomonoff 1964, Harman 1965, Hempel 1965, Ernis 1968, Thagard 1989, Sharger and Langley 1990, van den Bosch 1994, Li and Vitányi 1997).

The first trend can be associated to the view of learning as information compression[3], which was first formalised by (Solomonoff 1964). The principle of simplicity was revived when, later on, Rissanen introduced the popular Minimum Description Length (MDL) principle (Rissanen 1978). In its later formulation (Barron et al. 1998), the MDL principle advocates that the best description for a given data is the shortest one (summing up the model and the exceptions). Apart from all the methodological advantages of simplicity, and keeping in mind that the principle is not computable in general (it can only be used if we restrict the descriptive mechanism), the major reason for using the MDL principle is that it usually avoids over-specialisation (under-fitting) and over-generalisation (over-fitting). From here it is usually argued that "*the shorter the hypothesis the more predictive it is*". However, some parts of the model cannot be compressed at all, as they are extensional parts with no predictive character. This is even more frequent in incremental learning, because the hypothesis gets patched repeatedly until the length of coding the exceptions forces the appearance of a new radical hypothesis.

On the other hand, explanatory or intensional models are characterised by a well-balanced compression ratio, i.e. no intrinsic exceptions (patches) are allowed in the theory. In other words, all the data must be explained. In order to justifiably state that a theory that has been tested is robust and reliable it must be *intrinsically refutable*. Only in this case, if experimentation fails to refute it, can the theory be considered as confirmed or 'validated'.

In a similar way, consilience, as coined by (Whewell 1847), comprises the relevant basics in scientific theories: prediction, explanation and unification of fields. Consilience refers to the idea that the data must be covered by the same general rule, i.e., the evidence is unified by the theory. In this sense, consilience turns out to be stricter than intensionality.

Since a minimal revision (see e.g. Mooney 1997) is usually less costly than a deep revision, the trade-off between both paradigms is clear. While the conservative paradigm is usually less costly in the short term because it reduces the scope of revisions, the explanatory paradigm has advantages in the long term because it reduces the number of revisions.

---

[3] It is important to clarify that maximum compression (represented by the MDL principle) is not the same thing as elimination of internal redundancy. The rationale is that different equivalent models can have very different sizes, which are exactly equally predictive. A model has no redundancy if the same theory cannot be expressed in a shorter way. In this way, intensional or consilient models are compatible with the maxim of elimination of redundancy. The MDL principle should be better understood as "it is sufficient [...] to pick a hypothesis that is asymptotically shorter than the examples rather than pick the simplest hypothesis" (Natarajan 1991, p. 198).

2.2.3. Lazy and Eager Learning Methods

A second important question in induction is when and how the computational investment is to be made in the generation of the hypotheses. That is, the theories can be constructed either when new problems require new predictions or whenever possible. This difference has characterised the two most important families of inductive methods in ML: lazy methods and eager methods. A quite up to date comparison of both approaches can be found in (López and Armengol 1998).

The paradigms discussed in the previous subsection can be considered as belonging to the group of *eager* learning methods, just differing in their degree of eagerness. Eager learning extracts all the regularity from the data in order to work with intensional knowledge, i.e., a model. Examples of eager learning are Model Based Reasoning (MBR) and ILP.

However, very important and productive research has taken place for *lazy* learning methods (Aha 1997). Examples of lazy learning methods are: *k*-neighbouring or distance-based techniques, case-based reasoning (CBR) or instance-based reasoning and analogical reasoning (AR). Lazy methods usually *memorise* the examples as extensional knowledge with some information about their results (or class) and other characteristics. Whenever a query or a new problem appears, the system works hard to extract which previous experiences are more appropriate/similar to the new problem by making the most plausible analogy.

The advantages of lazy methods are their flexibility and the economy of resources in the short and mid terms, because a reasoning effort is only done when a new problem appears. Another important advantage is that revision is unnecessary, because no model of reality is constructed.

In contrast, the great advantage of eager methods is that, since they are constantly pre-processing all the received information, they can take advantage of idle time resources. If the model is accurate, the answer to a new problem is immediate. Moreover, most of the given examples can be forgotten, when their model is reliable enough, reducing storage and improving manageability in the large.

3. Predictive Software

Under the *incremental-revisionist* view of learning seen before, a predictive model attempts to minimise the number (and possibly the scope) of both anomalies and novelties. In the software engineering terminology, anomalies generate corrections to the specification, and novelties generate extensions to it. Both corrections and extensions are known as modifications.

Consequently, one way to improve the economics of software (Boehm 1981) is the reduction of modifications[4]. In this way, the goal of a software developer team is to devise software that does not need to be modified frequently. More precisely,

> DEFINITION 3.1 PREDICTIVE SOFTWARE
> A software system is *predictive* if it is stable for evolving requirements in the same context where the specifications originated. The term 'stable' means that the frequency of modifications is minimised.

The notion is not new, since it is often said that a good software developer should predict future requirement changes, devising more general (and easily adaptable) specifications. The concept of

[4] The other way is to reduce the scope of each modification. Whether and how this is compatible with the reduction of the number of modifications is not considered in this paper. Programming languages and techniques have evolved towards paradigms, such as polymorphism, that avoid the use of cases (extensional parts or patches), in order to minimise the scope of each modification.

predictive software differs from other *similar* concepts that have recently appeared, such as adaptable, smart or intelligent software.

Adaptable Software (Lieberherr 1996), understood as modifiable software, is unmistakably distinguished from predictive software in the sense that a software can be very adaptable, usually in a manual way, but it is constantly being revised/modified due to prediction failures[5].

Intelligent Software systems are usually characterised by the use of AI techniques (knowledge representation, modal logics, knowledge-based system tools, automated theorem proving, reverse engineering, etc.) to assist software development or to be included in the system itself. In this sense, expert systems can be considered intelligent software, but from the ML perspective, they are just 'idiots savants'. As a result, *intelligent software* and *predictive software* are completely different notions, especially in regard to purpose and the temporal-scale of the inductive techniques involved. Predictive software requires an eager methodology, which can be either manual or automated, to discover the regularity from the requirements in order to produce a predictive model. In contrast, adaptable and intelligent software is usually based on lazy techniques.

A lazy system lacks a model and therefore does not need to be revised as it is completely adaptive and adaptable. In the last section of this paper, we will discuss some software systems such as knowledge-based systems that may be more appropriately devised using lazy methods. Nevertheless, lazy methods are unsuitable in the large for most problems, because the response time increases as more previous cases are to be considered to give the result to each problem, as is shown in Figure 3.1.

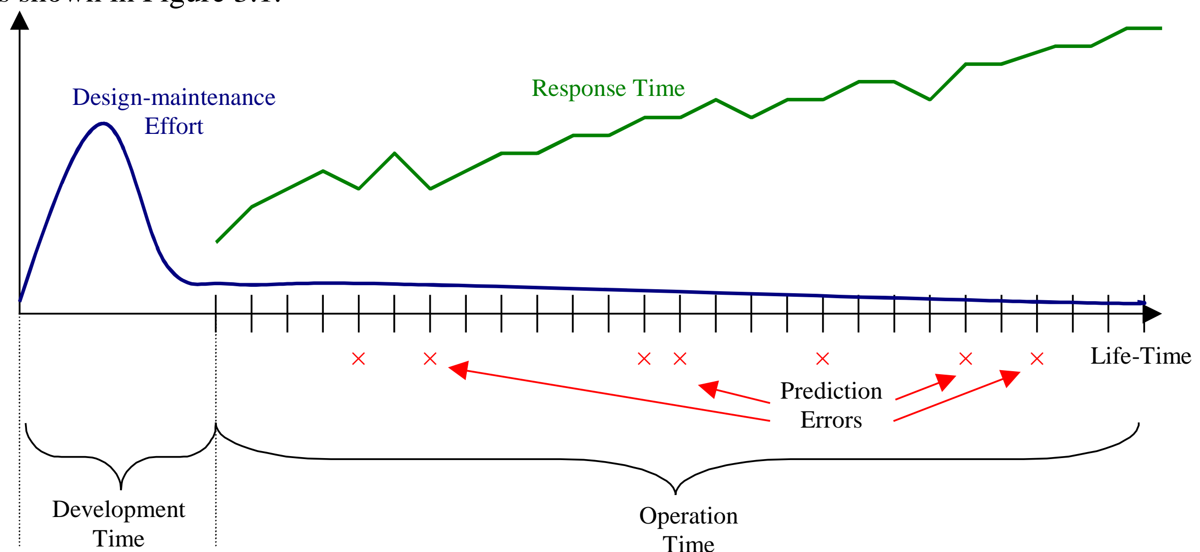

Figure 3.1. Design-maintenance Effort and Response Time in Lazy Methods

On the other hand, eager methods work with a model, and response time is given by the deduction of each future case with respect to the model. The response time can remain almost constant in the large, because the model is remade to conciliate the novelties and anomalies, as shown in Figure 3.2. Moreover, prediction errors will be less numerous in the case of eager

[5] Another interesting question is whether a model can be constructed to ensure a high probability that modifications will be easy to adapt. In other words, a *predictably adaptable* model would be a model whose probability of modification is not necessarily reduced and whose overall adaptability is not high either, but its adaptability is distributed in such a way that most modifications can be done easily and, hence, the maintainability is still high. To our knowledge, there are no ML methods which are specifically designed to deal with this issue.

methods than lazy methods in the large provided the unknown model can be identified in the limit.

Predictive software, in the end, emphasises a more long-term view of the software process. A great effort must be invested in the requirement elicitation stage in order to obtain a durable model. Obviously, despite this initial effort, some modifications will eventually have to be made. Within the same philosophy, each revision (modification) must be done carefully and consistently with the rest of the model in order to preserve the predictiveness of the model.

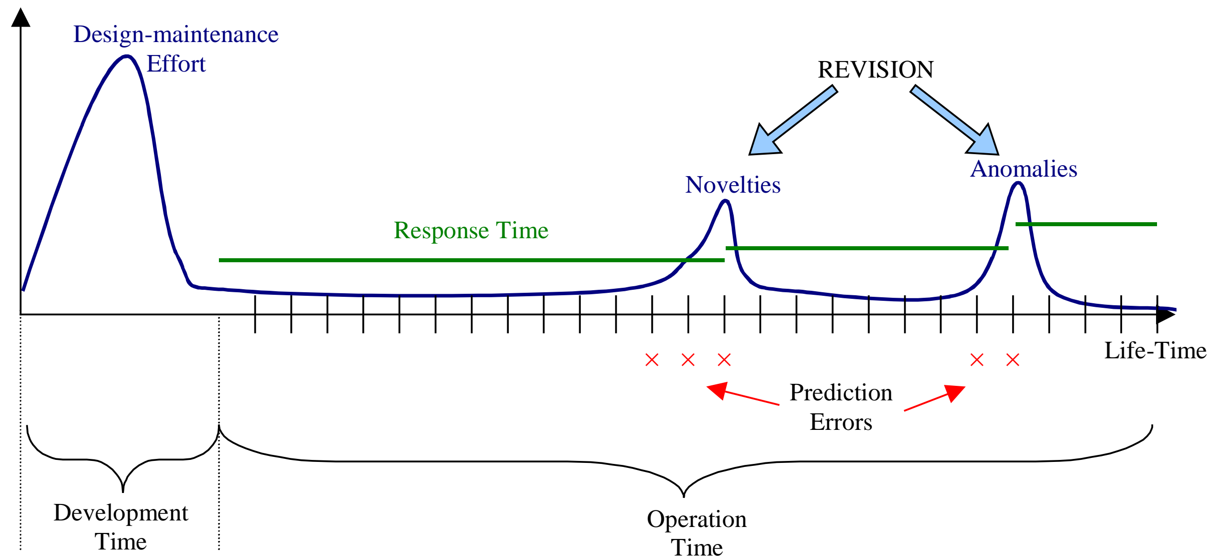

Figure 3.2. Design-maintenance Effort and Response Time in Eager Methods

Even under an eager characterisation, different degrees of eagerness or anticipation are possible. The degree of eagerness is precisely determined by the evaluation criterion, which establishes whether patches are allowed in the model or not (revisions are easier but more frequent). The most eager evaluation criterion turns out to be the intensionality criterion (characterised by anticipating or 'investing' in more complex theories). It is still more eager than the MDL principle, which is sometimes too conservative. According to the software counterpart to Popper's paradigm, a software system should only be considered 'validated' if it has remained unchanged for a varied sample of tests. More importantly, it must be intrinsically refutable. General and intensional models, not including exceptions or patches (which cannot be refuted), are then preferable for software practice. Only in this case, the validation of the system can be considered more reliable if a test set has been passed satisfactorily.

In the following subsections, we show how ML selection criteria can be used in software engineering to compare models in order to distinguish the most predictive one.

3.1. Example

The first example has been chosen in order to compare between two extreme cases of requirements models, although there are many other different models for this problem and many different combinations of both. In addition, we will use a declarative programming language since it is possible to know which part of the model covers each part of the examples given. This property makes the evaluation of the model easy and can be automated.

A university library gathers the information about the journals it is subscribed to and the articles that appear in them. The system allows the search by journal name, paper title and year,

as well as author(s) name. However, no information about the area or field of each paper is included in the database. A new query system is to be developed to allow for the search of journals and papers by field.

Suppose that part of the journal papers have been classified manually. Now the goal of the application is to suggest the field for the rest of the papers. Part of the initial database (scientific papers in computer science journals) is read and converted into the schema illustrated in Figure 3.3.

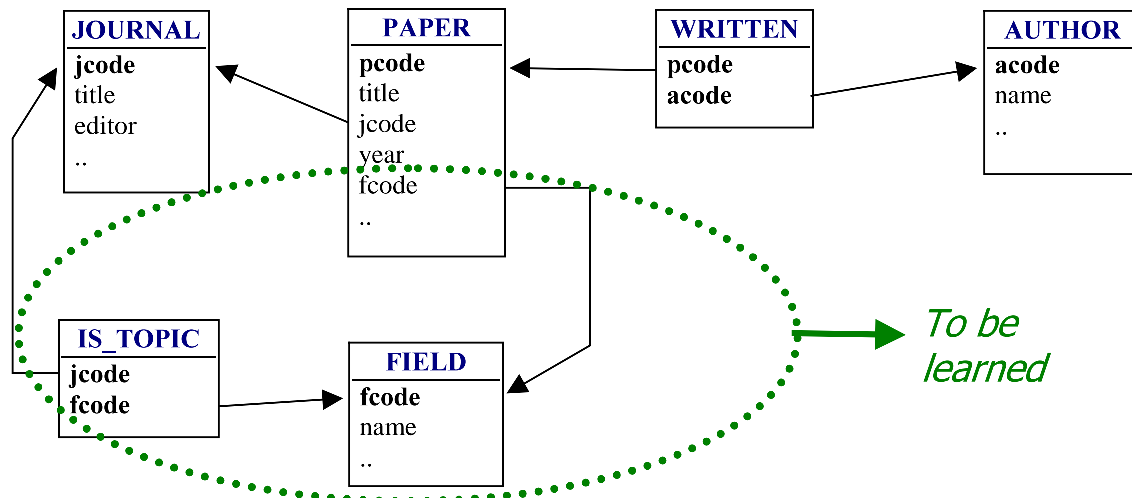

Figure 3.3. A Database Schema for The Field Query Problem

To better illustrate the point and trace the subsequent models, let us show a brief excerpt from the data:

J.jcode	J.title	P.pcode	authors[6]	F.fcode	F.name
ASE	Automated Software Engineering	1	Penix; Alexander	SW	Software Engineering
ASE	Automated Software Engineering	2	Chatzoglou; MacAulay	SW	Software Engineering
ASE	Automated Software Engineering	3	Cousot; Cousot	LP	Logic Programming
JLP	Journal of Logic Programming	4	Muggleton; De Raedt	ML	Machine Learning
JLP	Journal of Logic Programming	5	Lloyd	LP	Logic Programming
JLP	Journal of Logic Programming	6	Sannella; Wallen	LP	Logic Programming
JLP	Journal of Logic Programming	7	Cousot; Cousot	LP	Logic Programming
AAI	Applied Artificial Intelligence	8	Blockeel; De Raedt	DM	Data Mining
TCS	Theoretical Computer Science	9	Roscoe; Hoare	SW	Software Engineering
MM	Minds and Machines	10	Fetzer	SW	Software Engineering
DKE	Data & Knowledge Engineering	11	Lopez de Mant.; Armengol	ML	Machine Learning
CACM	Communications of the ACM	12	Valiant	ML	Machine Learning
CACM	Communications of the ACM	13	Hoare	SW	Software Engineering
CACM	Communications of the ACM	14	Fayyad, Uthurusamy	DM	Data Mining
CACM	Communications of the ACM	15	Fetzer	SW	Software Engineering
CACM	Communications of the ACM	16	Genesereth; Ketchpel	SW	Software Engineering
CACM	Communications of the ACM	17	Muggleton	ML	Machine Learning
DEB	Data Engineering Bulletin	18	Fayyad	DM	Data Mining
NGC	New Generation Computing	19	Muggleton	ML	Machine Learning
MLJ	Machine Learning Journal	20	Lopez de Mantaras	ML	Machine Learning
MLJ	Machine Learning Journal	21	Muggleton	ML	Machine Learning
MLJ	Machine Learning Journal	22	Angluin	ML	Machine Learning
AI	Artificial Intelligence	23	De Raedt; Dehaspe	ML	Machine Learning

[6] `A.name` is shown aggregated.

This data is generated by the following query:

```
SELECT J.jcode, J.title, P.pcode, A.name, F.fcode, F.name
FROM JOURNAL J, PAPER P, WRITTEN W, AUTHOR A, FIELD F
WHERE J.jcode = P.jcode AND P.pcode = W.pcode AND W.acode = A.acode AND P.fcode = F.fcode;
```

The first relation to be learned is `is_topic`, which is easily extracted as follows:

```
%    topics of journals
is_topic(J, F) :- paper(_, _, J, _, F).
```

which means that the topics of a journal are all the topics of its papers. This is possible because the relation between journal and topic is many-to-many, by the use of the extra relation `is_topic`. On the other hand, the problem of classifying the main field of a paper is more complicated because of the uniqueness restriction. Moreover, apart from the sample, there will be papers which will be manually classified and others will have to be classified automatically. To differentiate them, the classified papers are denoted by the predicate `paper`, and the papers to classify are denoted by `paper_`. From here, the following models can be induced:

MODEL ***A***:

```
%%%%%%%%% Learnt rules

%    Muggleton’s articles are about ML.
paper_(P, _, _, _, ‘ML’):- written(P, A), author(A, ‘Muggleton’), !.
%    ‘JLP’ papers field is ‘LP’ except from Muggleton’s articles handled above:
paper_(P, _, ‘JLP’, _, ‘LP’).
%    MLJ articles are always about ML.
paper_(P, _, ‘MLJ’, _, ‘ML’).
%    Valiant’s articles are about ML.
paper_(P, _, _, _, ‘ML’):- written(P, A), author(A, ‘Valiant’), !.
%    Fayyad’s articles are about DM.
paper_(P, _, _, _, ‘DM’):- written(P, A), author(A, ‘Fayyad’), !.
%    ‘CACM’ papers field is ‘SW’ except from Valiant’s, Fayyad’s and Muggleton’s:
paper_(P, _, ‘CACM’, _, ‘SW’).

%%%%%%%%%% Exceptions  // non-predictive
paper_(1, _, _, _, ‘SW’).
paper_(2, _, _, _, ‘SW’).
paper_(3, _, _, _, ‘LP’).
paper_(8, _, _, _, ‘DM’).
paper_(9, _, _, _, ‘SW’).
paper_(10, _, _, _, ‘SW’).
paper_(11, _, _, _, ‘ML’).
paper_(23, _, _, _, ‘ML’).
```

MODEL ***B***:

```
%%%%%%%%% Learnt rules ordered by priority (strength)

%    dependency between authors’ other papers field:
paper3(P, _, J, _, F) :- written(P, A), written(P2, A), paper(P2, _, J2, _ F),
                         written(P, A2), written(P3, A2), paper(P3, _, J3, _ F),
                         P <> P2, P <> P3, A <> A2.
%    dependency between author’s other paper field:
paper2(P, _, J, _, F) :- written(P, A), written(P2, A), paper(P2, _, J2, _ F).
%    dependency with journal field:
paper1(P, _, J, _, F) :- is_topic(J, F).
%    Unclassified papers follow these rules:
%    ‘JLP’ papers field is ‘LP’:
paper0(P, _, ‘JLP’, _, ‘LP’).
%    ‘ASE’ papers field is ‘SW’:
paper0(P, _, ‘ASE’, _, ‘SW’).

%%%%%%%%%%  uniqueness restrictions for the field of a paper
paper_u1(P, _, _, _, _) :- paper1(P, _,_ ,_ , F1), paper1(P, _,_ ,_ , F2),
                           F1 != F2, !, fail.
paper_u1(P, _, _, _, F) :- paper1(P, _,_ ,_ , F).
paper_u2(P, _, _, _, _) :- paper2(P, _,_ ,_ , F1), paper2(P, _,_ ,_ , F2),
```

```
                          F1 != F2, !, fail.
paper_u2(P, _, _, _, F) :- paper2(P, _,_ ,_ , F).
paper_u3(P, _, _, _, _) :- paper3(P, _,_ ,_ , F1), paper3(P, _,_ ,_ , F2),
                          F1 != F2, !, fail.
paper_u3(P, _, _, _, F) :- paper3(P, _,_ ,_ , F).

%%%%%%%%%%%%  priorities
paper_(P, _, _, _, F) :- paper_u3(P, _,_ ,_ , F), !.
paper_(P, _, _, _, F) :- paper_u2(P, _,_ ,_ , F), !.
paper_(P, _,_ ,_ , F) :- paper_u1(P, _,_ ,_ , F), !.
paper_(P, _,_ ,_ , F) :- paper0(P, _,_ ,_ , F), !.
```

In a semi-automatised ILP environment, both models would possibly need some meta-information to be generated (mainly the cuts for uniqueness in the second model), but they are not far from the possibilities of some ILP systems. Clearly, model A is much shorter and simpler than B, and would be chosen according to descriptive selection criteria, such as the MDL principle. However, model A is full of exceptions (extensional patches) and will fail to classify many new papers. Moreover, some rules are based on particular cases and will have to be revised soon, as more papers are added to the database.

In contrast, model B is much more complex, but it is also much more intensional, in the sense that it does not include many particular cases. Hence, the second model should be preferable. Maybe the choice does not seem difficult for this easy example by manually inspecting the code, but the techniques of how to select the second from the first one are known in the ML literature and philosophy of science and, more importantly, can be automated provided that the representational language is model-based, such as in declarative languages (see for instance Hernández 2000 for an effective measure applicable to this problem).

3.2. Automated Selection Example

Let us show with a second example how this model selection can be done automatically. We will use the FLIP system for this. The FLIP system (Ferri et al. 2000) is an application built in C which implements the Inductive Functional Logic Programming framework (Hernández & Ramírez 1998) (Hernández & Ramírez 1999). At its actual stage of development, the FLIP system allows conditional functional logic programs for background knowledge and hypothesis. Functional logic programs are a step forward in declarative programming because they subsume logic programs but allow a better handling of functions. The system works with two sets of facts: the positive examples and the negative ones and optionally an initial set of theories and background knowledge. Hypothesis selection is guided by the criteria which we have commented on in the first sections: simplicity and consilience.

The FLIP system is a versatile application that can operate as a pure induction system, a theory reviser and a theory evaluator. When there is no initial program, the system behaves as a generator of hypotheses which only outputs the best solution according to the FLIP selection criterion. In the case that different initial theories are supplied to FLIP and there are still new examples to consider, the system behaves as a reviser, which can modify or extend the initial theories according to the novelties or anomalies that the new examples could trigger. Finally, in the case that different initial theories are supplied to FLIP and there are no new examples to consider, the system behaves as a theory evaluator wrt. past examples. For this example, we use FLIP as a theory evaluator.

Consider the following set of examples of good clients of an insurance company:

```
goodc(p(p(v,woman),tall))=true,
goodc(p(p(v,nurse),woman))=true,
goodc(p(p(p(p(v,has_children),woman),joan),speaks_spanish))=true,
goodc(p(p(p(p(v,jane),woman),speaks_portuguese),tall))=true,
goodc(p(p(p(p(p(v,susan),has_children),teacher),woman),high_income))=true,
goodc(p(p(p(v,married),teacher),has_cellularphone))=true,
goodc(p(p(p(v,anthony),man),teacher))=true,
goodc(p(p(p(p(p(v,teacher),low_income),atheist),has_cellularphone),married))=true,
goodc(p(p(v,john),has_children))=true,
goodc(p(p(p(v,browneye),likes_coffee),has_children))=true,
goodc(p(p(p(v,has_children),nurse),cellular_phone))=true,
goodc(p(p(p(v,jane),plays_chess),has_children))=true,
goodc(p(p(p(p(v,mary),speaks_spanish),has_children),has_cellularphone))=true
```

where ‘p’ is the list constructor. From here, the following two theories have been generated:

MODEL ***A***:

```
goodc(X0) = member(has_children, X0)
goodc(X0) = member(woman, X0)
goodc(X0) = member(teacher, X0)
```

MODEL ***B***:

```
goodc(X0) = member(has_children, X0)
goodc(X0) = member(woman, X0)
goodc(X0) = and(member(married, X0), member(has_cellularphone, X0))
goodc(X0) = member(anthony, X0)
```

where ‘and’ is the ‘$\wedge$’ logic connective and ‘member’ is an incomplete function which returns true if the element is in the list. Both functions are defined in the background theory.

FLIP automatically chooses model A as the best solution by following its selection criterion wrt. the previous examples. In fact, it is to be expected that model B would be revised soon, because the last equation only covers one example and, in the best case, will only be applicable to clients whose name is ‘Anthony’. It is apparently a patch which should be avoided in a predictive software. Accordingly, FLIP rates the first model much more favourably than the second one.

4 Predictive Software Life-Cycle

In the introduction we have presented five main common stages in the development of scientific theories and software systems. The analogy is now exploited to re-design the software life-cycle with the objective of making it predictive.

4.1. The Predictive Life-Cycle

The following figure represents a mixture between an automated software construction cycle and scientific theory evolution. The terminology is used indistinctly, by either using a term from philosophy of science (or ML) or by using a term from software engineering.

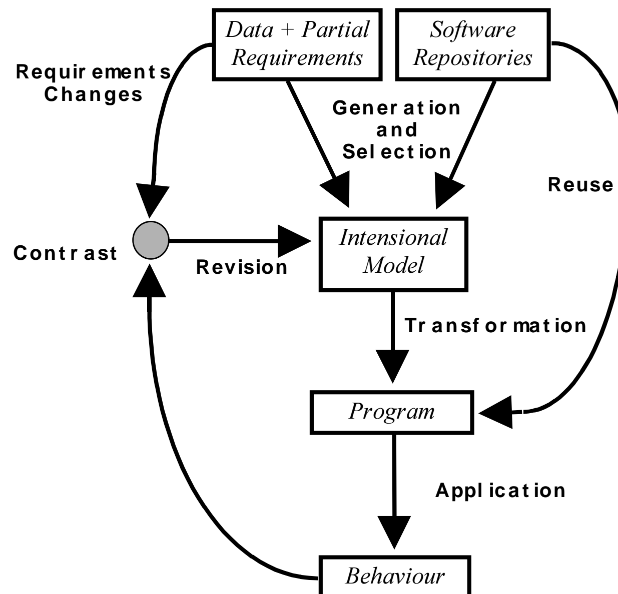

Figure 4.1. Predictive Software Life-Cycle

The first stage is specially appropriate for problems where the specification of requirements is difficult. Thus, in the example in Section 3.1 about the University library, if the goal is to avoid the manual classification of thousands of papers each month, then this requires *inventing* a ‘heuristic’ technique to solve the problem. However, there is no clear specification of the problem, and the most difficult stage of this example is precisely requirement elicitation. In this case, it seems better to learn the specification from the examples, possibly using the background knowledge or software repository. After a model is selected (model B in the example of Section 3.1), it can be transformed into a more efficient or appropriate form for operation. After this stage, a validation phase could be performed with another group of manually classified examples, something known in the ML literature as cross-validation. If the contrast detects anomalies or novelties, some revisions or extensions will have to be done, and these will have to be performed with the goal of maintaining the predictiveness of the model. Finally, the program is ready for application. The cycle could be reactivated by abnormal behaviour or requirements changes.

Obviously, this cycle could be more detailed depending on the automated or non-automated character of each stage. For instance, in a non-automated developing schema, an analysis stage could be introduced between the partial specification and the model, without using previous software. The design would convert this initial model into a refined model using the repositories.

The predictive life-cycle is similar to the transformational one (Balzer 1985) with regard to the later stages of the development: definition of a model or specification of requirements and its transformation into the final program. Also, the maintenance stage consists of a revision and re-derivation of the model. Nevertheless, the difference between both life-cycles lies in two aspects: the way in that intensional models or formal specifications (respectively) are built, and the criteria used for generating and selecting them. In the transformational approach, the prototype is usually generated by hand. The criteria taken into consideration for guiding this construction (at least in a first stage) are usually specifically developed for software engineering and are not inspired in other model selection criteria from other disciplines.

In other cases, the inductive stage and the transformation stage could be joined using mixed techniques as the inductive functional programmer ADATE (Olson 1995).

Finally, the adoption of this life-cycle also depends on the choice of the representational language. In the same way as many modern non-declarative methodologies were not adapted to

the automated programming paradigm, because automated deductive techniques required a well-established and manageable semantics, the first inductive stages are even more difficult to apply to non-declarative languages, because model-based languages have been shown to be more appropriate for automated induction of expressive and understandable models.

4.2. Predictive Declarative Programming

ML techniques can also be employed to generate the models and not only to evaluate them. However, it is difficult to adapt these techniques to languages which are procedural, because an inductive hypothesis or theory is usually declarative or model-based. On the other hand, any general programming paradigm should be based on full-expressive languages, thus excluding many declarative non-programming languages used in ML, such as attribute languages, grammars, etc. Another factor to be taken into account is comprehensibility.

This all restricts the possible languages which can make the predictive programming paradigm successful. Any full-expressive model-based language (such as functional languages, logic languages, functional-logic languages, etc.) that is, almost any declarative programming language, is theoretically appropriate for this paradigm. In practice, only logic programming has a well-developed literature and experience about the four important processes in the paradigm: generation and evaluation, transformation, reuse and especially revision, and more recently, functional logic programming (Hernández and Ramírez 1999).

Three facts make the step from ML to programming within declarative languages more plausible. First of all, the use of background knowledge and intensional data as evidence has been incorporated in many modern ILP systems. This allows for the use of partial specification + data, which reduces the learning complexity. The second fact is more insightful: ILP slowness is not so worrying for software engineering as for ML. In the generation of hypotheses from data, programming scale is not in milliseconds or seconds, but rather hours or even days. The third fact is comprehensibility; there are few inductive frameworks in ML apart from ILP where the hypotheses are expressible enough and easily comprehensible to humans.

Muggleton and Michie (Muggleton and Michie 1996) express this appropriateness quite well: "*ILP is relatively slow in generation of hypotheses, but allows a rich relational representation, has an explicit search bias (background knowledge) and [this bias and the resulting hypotheses are] comprehensible to human experts*".

The second process, transformation, is more necessary in our paradigm than in the classical automated programming paradigm, because the specification generated by ILP can be less prepared for execution than a specification which is generated manually. Program transformation usually requires a complete specification (or model) which is usually given as a logic, functional or functional logic program where the output is another program which must follow some structural, efficiency or space properties, while always maintaining the same semantics. Program transformation techniques (Alpuente et al. 1998, Pettorossi and Proietti 1990, Dershowitz and Reddy 1993, Pettorossi and Proietti 1996a, 1996b) are mainly based on folding and unfolding techniques, function orderings, mathematical induction[7] and partial evaluation (Jones et al. 1993). Although the techniques are mostly deductive, inductive techniques have sometimes been introduced for the invention of functions for folding and unfolding, something known as Eureka functions or predicates.

A third important process, *reuse*, has been extensively addressed in many declarative and non-declarative languages. In logic programming, modular extensions would be useful (Sannella and

[7] It is important not to confuse 'mathematical induction' with (scientific) induction, as used in this paper.

Wallen 1992), although the ILP framework usually incorporates medium-sized background knowledge without much difficulty.

Finally, the fourth process, revision, is where logic programming is clearly superior to other languages, because computational logic has been used in artificial intelligence. Non-monotonic extensions are very useful to the extension and modification of logic programs. Specifically, abductive logic programming (Kakas et al. 1993) could also be used for programming. Some other approaches to revision are based on minimal revisions (Richards and Mooney 1995, Wrobel 1996), which are contrary to the paradigm of predictive software. A more appropriate solution, which joins revision and reusability is dynamic logic programming. Dynamic Logic Programming (Alferes et al. 1998), which is more specialised to the revision process, allows for the update of an extended logic module *P* with another extended logic module *U*, solving the possible inconsistencies of the conjunction of two theories developed from different environment and purposes: "*dynamic program updates describe the evolution of a logic program which undergoes a sequence of modifications. This opens up the possibility of incremental design and evolution of logic programs, leading to the paradigm of dynamic logic programming. We believe that dynamic programming significantly facilitates modularization of logic programming, and, thus, modularization of non-monotonic reasoning as a whole. [...] dynamic programming provides the means of representing the evolution and maintenance of software specifications*".

As a result, we can chain all these techniques in a unique paradigm. Using logic programming, our cycle is practically the same as the generic one, but now using concrete (and in many cases automated) techniques, as is illustrated in Figure 4.2:

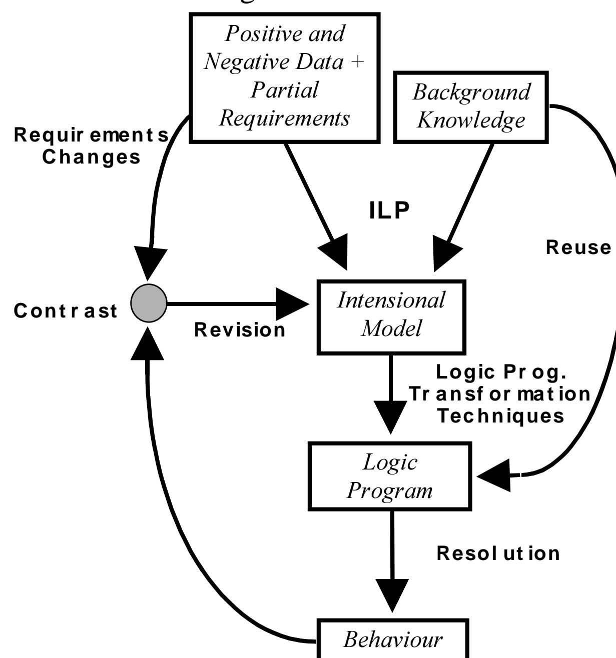

Figure 4.2. Predictive Logic Programming Cycle

The extension to other declarative languages seems difficult at the current time, except in the functional logic programming case, because generation and evaluation of inductive hypotheses and transformation processes are now available (Hernández and Ramírez 1999, Pettorossi and Proietti 1996a). In the end, the functional extension may be crucial for the acceptance of the predictive declarative programming paradigm, because the definition of functions is a profoundly established custom in software engineering. Furthermore, the examples which are more

appropriate for the use of ML techniques are classification problems, which are better handled as functions than as predicates.

5. On Automation of Other Stages

In Section 3 we have stated that evaluation methods can be implemented and easily automate the evaluation stage if the description language has model semantics, as in declarative languages. After the predictive life-cycle of the previous section, there are more stages to automate and they are more complex. Although the application stage is usually fully automated (and the goal of a software system was supposed to be precisely this, the automation of this stage), the other stages: transformation, generation, revision and reuse are much more difficult to automate fully for complex problems. Consequently, we will discuss two different approaches for automation: whole automation of small software cases and partial automation of more large-scale problems.

5.1. Whole Automation of Small Software Cases with the FLIP system

Let us consider an example which first appeared in (Cendrowska 1987). An optician requires a program to determine which kind of contact lenses should be used first on a new client/patient. The optician has many previous cases available where s/he has finally fitted the correct lenses (either soft or hard) to each client/patient or has just recommended glasses. The evidence is composed of 24 examples with the following attributes and possible values for them:

Attribute	Value
Spectacle Prescription:	{myopia, hypermetropia}
Astigmatism:	{no, yes}
Tear Production Rate:	{reduced, normal}
Age:	{young, presbyopic, prepresbyopic}

The goal is to construct a program that classifies a new patient into the following three classes {soft, hard, no}. Obviously, we can make this program by hand, by trying to discover some rules that could successfully be used in such a classification. However, depending on the size of the problem, we can also essay the automation of the generation stage. To do this we will use the FLIP system. After feeding FLIP with the 24 lens examples from (Cendrowska 1987), the result obtained in approximately six seconds on a personal computer (without the incremental option) is:

```
lens(X0,hypermetropia,no,normal) = soft
lens(young,myopia,no,normal) = soft
lens(X0,myopia,yes,normal) = hard
lens(young,hypermetropia,yes,normal) = hard
lens(prepresbyopic,myopia,no,normal) = soft
lens(prepresbyopic,hypermetropia,yes,normal) = no
lens(presbyopic,myopia,no,normal) = no
lens(X0,X1,X2,reduced) = no
lens(presbyopic,hypermetropia,yes,normal) = no
```

Now suppose that the optician wants to ‘internalise’ the program, at least for young people, who are the patients who most frequently request contact lenses. A transformation tool could automatically specialise the previous program into:

```
lens(young,X0,no,normal) = soft
lens(young,X0,X1,reduced) = no
lens(young,X0,yes,normal) = hard
```

Another option is to re-generate the program by only using examples of young people. The previous program can be generated by FLIP in just 0.19 secs.

Another stage that can be automated for this example is the revision stage. Consider a new kind of soft lens which is more permeable and can be beneficial to patients suffering from reduced tear production rate. Given the initial program with the old and the new examples, the FLIP system is able to revise or reuse the initial program when novelties and anomalies appear, constructing the following program:

```
lens(X0,hypermetropia,no,normal) = soft
lens(young,myopia,no,normal) = soft
lens(X0,myopia,yes,normal) = hard
lens(young,hypermetropia,yes,normal) = hard
lens(prepresbyopic,myopia,no,normal) = soft
lens(prepresbyopic,hypermetropia,yes,normal) = no
lens(presbyopic,myopia,no,normal) = no
lens(presbyopic,hypermetropia,yes,normal) = no
lens(young,X1,X2,reduced) = permeable
lens(presbyopic,X1,X2,reduced) = permeable
lens(prepresbyopic,X1,X2,reduced) = no
```

In this example we have seen that generation, transformation, revision and reuse can be fully automated. Moreover, the evaluation criterion embedded in the FLIP system ensures that the model is intensional, preserving predictiveness.

More information about the FLIP system and other examples that have been addressed can be found in `http://www.dsic.upv.es/~jorallo/flip/` or (Ferri et al. 2000).

5.2. Partial Automation and Declarative Programming

There is a huge amount of literature on (semi-)automatic program construction from the *specification* or the model of the problem (see e.g. (Nishida et al. 1991)), where the model is supposed to be known and stable. Theoretically, the subsequent stages of the whole process can be completely automated because the specification can be executed directly or after transformations. This used to be one of the main reasons to support declarative programming.

In our *predictive* paradigm, the problem of automating software construction is centred on the *intelligent* part of the process, modelling and modification, because it can be considered as a re-modelling. The previous example (and ILP in general) has shown that a certain amount of automation can be done, at least for small and very formal cases or where the specification is data-based or case-given (control systems, expert systems, medical systems, etc.). We have also seen that complex problems using declarative programming can automate the evaluation/selection stage, which is, as we have seen, fundamental to the cost of the entire software life-cycle.

Nevertheless, full automation of the entire process is neither scalable to complex problems nor problems for which continuous variables are relevant. For the time being, it is better to make a ‘lightweight’ use of logic and formal methods as used in conceptual modelling (Robertson and Agustí 1998). Other techniques, such as case based reasoning or temporal reasoning, could be incorporated. It is important to note that logics can be used to “model problems” or to “specify solutions of problems”, whenever the techniques are mainly inductive or deductive.

A hybrid or semi-automated approach is possible by the intrinsic comprehensibility of logic. Both hypotheses and background knowledge can be partially hand-written by humans to guide the search (something which is very difficult to do in other ML paradigms, like neural networks), and the results are comprehensible to allow manual modifications or improvements. Hence the

understandability of declarative programming is what mostly supports its use, rather than the automation of the evaluation or transformation stages.

As we have commented on in the introduction, the case of inductive programming has been recently highlighted by Partridge (Partridge 1997) and will become of even more importance in the future. However, at the present time, a full and general software engineering life cycle can only incorporate automated inductive programming provided that the whole cycle is automated and infallible, or provided that the representational language input and output of some of the processes that can be automated (evaluation, transformation) is understandable by humans. Since the first situation (full automation) is a Utopia in the short-term and mid-term for complex problems, we foresee a forthcoming increase of the use of declarative languages.

Other issues which have been introduced in this paper can also be illustrated with the example of section 3.1. The choice of a declarative paradigm allows a manual modification of the program to achieve better performance. It also allows for the inclusion of extensions (such as the consideration of the words of the title), which, after some manual intervention, could be re-generated again.

6. From ML to Other Kinds of Software Systems

Some of the ML paradigms and techniques that were seen in section 2 have not been applied or included in the new life-cycle because they are not compatible with the predictive software paradigm or with the view of software as an incremental learning session from examples. This is a result of a finally accepted principle in ML: there is no best paradigm for all kinds of problems. The same thing happens in software engineering. In this section, we emphasise two different ML paradigms that can be fruitfully adapted for other kinds of software systems.

6.1. Interactive Software and Query Learning

Hitherto the paper has implicitly adopted the common assumption of software engineering that the software developer is different from the software user. In this case, the lack of functionality (prediction errors) is mainly perceived by the user and reported to the developer, as shown in Figure 6.1.

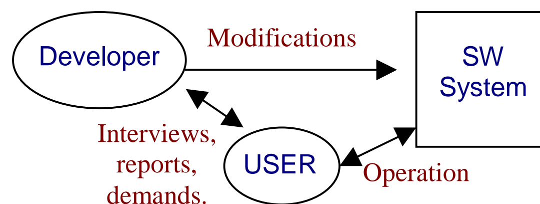

Figure 6.1. Peer Interaction in Classical Systems

On the other hand, in a (semi-)automated software framework, where part of the development is performed by an inductive/learning system that could understand user demands, the scene changes as follows:

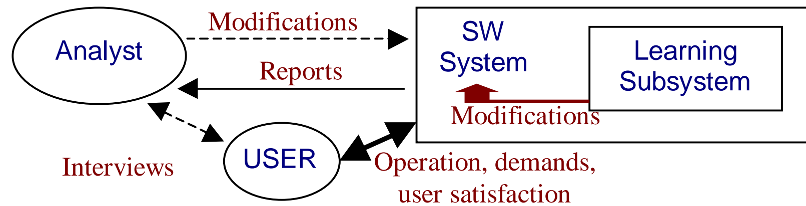

Figure 6.2. Peer Interaction in Interactive Systems

Specifications are remade and updated as the software system interacts with the user, learning what s/he wants, as well as what s/he will want, following a predictive view. It has often been advocated (Rumbaugh 1994) that learning from interaction with the user is the best way of capturing requirements, in perfect parallel with ML, where it has been shown that learning can be accelerated if the learner can interact with the environment or a teacher.

This superiority has recently been vindicated as a 'new' idea by (Wegner 1997), arguing that "interactive machines" are qualitatively more powerful than passive Turing machines (the input and program are fixed before the program starts). In our opinion, the *new* proposition is just a new expression of well-known facts, which are somehow equivalent: first, Turing machines with an oracle are more powerful than Turing's machines (as shown by Turing himself) and secondly, a learner with the ability of interacting with the environment, via experiments of its hypotheses, actions and responses, or queries to a teacher (Angluin 1987, 1988), is more powerful than a passive learner who only receives a sequential input from the environment.

For this interaction to take place, it is necessary for the communication to be established from peer to peer. The interaction with a heterogeneous environment and especially with human beings is a very difficult issue. Agent-oriented software (Shoham 1993) has also been seen as an evolution of object-oriented software where agents are not passive objects but proactive and reactive, and they have an independent decision control to messages (demands). This approach, however, does not solve the problem of interaction with the user.

As a first approach, information about the user or environment satisfaction, i.e., system behaviour, must be improved. Reinforcement Learning (RL) (Sutton 1991, Kaelbling et al. 1996, Hernández 2000) provides a very useful paradigm for this. In RL, it is usually assumed that the learner receives some reward (or penalty) value for its actions. This can be applied to software systems in many different ways, from the simplest one of introducing a reward button in the applications to knowing dynamically the user's level of satisfaction with the system behaviour. For the example of section 3.1, the user (any person who makes the queries) can see whether a paper is misclassified. For this situation, the application could provide an interface to allow authorised users to correct (or simply penalise) the misclassification. In this case, a revision of the model can be triggered by the user, ameliorating the interaction between the user and the application. This would improve its behaviour by positive and negative reinforcement.

On the other hand, the communication of user demands must also be ameliorated by increasing the external intelligibility of software. Human beings should be able to tell the system to do something that it is not included in the menu options or inside the commands of a strict interpreter. Although some of these other simple demands can be incorporated inside what has been dubbed "intelligent software agents" (or program by demonstration, Lieberman 2000), for more complex needs, there must be a learner that *understands* these needs. Moreover, the user must also *understand* the agent, in order to be sure that it has understood what s/he wanted.

To reach this communication between human beings and software systems, Muggleton and Michie advocate the logic programming paradigm by using ILP techniques: "*software which interacts with human beings [...] should incorporate declarative machine learning at its core.*

[...] Using today's machine learning technology, learning at the declarative level would necessarily be carried out using Inductive Logic Programming" (Muggleton and Michie 1996). More precisely, they suggest a two-layer approach for human-computer interaction: the upper *declarative* layer should be capable of extensive human interrogation at runtime, supporting simple deductive and inductive inference, which translates the user's *demands* into *precise functions* to the lower *procedural* layer. Finally the result is given to the user after pre-processing by this *declarative* layer.

In this case, the relative slowness of ILP is a drawback, because the response time scale is in milliseconds or seconds. Nonetheless, ILP possibilities are currently at this borderline, and ILP agents are beginning to appear. For instance, ILP web agents (Lavrac 1998) must advertise their *services*, must negotiate, self-sell, deliver, install, update, learn from users needs, desires and acquisitive capacity.

6.2. Knowledge-Based Systems and Lazy Learning

As we said in section 2, eager techniques are necessary for the construction of a predictive model in the large. However, some kinds of problems are difficult to convert into a model, especially when new cases are quite varied, highly knowledge-dependent or difficult to predict (Scacchi 1991, Cuena 1993, Guida and Tasso 1994). The knowledge-based approach for software has been vindicated to address this kind of complex problems.

Traditionally, the knowledge of expert systems or knowledge based systems (KBS) was manually selected from a domain, pre-processed and stored in an appropriate representation, and the KBS was able to use several reasoning techniques to give solutions to different *unpredictable* problems. Although initially, the inference engine was almost exclusively constituted by automated deduction techniques, more reasoning methods have been increasingly introduced: case-based reasoning, analogical reasoning and abduction. This has increased the percentage of factual or extensional data (previous cases), which makes modern KB systems clear examples of lazy systems.

Recently, however, expert systems (and KBS) are beginning to be less "idiots savants" by including some inductive techniques, especially for the stage of knowledge acquisition. In this case, eager inductive techniques can be used, which gives a life-cycle which is more compliant with the life-cycle in Figure 4.1 than with traditional software cycles.

In the end, the combination of eager and lazy techniques must be seen as a failure of eager techniques to find a good model for some kinds of problems (or some parts of the problem), but this must not preclude a predictive view of what actually can be modelled or revised.

6.3. Other Transfers

In general, any technique of ML (see e.g. Mitchell 1997) should have its parallel in software engineering. For instance, the oldest transfer took place between Evolutionary Software and Genetic Programming. Nowadays, they have the close relationship that the other transfers should have, and they even have a common and agglutinative name, Evolutionary Computation (Goldberg 1994). We think that similar names should be given to eager computation or lazy computation.

Evolutionary Software matches perfectly with incremental learning, and different software paradigms have been employed under this scheme: object-oriented (Cox 1987), functional (Olson

1995), functional and logic (Hernández and Ramírez 1998, 1999). Some of them complement the usual techniques and operators from ILP with new techniques inspired in genetic programming.

We also consider that the dynamic part of modelling could be addressed by using some theories of change and time from AI, such as situation calculus or event calculus, when proper induction frameworks for these representations are developed further.

7. Conclusions

This paper has introduced a new paradigm for software development inspired by the analogy between a software development process and an incremental learning session. Predictive paradigms and techniques of ML have been adapted to introduce the notion of 'predictive software' in order to minimise software modification probability. Intensionality and consilience have been vindicated as the most appropriate evaluation criteria for software modelling. Moreover, this evaluation can be fully automated. Although further experimental validation of more complex software systems with different selection criteria should be done, the existing theoretical and experimental results of these criteria in ML cannot be neglected and can be reasonably reused rather than re-obtained for upholding the 'predictive software' claim.

A new life-cycle has been introduced, clarifying the location of different inductive and deductive stages, thus making it possible for a (partial) automation of the life-cycle. Both inductive and deductive declarative programming techniques will play a central role in this automation. This life-cycle is more suitable for classification problems. Much still has to be done to fully automate the life-cycle that has been presented, particularly for more general and complex applications. Nonetheless, the vindication for intensional models in software engineering is useful for reducing the number of future modifications, as has been shown in ML. This predictive character of software is useful independently of the level of automation of the process.

The forthcoming decades will surely bring a broader range of applications of automated induction in software engineering. We think that this work has helped to clarify the generation framework and selection criteria for inductive programming.

8. References

Abe, N. 1997. Towards Realistic Theories of Learning. *New Generation Computing*, 15: 3-25.

Aha D.W. 1997. Lazy Learning. Editorial. Special Issue about "Lazy Learning" of Artificial Intelligence Review, Vol. 11, Nos. 1-5.

Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., and Przymusinski, T.C. 1998. Dynamic Logic Programming, in Freire et al. (eds) Proc. Joint Conf. of Declarative Prog., pp. 393-408.

Angluin, D. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75: 87-106.

Angluin, D. 1988. Queries and concept learning. *Machine Learning* 2 (4): 319-342.

Balzer, R. 1985, A 15 Year Perspective on Automatic Programming, *IEEE Transactions on Software Engineering*, 11(11): 1257-1268.

Barker, S.F. 1957. *Induction and Hypothesis*. Ithaca.

Barron, A., Rissanen, J. and Yu, B. 1998. The Minimum Description Length Principle in Coding and Modeling. *IEEE Transactions on Information Theory*, Vol. 44, No. 6, 2743-2760.

Bergadano, F. and Gunetti, D. 1995. *Inductive Logic Programming*, The MIT Press.

Berry, D.M. and Lawrence, B. 1998. Requirements Engineering. *IEEE Software*, pp. 26-29.

Boehm, B.W. 1981. *Software Engineering Economics*. Prentice Hall.

van den Bosch, A. 1994. *Simplicity and Prediction*. Master Thesis, department of Science, Logic & Epistemology of the Faculty of Philosophy at the University of Groningen.

Bratko, I. and Dzeroski, S. 1995. Engineering Applications of ILP. *New Generation Computing*, 13: 313-333.
Cendrowska, J. 1987, PRIS: An Algorithm for Inducing Modular Rules, *International Journal of Man-Machines Studies*, 27:349-370.
Cox, Brad 1987. *Object Oriented Programming, An Evolutionary Approach*. Addison Wesley 1987.
Cuena, J. (ed.) 1993. *Knowledge-Oriented Software Design*. Elsevier.
Dershowitz, N. and Reddy, U.S., 1993, "Deductive and Inductive Synthesis of Equational Programs", Journal of Symbolic Computation, 15, 467-494.
Ernis, R. 1968. Enumerative Induction and Best Explanation, *The Journal of Philosophy*, LXV (18), 523-529.
Ferri-Ramírez, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J., 2000, FLIP: User's Manual, Dpto. de Sistemas Informáticos y Computación, Valencia, TR: (II-DSIC-24/00).
Flach, P.A. and Kakas, A.C. (editors), 2000. *Abduction and Induction: Essays on their relation and integration.* Kluwer Academic Publishers.
Flener, P. and Yilmaz, S. 1999. Inductive synthesis of recursive logic programs: achievements and prospects. *The Journal of Logic Programming*, 41, 141-195.
Gold, E. M. 1967. Language Identification in the Limit. *Inform and Control*, 10: 447-474.
Goldberg, D.E. 1994. Genetic and evolutionary algorithms come of age. *Comm. of the ACM*, 37(3): 113-119.
Guida, G. and Tasso, C. 1994. Design and Development of Knowledge-Based Systems, Wiley & Sons.
Harman, G. 1965. The inference to the best explanation. *Philosophical Review*, 74, 88-95.
Hempel, C.G. 1965. *Aspects of Scientific Explanation*. The Free Press, New York, N.Y.
Hernández-Orallo, J. 2000. Constructive Reinforcement Learning, *International Journal of Intelligent Systems*, 15(3): 241-264.
Hernández-Orallo, J. and Ramírez-Quintana, M.J. 1998. Induction of Functional Logic Programs, in Lloyd (Ed) *JICSLP'98 CompulogNet Area Meeting on Computational Logic and Machine Learning*, 49-55.
Hernández-Orallo, J. and Ramírez-Quintana, M.J. 1999., A Strong Complete Schema for Inductive Functional Logic Programming, in Flach, P.; Dzeroski, S. (eds.) *Inductive Logic Programming'99* (ILP'99), in V. 1634 of the LNAI series, pp. 116-127, Springer-Verlag.
Hernández-Orallo, J. and Ramírez-Quintana, M.J. 2000. Software As Learning. Quality Factors and Life-Cycle Revised, *Foundational Approaches to Software Engineering*, Berlin (FASE'2000 / ETAPS'2000) appeared in Tom Maibaum (ed.) "Fundamental Approaches to Software Engineering" Volume 1783, pp. 147-162, of the Lecture Notes in Computer Science (LNCS) series, Springer-Verlag.
Humphrey, W.S. 1990. *Managing the Software Process*. Addison-Wesley, London.
Jones, N.D., Gomard, C.K. and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.
Kaelbling, L., Littman, M. and Moore, A. 1996. Reinforcement Learning: A survey. *Journal of Artificial Intelligence Research*, 4: 237-285.
Kakas, A.C.; Kowalski, R.A.; Toni, F. 1993, Abductive Logic Programming, *J. of Logic and Computation* 2(6).
Katsuno, H. and Mendelzon, A. 1991. On the difference between updating a knowledge base and revising it, in James Allen, Richard Fikes and Erik Sandewall (eds) Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference (KR91), pages 230-237.
Kuhn, T.S. 1970. *The Structure of Scientific Revolutions*. University of Chigago.
Kuvaja, P. 1994. *Software Process Assessment and Improvement: The Bootstrap Approach*. Blackwell.
Lavrac, N. 1998. ILP. The next years, in Lloyd (ed.) JICSLP'98 CompulogNet Meeting on Computational Logic and Machine Learning.
Lavrac, N. and Dzeroski, S. 1994. Inductive Logic Programming: Techniques and Applications. Ellis Horwood.
Li, M. and Vitányi, P. 1997*. An Introduction to Kolmogorov Complexity and its Applications*. 2nd Ed. Springer.
Lieberherr, K. J. 1996, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston.
Lieberman, H. (guest editor), 2000. Programming by Example, Special Issue, Comm. of the ACM, 43, (3): 73-114.
López de Mántaras, R. and Armengol, E. 1998. Machine Learning from examples: Inductive and Lazy Methods. *Data & Knowledge Engineering* 25, 99-123.
Mitchell, T. M., 1997. *Machine Learning*, McGraw-Hill International Editions.
Mooney, R.J. 1997. Integrating Abduction and Induction in Machine Learning. in Peter Flach and Antonis Kakas (eds), Proceedings of the IJCAI'97 Workshop on Abduction and Induction in AI.
Muggleton, S. 1991, "Inductive Logic Programming" *New Generation Computing, 8, 4*, pp. 295-318.
Muggleton, S. and De Raedt L. 1994. Inductive Logic Programming — theory and methods. *Journal of Logic Programming*, 19-20:629-679.

Muggleton, S. and Michie, D. 1996. “Machine intelligibiity and the duality principle”. *British Telecom Technology Journal* http://www.cs.york.ac.uk/mlg/.

Muggleton S. and Page. C.D., 1999, “A learnability model for universal representations”. Submitted to JACM. http://www.cs.york.ac.uk/mlg/.

Natarajan, B. K., 1991, “Machine learning : a theoretical approach”, Morgan Kaufmann.

Nishida, F., Takamatsu, S., Fujita, Y., and Tani, T. 1991. Semi-automatic program construction from specifications using library modules. in *IEEE Trans. on Software Eng*, 17, (9): 853-871.

Olson, R., 1995. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74 (1).

Partridge, D. 1997. The Case for Inductive Programming. *IEEE* Computer, pp. 36-41.

Pettorossi, A. and Proietti, M., 1996a. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys,* Vol. 28, no. 2.

Pettorossi, A. and Proietti, M., 1996b. Developing Correct and Efficient Logic Programs by Transformation. *Knowledge Engineering Review*, Vol. 11, No. 4.

Popper, K.R., 1968. *Conjectures and Refutations: The Growth of Scientific Knowledge* Basic Books.

Pressman, R. S. 1997, *Software engineering : A practitioner's approach.* New York, McGraw-Hill.

Proietti, M. and Pettorossi, A. 1990. Synthesis of eureka predicates for developing logic programs. in *Proceedings of ESOP '90* (Copenhagen), *Lecture Notes in Computer Science 432*, Springer Verlag, 306-325.

Rich, C. and Shrobe, H. 1978. Initial Report on a Lisp Programmer’s Apprentice. *IEEE Transactions on Software Engineering*, SE-4(6): 41-49.

Richards, B.L. and Mooney, R.J. 1995. Automated Refinement of First-Order Horn-Clause Domain Theories. *Machine Learning*, 19(2), 95-131.

Rissanen, J. 1978. Modelling by the shortest data description. *Automatica-J.IFAC*, 14:465-471.

Robertson, D. and Agustí, J. 1998. *Software Blueprints. Lightweight Uses of Logic in Conceptual Modelling*, Addison Wesley / ACM Press.

Rumbaugh, J. 1994. Getting Started: Using Use Cases to Capture Requirements. *J. Object-Oriented Programming*.

Sannella, D.T. and Wallen, L.A., 1992. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12:147-177.

Scacchi, W. 1991. Understanding software productivity: Toward a knowledge-based approach. *Int. J. Software Eng. and Knowledge Eng.*, vol. 1: 293-320.

Shoham, Y. 1993. Agent-Oriented Programming. *Artificial Intelligence* 60 (1), 51-92.

Shrager, J. and Langley, P. 1990. *Computational Models of Scientific Discovery and Theory Formation* Morgan Kaufmman.

Solomonoff, R.J. 1964. A formal theory of inductive inference. *Inf. Control*. 7(1-22), 224-254.

Sutton, R.S. 1991. Special issue on reinforcement learning. *Machine Learning*.

Tessem, B., Bjørnestad, S., Tornes, K. M. and Steine-Eriksen, G., 1994. *ROSA = Reuse of Object-oriented specifications through Analogy: A Project Framework*. Report no. 16.

Thagard, P. 1989. Explanatory coherence, *The Behavioural and Brain Sciences*, 12 (3), 435-502.

Valiant, L. 1984. A theory of the learnable. *Communication of the ACM* 27(11): 1134-1142.

Wegner, P. 1997. *Interactive Foundations of Computer*. Brown University, April 1997.

Whewell W. 1847. *The Philosophy of the Inductive Sciences*. New York: Johnson Reprint Corp.

Wrobel, S. 1996, First Order Theory Revision, in L. De Raedt, editor, *Advances in Inductive Logic Programming*, IOS Press, 14-33..