Resistance is Futile;
Formal Linguistic Observations on Design Patterns

Peter van Emde Boas

ILLC, FWINS, Universiteit van Amsterdam,
Plantage Muidergracht 24, 1018 TV Amsterdam
February 19, 1997

Abstract

Inspection of the current literature on Design Patterns shows that the Prime Directive for this community is Pragmatics. It hardly matters what patterns are, or how Patterns are represented formally or syntactically. What does matter is their role in enhancing the reuse of good solutions to recurring problems.

In this article I want to show that minimal assumptions about the pragmatic use of Patterns suffice to show that Design Patterns form just another formal language, which can be shown to be at least Recursively enumerable. Whether the language is Recursive depends on further conditions on the actual relation which is assumed to hold between a pattern and its possible invocations. There are no a priori reasons enforcing that this should be an easily decidable relation; quite to the contrary: a little amount of linguistic expressivity suffices for showing that this relation is likely to be complex. Without restrictions on the linguistic tools allowed for expressing design patterns this relation will become undecidable. For Non-Design Patterns which leave no visible trace in the designs constructed using them, the argument doesn't apply.

There is no content to patterns—no one will ever produce a journal article or conference paper whose content is a pattern or pattern language—and if someone does, he's missed the point.

Richard Gabriel in the introduction to [29]

1 Backgrounds

During the last few years Design Patterns have become popular as a tool for registering and sharing experience about good design, analysis and construction of reusable software. As is the case with other recent developments in Information Technology the driving forces reside in Industry rather than Academia, and so far the pattern movement has drawn only minor attention from the more fundamentally oriented Computer Science community. Moreover, the notion of Patterns and Pattern languages currently is invoked in areas quite different from Software Design, like System production, the management of the process leading to this production or just Management in general. Evidently this diversification makes it even harder to figure out what Patterns are intended to be.

Inspection of the recent literature on this topic shows that there exist in fact various motivations and purposes which interact together as possible driving forces behind the pattern movement.

1.1 Patterns, Experience and Literate Programming

The first motivation is shared with the original Pattern concept as put forwards by Christopher Alexander in Architecture. Patterns describe those elements in our constructed environment which make this environment *livable*; the elusive notion of *the quality without a name* [2, 3, 10].

In our world this purpose is the one which is quite well described by the slogan *Patterns describe good solutions to frequently recurring problems.* These problems may relate the application itself, the code or the design of the system, or just the process leading to this design.

For a mathematician and theoretical computer scientist like myself, the esthetic ideas and notions are far beyond my level of competence, and I will abstain from contributing to the debate on these issues. This aspects becomes meaningless outside the scope of the *human factor* which in general computer science (as is the case with other technical sciences like mathematics, physics, chemistry and biology) is eliminated from the models, abstracted away or just disregard.

There does exist a relation with the topic called *Literate Programming*, where esthetic ideas are key as well. Literate programming by no means is confined to the realm of Object-Orient programming or design. There exist some well known books and papers on this topic in the traditional field of imperative programming [5, 6, 20]. There even exists a research journal *Science of Computer Programming* whose scope substantially overlaps with the topic of literate programming.

In the classic procedural context literate programming has been done using tools, concepts and notions inspired on existing and used languages. Themes are the selection of the "right" primitives like sequential composition, conditioning and iteration, the use of nice and appropriate data structures, and the invention of intuitively convincing correctness arguments. The conceptual gap between the language used for expressing the esthetics of programming and the languages actually used for writing programs is not very wide. Literate programming is performed using idealized simplifications of the existing sequential languages.

1.2 Patterns and Pidgins

The idealized languages used for literate programming serve also another purpose: they can be used as a framework for analysis of correctness, and performance. Using them one can express the various possible approaches and the trade-offs involved in choosing between them. In the world of Object Orientation people want to perform such analysis as well, not being subjected to the constraints enforced by actual OO languages. Design Patterns are claimed to be useful for this purpose.

As stated by Gamma et al. [11] *Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance", "Encapsulation" and "Polymorphism".*

I read these sentences and the surrounding text in Chapter 1.1 of their book as a call for an idealized simplified Object-Oriented language which allows us to escape from the murky details enforced by the use of existing accepted languages. In the procedural world this need was experienced during the early 1970-ies when the field of Analysis of Algorithms came to development. When Knuth started his series of books *The Art of Computer Programming* [19] in 1968, he represented the algorithms he was analyzing in the assembly code of an idealized machine called MIX. Six years later, in the standard textbook [1] by Aho, Hopcroft and Ullman algorithms are represented at a higher level using the "skeleton" language known by the name of *Pidgin Algol*. Experience shows that this pseudo higher level language suffices for analyzing algorithms with respect to their time and space behavior, and that is the main theme of their book.

In the world of object-orientation a generally accepted version of a pidgin language so far seems not to exist. I see several possible causes for this state of affairs. The diversity between existing OO languages seems large compared to the diversity between procedural languages 30 years ago. There are choices to be made: you can't at the same time allow multiple inheritance and prohibit it. Similarly it seems difficult to allow procedural elements in a pidgin OO language for convenience and at the same time claim that this Pidgin is a pure Object Oriented language. Another possible cause for the absence of an OO Pidgin language is the fact that the use of graphical and visual representations for Object Oriented programs and systems is far more dominant than the corresponding visual analogue of flowcharts in the procedural world. In an OO Pidgin language one would like to include generally used graphic notations like class, object, state transition and interaction diagrams. Design Patterns allow you to do so, free from the constraints and limitations of actual OO languages and therefore they start to perform the role of a pidgin OO language. If this indeed is what the Pattern Community wants, it becomes a worthwhile investigation to discover the "right" ingredients for such a pidgin language.

1.3 Design Patterns for OO Software

The third purpose of patterns is the mere representation of valuable experience in design of Object Oriented systems themselves. What are the ingredients of good OO software design which enhance flexibility and reuse. This actually is the dominant motivation encountered when reading the prefaces of [11].

The description of this experience uses a format combining diagrams, text in natural language, and code fragments, which, if consumed by a well intended reader, suffices for carrying across the intended idea. Reading the examples with the sharp eye of a critical formalistic mathematician, it is not hard to locate trouble spots and mistakes, including deviations against the few syntactic rules the authors seem to have agreed to observe, but such minor mistakes should not be taken too seriously.

Compared to the representation of the more managerial patterns as found for example in [29] the representation style of Gamma et al. looks very formal indeed. Their book however remains imprecise and vague on the very basic issues a standard computer scientist, mathematician or logician would ask when faced with some structure which could call itself a *language of Patterns*.

1. What is the syntax of the language described?

2. What is the intended semantics for the legal expressions in the language?
3. assuming that the language is part of a more elaborate calculus for synthesizing OO software, containing rules of inference, proofs, etc., what is the structure of this calculus, and does it have important properties like soundness and completeness?
4. What is the intended realm of applicability of the language?

Evidently, the prime directive of the Pattern movement is pragmatics, thus answering the last question: we want to present proven solutions to recurring design problems. At the same time the mere asking of the first three questions seems to go against the flow of the Pattern Community, particularly for adherents whose prime motivation resides with the human factor aspects.

In fact my paper was provoked by the claim by Richard Gabriel cited above: *There is no content to patterns—no one will ever produce a journal article or conference paper whose content is a pattern or pattern language—and if someone does, he's missed the point.* [29] I read this statement to be interpreted as a denial that the concept of pattern or pattern language *on behalf of its intrinsic nature* can be studied as a formal object. It shouldn't be touched from the perspective of formal linguistics, logic or theoretical computer science in general.

The production of patterns deviates substantially from the traditional way (linguistic) objects are created in Computer Technology. As Gabriel continues [29]: *How do you produce generative patterns? You don't—you find them. They are in systems written by virtuosos, they are in systems that are a dream to maintain. If a great programmer comes to you and says, "Hey, look at this code I found," that's where you'll be able to mine generative patterns.* No wonder that Patterns then are produced during so-called *writers' workshops*, where the author hears how the audience discusses the paper and debate its strengths and weaknesses (participation in the discussion by the author being disallowed); see the introductions to [29].

1.4 Design vs. Non-Design Patterns

At this point I think that it is necessary to distinguish between the Design Patterns as presented in [11] and the more general, managerial Patterns, which I will call *Non-Design Patterns* in the sequel. Design Patterns represent basic ideas about the structure of OO code, and leave a visible trace in the software itself (or in its design and/or specification). Non-Design Patterns are used in the process leading to the construction of software, but once the product is finished, it doesn't contain the pattern itself. For example, if a system has been designed using the **Bridge** pattern from [11] we may hope to recognize this to be the case by inspecting the class diagram of the system. However, if the team designing the system has invoked the **Threshold Switch** pattern from [4] there is no way to read this from the code or specification (at best it is told somewhere in the documentation of the system).

It is curious to see that, notwithstanding the fact that the editors of [29] seem to be aware of the difference between Design and Non-Design Patterns (cf. the introduction to Part 2), they fail to observe this key aspect of the difference: does the pattern reside in the system produced or only in its production.

The purpose of this paper is to point out that at least for the case of the Design Patterns as described by Gamma et al., the claim that a possible formal study of Patterns is meaningless or even impossible is wrong. As I will point out basic Computation Theory conveys, on this issue to the Pattern Community the very same sinister message that has been transmitted to the Federation by the Borg collective on several occasions [8]: *... We will add your technological distinctiveness to ours. Resistance is Futile. You will be assimilated.*

I will show that minimal assumptions which are entailed by the pragmatic idea that a Design Pattern resides in a program (or the design or specification leading to this program) and that its presence must be communicable, suffice for proving that Design Patterns populate a formal language which, like any other effectively generable language in Computer Science, will be Recursively Enumerable.

The more interesting question is whether the language is Recursive and, if so, whether it will be easily decidable (as far as its Computational Complexity is concerned). I will show that the question of decidability is equivalent to the existence of bounds on the size of occurrences of a pattern in terms of the size of the pattern itself. The functions involved in estimating these bounds are relevant for measuring the algorithmic complexity of deciding whether something is a Design Pattern or not.

It is at this point that the Pattern Community may invoke all the freedom they want, but this freedom goes with a price. Even with a small amount of linguistic expressive power it is not hard to find examples of patterns which can occur is a designed system but are hard to detect; standard NP-complete problems easily are encoded in this format. Without constraints on the linguistic tools used for describing patterns life indeed becomes misreable: whether something is a Design Pattern easily will become undecidable.

The above observations hold for Design Patterns only. Non-Design Patterns by their nature violate the basic assumptions on which my argument is based, since they leave no visible trace in the developed system. At best they will be mentioned in the documentation of the system. Such documentation being written in *Natural Language* presumably escapes from my arguments. I will therefore disregard Non-Design Patterns in the sequel.

1.5 Structure of the paper

The remainder of this paper is structured as follows: in Section 2 I present an intuitive survey of standard Computation Theory, Formal Language Theory, and the notions required for understanding my analysis. Readers familiar with these subjects should skip this section.

In Section 3, I will state and justify the minimal pragmatic assumptions about Design Patterns and their occurrences within programs, and show how they lead to the conclusion that Design Patterns form a recursively enumerable language: hence a Syntax for Design Patterns may be inferred from their use in practice.

Section 4 deals with the issues of Decidability and Complexity. If the property of being a Design Pattern is decidable, there must exist a fixed upper bound for a shortest certificate witnessing the use of this Pattern. The converse implication that such a bound entails decidability is evident.

Finally in Section 5 I will illustrate how easy it is to encode standard *NP*-complete problems by questions of the form: does a particular system contain this particular Design

Pattern or not. I will also show that the lack of linguistic constraints on how te specify this relation between Design Patterns and programs suffices for making this relation arbitrarily hard to decide, up to the level of being undecidable.

Section 6 contains a list of possible outcomes for the decidability status of the various relations introduced in Section 3. My results are subject to hypotheses, and whether these hypotheses will come out to be true or not is up to the choice of the Pattern Community, in particular the users of design Patterns. These possible worlds may indicate to the Pattern Community the consequences of their decisions.

2 Theory of Computation

2.1 Machines and Computations

Theory of computation finds its origin in a question which arose in mathematics and logic during the first half of this century. *What does it mean for something to be effectively computable.* This something may be specialized to evaluating some function at some argument or deciding some property for some object.

Objects and arguments in this theory are *finitistic* (finitely describable) entities. In general a treatment of computation theory will select a fixed domain of finitistic objects like *numbers (non-negative integers), words (strings)* over a finite *alphabet*, or *finite labeled graphs*. The example of graphs shows that it makes sense to consider finitely describable rather than just finite objects: a graph when considered a combinatorial structure is a finite object; when considered a geometric object (i.e., a drawing in the plane) it becomes infinite.

Computation theory learns that the selection of a domain can be done without loss of generality, due to the fact that other classes of finitistic objects can be *encoded* by the chosen ones, using codings which are evidently easy to use and/or decode. A natural example of such an encoding is given by the decimal number system which is one of the foundations of our culture. Our technology is based on a similar encoding by binary numbers. Whenever information of any kind is communicated over the internet it first has to be encoded in bytes and bits; consequently all information transmitted over the net is included in the collection of objects which are representable by binary strings.

Based on such encodings, the theory of computation becomes independent on the choice of the domain used for its formalization. It doesn't matter whether one encodes data by means of numbers, strings or graphs; the same class of functions and/or predicates will turn out to be computable.

Two properties of the domain are important for the sequel. In the first place the domain always constitutes a *Countable Universe*; one can enumerate the elements in the domain using nonnegative integers. In the second place the domain caries a (natural) linear discrete order. For numbers we have the standard order, for strings we have the lexicographical order combined with the order by length: the shorter string always precedes the longer one and two strings of equal length are compared using the lexicographical order (based on some fixed order of the finite alphabet). Only in the case of finite graphs there is no natural order (but one always can borrow an order from a different domain using some encoding).

Finitistic objects play several roles in Computation Theory. They can be acting as a piece of *data* (to be read, processed or being printed). Next they can play the role of the *algorithm* or *program*. Finally they can be used as representation of a *machine configuration*.

A *computation* by a program p is a sequence of configurations $\ldots, c_i, c_{i+1}, \ldots, c_j, \ldots$, such that all configurations c_{i+1} in the sequence are obtained from their predecessor c_i by performing a computation step allowed by the program p; in formula $c_i \overset{p}{\to} c_{i+1}$. The machine step is both local and finite; in terms of the (unspecified) details of the encodings of the configuration c_{i+1} is obtained from c_i by changing a small localized part or a small set of symbols. Moreover the set of configurations c_{i+1} which may follow c_i is a finite list which can be computed by very simple means.

In our description of computations the successor configuration is not uniquely determined. This is indeed what we want. Our model describes the concept of *nondeterministic* computation. When there exists for every configuration at most a single successor configuration we call the program *deterministic*.

The above notion of a computation leaves many details unspecified. A more precise formalization determines the specific machine model on which your computation theory is based. It is nowadays a tradition to use the *Turing Machine* model [27]. Here the program p is a finite list of quadruples, and the configuration c is a finite string over some alphabet. A legal transition is obtained by substitution of a substring consisting of no more than three adjacent symbols by another substring according to rules specified in the program.

Another favorite model is the *Random Access Machine* which resembles a true though rather primitive von Neumann computer, be it without a limitation of word length or address space. The *Storage Modification Machine* operates on finite labeled directed graphs and operations involve creation of new nodes and redirection of edges. Various *term Rewriting Systems* and *Logical Calculi* which are used by mathematicians and/or logicians for proving theorems or evaluating complex expressions, also can be seen to be specific instances of the abstract framework.

For further details and a discussion of the relative merits of the various models I refer to the literature [9, 14, 18, 22, 26, 28]. Inspection of the literature will also eliminate the implicit circularity in the definitions as given in this paper: when referring to structures in the formalization being "easy" or "local" I am invoking (be it at the meta-level) the very concept of efficient computability I am trying to describe. Choosing a specific model makes it possible to write down all the details and then this circularity evaporates (but so does the fundamental insight that all models in essence are equal).

Among the machine configurations two subclasses are of importance. Given a program p and data x there exists an *initial configuration* $c_0(p, x)$ representing the start of the computation of p on input x. The other class consists of the *final configurations* being those configurations which have no legal successor according to the program. A *full computation* of p on x is a computation starting with the initial configuration $c_0(p, x)$ and ending in some final configuration. The program p *terminates* on input x if some full computation of p on x exists.

Given a computation, be it a full one or not, it must be very easy to check whether it follows a given program or not. Given a program and some input data construction of the initial computation must be very easy as well. Moreover, for final configurations it must be very simple to extract from this configuration the result of the computation leading to it, be it some output data or just a single status bit informing the world whether the computation was successful (*accept*) or a failure (*reject*). So almost everything about computations should be easy to read and understand.

We can express the above properties by introducing formula's like:

$$\gamma \models p(x) = y \text{ iff } \gamma \text{ represents a full computation of } p \text{ on } x \text{ yielding output } y$$

$$\gamma \models p(x) < \infty \text{ iff } \gamma \text{ represents a full computation of } p \text{ on } x$$

The one thing what is hard about computing is that one can in general not predict how many transitions the computation will require before having transformed the initial configuration into a final one. This number of configurations is known as the *time* consumed by the computation. Neither can one estimate how large the individual configurations encountered during the computation will turn out to be. This size is known as the *space* consumed by the computation. Time and space are the fundamental measures involved in the study of *Computational Complexity*.

The concept of a machine using *memory* or not also is related to the structure of these configurations. If a configuration aside of the input data consists of only a bounded structure whose size is independent of the input size we say that the program or machine is a *finite state device* or a *finite automaton*. Otherwise the machine consumes some form of memory; how it uses it may be subjected to various constraints involving size and access methods. These issues are studied in *Automata Theory*.

2.2 Recursion Theory

Given the above model it becomes possible to define what it means for something to be effectively computable and/or decidable. This is represented by the notion of a *recursive* function or predicate:

Definition 1 *A function f mapping an argument domain A into a value domain B is recursive iff the following conditions can be fulfilled:*

- *both A and B after suitable encoding are easy to recognize subsets of the data and result domain of our machine model*
- *there exists a program p which is deterministic and terminates on all inputs $x \in A$*
- *$f(x) = y \Leftrightarrow \gamma \models p(x) = y$ where γ denotes the unique full computation of p on x*

If we omit in the above definition the condition that the program terminates on all inputs $x \in A$ the function is called *partial recursive* on A.

Definition 2 *A predicate q over an argument domain A is recursive iff the following conditions can be fulfilled:*

- *A after suitable encoding is an easy to recognize subset of the data domain of our machine model*

- *there exists a program p which is deterministic and terminates on all inputs $x \in A$*
- *$q(x) \Leftrightarrow \gamma$ accepts, where γ denotes the unique full computation of p on x*

As it turns out in the latter definition determinism is inessential, but termination is. The same holds for the concept of recursive function provided one is willing to replace the notion of a function by a many-valued function or a relation.

The elimination of nondeterminism is based on the observation that the possible computations by a nondeterministic machine have the structure of a finitely branching computation tree. If all branches in this tree terminate then the entire tree is finite (evidently the converse holds as well; all branches in some finite tree are finite). And given some machine whose computation tree on some input is finite we can design a different deterministic machine which simulates a full tree traversal, and thus can determine what the earlier machine may do.

The notion of recursiveness is easily transferred to other mathematical structures like sets and graphs. A (sub)set $B \subseteq A$ is recursive in A provided the membership predicate $q(x) := x \in B$ is recursive over A. A graph $< V, E >$ is recursive provided V is a recursive set and E is a recursive subset of $V \times V$, etc.

There is one more fundamental notion in computation theory required for the sequel. For recursive sets membership of an arbitrary element can be both affirmed or denied. If we drop the latter requirement we obtain the weaker notion of a *Recursively Enumerable* set:

Definition 3 *A set B is called* Recursively Enumerable *provided it is equal to $F(A)$ for some recursive set A and recursive function F*

The above definition is different from but equivalent to the standard definition in the literature, where Recursively Enumerable sets are defined to be the domain of a (partial) recursive function. In fact one has the following equivalences:

Proposition 1 *The following are equivalent:*

- *a) The set B is recursively enumerable.*
- *b) The set B is the domain of some partial recursive function.*
- *c) The set $B = \emptyset$ or B is the range of some total recursive function H.*

proof:
$b \Rightarrow a$: Let B be the domain of the partial function f, and let p be a program for f; then consider the set A consisting of all pairs $< x, \gamma >$ such that $\gamma \models p(x) < \infty$ which (by the construction of our theory) is an easy and therefore recursive set. F will map the pair $< x, \gamma >$ on x; this map also is evidently recursive; moreover $F(A)$ equals the domain of f.
$c \Rightarrow b$: If $B = \emptyset$ then B is the domain of a function computer by some program which never terminates. Otherwise B is the domain of the function computed by some program which on input x looks for the first argument y with $H(y) = x$.

$a \Rightarrow c$: Assume that $B = F(A)$ with A and F recursive; now construct a total recursive function f which on input z inspects all possible arguments y in the Universe preceding z in the ordering; whenever it finds a $z \in A$ it evaluates $F(z) = x$ The program for f remembers the last such x found, and at the end of the computation this x is given as output. Evidently this program computes a function whose range equals A, but it doesn't yet compute a function which is total, since for small arguments y there may not exist a $z \in A$ preceding y. Now either B is empty, and A must be empty as well, in which case c is satisfied, or we can select a fixed $x_0 \in B$ and define $f(y) = x_0$ for these initial values of y. $\Box$

A classic result which is proven by the technique known as *Diagonalization* shows that there exist sets which are recursively enumerable but not recursive. Such sets are called *Undecidable*. The proof involves the notion of a *Universal Machine*, a single program which reads pairs consisting of an encoded program p and some input x and subsequently simulates the behavior of p on x. If we assume that all recursively enumerable sets are recursive, then so is the set Δ of all codes i such that i if interpreted as a program given i as input rejects. Now assuming that Δ itself is computed by some program j leads to the contradictory conclusion that $j(j)$ accepts if it rejects and conversely.

Once having established the existence of undecidable sets it is not difficult to locate them everywhere in logic, mathematics, computer science and other trades and sciences. The key notion used for this purpose is the *reduction*, a computable total (and frequently very easy to compute) function $h : A \to B$ with the property that we have $f(x) \in B \Leftrightarrow x \in A$. Existence of such a reduction can be used in two directions: it shows that A is recursive provided B is; similarly, if A is known to be undecidable then so is B.

Typical examples of undecidable sets (other than the diagonal set Δ introduced above), are the *halting problem* consisting of pairs $< p, x >$ such that $p(x) < \infty$, the set of true formulas in arithmetic, and the set of programs computing a total recursive function. As expressed by *Rice's theorem* in fact every non trivial *extensional* property of programs is undecidable. These are properties which depend on the behavior of the program only in the sense that two programs which do the same (i.e., they compute the same function) either both have the property or neither of the two has it; non-triviality means that some program has the property while another one hasn't.

If we consider functions rather than sets we observe that there exists a class of total recursive functions computed by deterministic programs terminating on all inputs, and a class of partial recursive functions which are computed by just deterministic programs. Among the latter class we find the special class of the *measurable* functions f which have the special property that their graph, being the set of pairs $< x, f(x) >$ is decidable. Not all partial recursive functions are measurable, but an important subclass including the *run-times* of programs p, being the function which on input x returns the time consumed by p on x are. The same holds for the *space* consumed by a program.

2.3 Complexity Theory and Formal Languages

Given the limitations of the real world, complexity theory makes a difference between what is computable in principle (recursive and recursively enumerable sets) and what is computable in practice. The generally accepted borderline is that, in order to be computable in practice, the run-time of some program should be bounded by a polynomial function in terms of the size of the input. The class of sets (functions) computable using these practically computable programs is called P (PF). Functions and sets which require more time (like for example exponential time) are generally held to be intractable.

The choice for this borderline has the nice property that it is sufficiently large to absorb the simulation overheads which are implicit in the proofs establishing the equivalence between the various models of machine based computation [28], whereas it is narrow enough to become a meaningful boundary between the tractable and the intractable problems.

There exists however one feature of the computational model for which this equivalence has not been established: the use of nondeterminism. The class *NP* consists of those sets which have a polynomial time bounded acceptor; equivalently stated the sets A for which a witness predicate W exists such that $W \in P$, and $x \in A \Leftrightarrow < x, w > \in W$, where w moreover should have a size bounded by some polynomial in terms of the size of x. Stated otherwise: *NP* sets are populated by puzzle instances which we don't know how to solve in a tractable way but for which checking a given solution is tractable.

The most important unsolved problem for Complexity theory (if not for Computer Science in general) is expressed by the question $P = NP$? Research during the past 25 years has been undecisive, although the conventional belief is that the answer should be negative.[1] What we do know since 25 years is the existence of characteristic sets X in *NP* which have the property that the entire class *NP* can be reduced to X by *polynomially time bounded* reductions. This implies that $P = NP$ iff $X \in P$. These characteristic sets are known by the name *NP-complete problems*; they frequently are encountered in the area of combinatorial optimization problems [7, 16, 21, 12, 15].

Similar issues arise when considering space instead of time, and when considering borderlines higher up in the complexity hierarchy. There exists a hierarchy of fundamental complexity classes for which almost every inclusion between adjacent classes hides a notorious unsolved problem, and for which we know at most levels the existence of complete problems both in the form of theoretical constructs and actual problems encountered in computer science and elsewhere [15].

I conclude this section by some remarks on the relation between Computation Theory and Theory of Formal Languages. In Formal Language Theory, the Universe always consists of finite strings over some fixed alphabet Σ. Hence a language is a subset of $\Sigma^\star$. Computation theory characterizes formal languages by looking at programs and formal devices *recognizing* or *accepting* such a language. Here a language acceptor is a partial recursive function which has the language as its domain (so the language is recursively enumerable). A language recognizer is a total recursive function over a Universe of strings witnessing that the membership predicate for the language is recursive.

The other side of formal language theory is the description of languages by means of grammars. A *formal grammar* is a (nondeterministic) computational device operating on strings. Its actions are described in terms of local string substitutions as encoded by its program (which is called the set of *production rules*). It has a unique starting configuration called the *start symbol*, and it has its terminal configurations characterized by the absence of so-called *nonterminal* symbols. A computation is called a *derivation*. The resulting string which consists of *terminal* symbols only is the string *produced by the derivation*.

The *Chomsky Hierarchy* characterizes four classes of languages, both by constraints on

[1] In fact contemporary cryptography is based on (strengthened versions of) this belief.

the structure of the production rules in the grammar, and by constraints on the program of the recognizer (acceptor). For general languages (*type-0*) there are neither constraints on the grammar rules nor the machines; evidently this class coincides with the recursively enumerable sets.

The *context sensitive* languages are characterized by grammars with rules which don't erase symbols, and (nondeterministic) recognizers which succeed in completing their computation in the space occupied by the input in the initial configuration (*linear bounded automata*).

The next class consists of the *context free* languages. Their grammars are characterized by rules which rewrite a single symbol independently of their context; the corresponding machines are nondeterministic devices consisting of a finite automaton operating on a single stack. The most restrained class is the class of *regular* languages; their grammar have rules of a very simple type, whereas their acceptors consist of a finite automaton only. These languages are furthermore characterized in terms of the so-called *regular expressions*. For details see [14].

Both context free and regular languages can be invoked for giving a specific meaning to the notion "very simple" invoked in the description of computation theory in this section.

The correspondence between complexity theory and formal language theory shows on both sides unfilled open slots. The notion of a recursive set has no analogue in the Chomsky hierarchy. Context sensitive sets coincide with nondeterministic linear space-bounded sets. The classes P and NP have no characterization in terms of grammars, whereas the context free languages have no definite characterization in terms of complexity (there remain some important open problems about the complexity of context free recognition). It is no coincidence that the machines characterizing the context free languages were the last family on the machine side of the Chomsky Hierarchy to be discovered [13]. The regular languages characterize the power of on-line memory-free computation.

3 Pragmatics entails Syntax

In this section I present my argument showing that Design Patterns, if they can be dealt with in a meaningful way at all, must populate a formal language. Assuming that patterns can be indicated to arise in some design, program or system specification, and that the evidence required for showing such an occurrence of a pattern in a program can be communicated in a standard way (by writing or over the internet), the conclusion will follow. It will be possible to gather all possible communications, extract the patterns from these communications and gather them together in a Recursively enumerable Language.

Design Patterns are supposed to describe solutions to recurring problems in software design. As such it seems reasonable to require that one can recognize a candidate pattern when someone is given one:, i.e., that one can solve the following recognition problem:

PATTERNNESS
Input: a description π of a candidate pattern,
Question: Does π represent a true design pattern ?

This would yield an immediate positive answer to the question of the nature of the set of possible patterns. Requiring the above decision problem to be solvable is equivalent to requiring that the patterns form a Recursive set. The required decision method would moreover yield a definition of the concept of a pattern. It is evident that this is not what the Pattern community at this stage of the development has in mind. Patterns are not produced — they are found in existing designs [29].

I consider a simple but weird example in order to show that is not unreasonable to question the recursiveness of the set of Design Patterns. It abuses the fact that we might invoke rather curious conditions on what we consider to be a *good* design.

The circle represents the classic ideal of perfectness, and evidently we want that our designs are perfect as well. Therefore we require that, in order to become a Design Pattern, some encoding of the candidate pattern using decimal digits does occur somewhere as a consecutive block of digits in the decimal representation of the number *pi*. This riduculous requirement will make (for all we know) the above decidability requirement unlikely to be satisfied.[2] I don't believe that anybody in the Pattern Community ever has proposed such a condition. But as long as we have no fixed definition of what Patterns are someone may come forward with a definition which involves undecidable properties (for example by referring to extensional behaviour of some program captured in the pattern).

Rather than producing a pattern by performing production steps according the rules of a formal grammar, the writer of a pattern is supposed to present it to the audience at some workshop and debate its strength and weakness. So rather than focusing on patterns themselves we better look at the relation between a pattern and a program, design or software specification where this pattern is used. We should agree that something can't be a pattern unless it occurs in some program. This leads to our next recognition problem:

PATTERN OCCURRENCE
Input: a description π of a candidate pattern, and a description p of a program
Question: Does π represent a design pattern which is used in p ?

If we can decide this question it must be possible to confirm or deny that some pattern π occurs within a program p. I leave it open whether the word *program* in this context should denote the code of a program, its specification or just some description of its design. Using the first two possibilities (code or specification) we know that the program p can be represented by a string of characters, whereas the more abstract notion of a design is more loosely coupled from such linguistic restraints.

But once more it seems that requiring the above decision problem to be effectively decidable is overly restrictive. In our toy example, for all we know the occurrence of some particular pattern in the decimal representation of π might be undecidable.

Inspection of the examples in [11] makes it evident that it is not at all clear how one should decide whether a pattern arises within a system. Even in the simple case where both the pattern and the system is represented by its Object Diagram, the rules telling you how the pattern Object Diagram should be realised within the system's Object diagram, are left unspecified. Classes in the pattern can be replicated in the system, inheritance edges may be mapped onto paths, and labels may be changed. But mapping the entire

[2] It is a traditional technique for the construction of curious examples in Intuitionistic Mathematics to invoke conditional definitions where the condition involves the presence of the pattern "0123456789" in the decimal representation of π.

pattern onto a single node and thus claiming that the pattern arises in the system should be unacceptable.

What is important about the relation between patterns and programs is that whenever a pattern occurs in a program, it must be possible to convince an observer that this occurrence indeed exist. There must exist an *argument*, or *witness* for this fact. And when such an argument can be given the pattern indeed occurs. So this leads to the next (and final) version of a decision problem:

WITNESS OF PATTERN OCCURRENCE
Input: a description π of a candidate pattern, a description p of a program, and an argument w
Question: Does w represent a valid piece of evidence showing that π represent a design pattern which is used in p ?

In this form the relation indeed must be decidable. When the pattern π arises in the program p it must be possible to show that this is the case, and the argument w conveying this information must be checkable. Either the argument is convincing and we are satisfied, or the argument is defective, incomplete or meaningless and we know nothing.

In our toy example the corresponding condition indeed is decidable: given some definite pattern we can assert that it occurs somewhere in the decimal representation of π by writing out the position and evaluating an approximation of π up to the required precision. This entire computation becomes a witness for the occurrence of our pattern in π.

Going up one more level makes no sense. Asking that the argument itself possesses a *proof of argumentness,* or *meta argument* w' will not change the picture. Then it will be required that w' can be recognized for what it is good for, so the collection of all possible meta-argument invocations (w', w, π, p) will be decidable. But then the concatenation of the argument w and the meta argument w' will be a valid argument showing that π arises in p.

My first assumption on the nature of Design Patterns now can be stated as follows:

Assumption 1 *The collection Prag of triples (w, π, p) such that $w \models \pi \in p$, expressing that the argument w witnesses the fact that the pattern π arises in the program p, is a decidable set.*

Note that we have said nothing so far about the domains from which patterns, programs and/or arguments should be selected. It is however a reasonable requirement that triples as described above can be communicated by ordinary means like written or spoken language. Evidently, if we accept the possibility that being a pattern can be conveyed by means of telepathy, my argument will break down, but so will our accepted ways and means of doing science. My second assumption on the nature of design patterns therefore can be stated as:

Assumption 2 *Triples (w, π, p) such that $w \models \pi \in p$, expressing that the argument w witnesses the fact that the pattern π arises in the program p, can be communicated over the Internet.*

The impact of this assumption is that the above triples can be encoded by means of bitstrings, and thus are available as elements in a domain like $\{0,1\}^\star$ which can serve as the domain of the theory of computation. The assumption moreover entails that these communications can be understood and analyzed correctly. In particular, given the (encoded) triple, one can effectively extract from it the (encodings of the) program p, the candidate pattern π and the argument w.

As usual in computation theory we will make no further distinction between the various entities (arguments, programs and patterns) and their encodings as bitstrings in $\{0,1\}^\star$. Communicability means that encoding and decoding are effective and reversible operations (up to inessential equivalences).

These assumptions now suffice to prove our first theorem:

Theorem 1 *The collection of design patterns Patrn is a Recursively Enumerable set.*

Proof: By our two assumptions the pragmatics *Prag* is decidable; so its encoding by means of bitstrings is a recursive subset of $\{0,1\}^\star$.

Now the following two sets are obtained as the image under an effectively computable projection mapping:

$$Patocn := \{(\pi,p) \mid \exists_w[(w,\pi,p) \in Prag]\} = \{(\pi,p) \mid \exists_w[w \models \pi \in p]\}$$

$$Patrn = \{\pi \mid \exists_{w,p}[(w,\pi,p) \in Prag]\} = \{\pi \mid \exists_{w,p}[w \models \pi \in p]\}$$

The set *Prag* represents the pragmatics of the use of patterns as objects which can be witnessed to arise in concrete programs and systems. By projecting away the argument we obtain the set *Patocn* describing the relation which pattern occurs in which program. And by subsequently projecting further the set *Patrn* of design patterns remains. It is an elementary result from computation theory that the image under projection of a Recursive set is Recursively Enumerable. □

4 Decidability entails bounded witnesses

The result in the preceding section is based on the two assumptions (decidability and communicability of the pragmatics *Prag*) together with the self-evident stipulation that something only can be a pattern if it occurs within a program, and that such an occurrence can be witnessed by an argument.

This observation shows a possible road towards deciding membership in the sets *Patocn* and *Patrn*: to see whether π is a pattern find a program p in which it occurs, and to assert this latter fact find a witness w for the occurrence. If we would know how large for a given candidate pattern π this program p and this witness w can be (if they exist at all) we obtain an effective (though rather inefficient) decision method: investigate all triples (w,π,p) up to the given lengths for w and p until you find a triple satisfying $w \models p \in \pi$; if so π is a pattern in *Patrn*; otherwise, if no such triple can be found π is not a pattern.

So bounding the length of arguments and programs witnessing patternhood entails that the set of patterns is recursive. However, as it turns out, the converse holds as well: if the set of patterns is recursive there must exist a bound on the length of the shortest argument–program pair witnessing to it being a pattern:

Theorem 2 *The following are equivalent:*
a) The set Patrn of patterns is recursive (decidable)
b) There exists a total recursive (numeric) function f such that for every pattern π there exist program p and argument w satisfying $w \models \pi \in p$ and $|w|, |p| \leq f(|\pi|)$.

proof: The implication $b \Rightarrow a$ is trivial and has been explained above. To see the other direction we introduce the following function H:

$$H(\pi) := \begin{cases} 0 & \text{if } \pi \text{ is not a pattern} \\ \min\{|w| + |p| \mid w \models \pi \in p\} & \text{otherwise} \end{cases}$$

I claim that H is a total recursive function. The above formula describes an algorithm for computing $H(\pi)$: if π is not a pattern the computation yields zero immediately, but if π is a pattern we know that an argument–program pair (w, p) satisfying $w \models \pi \in p$ must exist. So, by searching all pairs in increasing length we will find one (and in fact the shortest possible one first). By the assumption that patterns are decidable we know in which case we are, and by the decidability of *Prag* we can effectively decide for each pair (w, p) whether it is a witness or not. So the algorithm indeed terminates for every input π.

Next define the function f by

$$f(m) := \max\{H(\pi) \mid |\pi| \leq m\}$$

It follows that whenever π is a pattern there exists a witness pair (w, p) satisfying $|w|, |p| \leq |w| + |p| \leq H(|\pi|)$. Since $f(m)$ is computed as the maximum of a total recursive function H over a finite set of arguments $\{\pi \mid |\pi| \leq m\}$ it is a total recursive function as well. □

Using a similar argument one can prove the next theorem:

Theorem 3 *The following are equivalent:*
a) The set Patocn of pattern-occurrences is recursive (decidable)
b) There exists a total recursive (numeric) function f such that for every pattern π and program p there exists an argument w satisfying $w \models \pi \in p$ and $|w| \leq f(|\pi| + |p|)$.

5 Linguistic Expressivity and Complexity

The results in the previous section show that there exists a connection between decidability of *Patrn* and *Patocn* and the possibility of estimating the size of some witness argument establishing the fact that some candidate pattern indeed occurs in some program and therefore correctly may be called a pattern. In the present section I will investigate whether it is reasonable to expect that such an estimate on witness size will exist.

Inspection of the examples of patterns described in [11] is not very helpful. A complete description of some pattern would require the presence of some class diagram, an object behavioral diagram for the relevant participants and an object interaction diagram. For most patterns only the first diagram is given under the heading **Structure**.

Next one would expect that this structure diagram indeed would be reflected in the example given under the heading **Motivation**. Indeed some structural similarity is easily recognized, but any attempt to describe this similarity by more rigid formal methods (using for example the language of graph grammars) seems doomed for failure. I do consider it possible however that a bright student will train a neural network to recognize the similarity in the 23 examples (but for this strategy to work we would need *negative* examples as well).

Instead I will, for the time being, propose an (incomplete) suggestion for a definition (at least at the level of class diagrams).

The class diagram follows the OMT-like syntax used in [11]. Within this diagram some classes can de designed to be *replicable* like for example the class named `ConcreteBuilder` in the **Builder** pattern. Being replicable means that the node corresponding to this class can be copied as many times as one needs, together with its outgoing edges.

Sometimes an entire subgraph must be designated as being replicable, in order to preserve some correspondence between different nodes in the class diagram. For example in the **Abstract Factory** pattern the class named `ConcreteFactory` together with the two `Product` classes must be replicated. The same example shows also that this replication operation can be nested, since the number of `Product` classes itself can be copied several times, before they are connected to the `ConcreteFactory` class.

After having performed these replications the expanded graph must be embedded in the program's Object diagram as a subgraph. Again the latter condition is too restricted; in the **Chain of Responsibility** the inheritance edge between the `ConcreteHandler` and the `Handler` in the **Structure** section obtains an intermediate class `Widget` in the **Motivation** section.

During the replication and embedding also the labels in the class diagrams are renamed. I abstain from inventing rules for such renaming. I also have disregarded the distinction between abstract and concrete classes which should or shouldn't be preserved during replication and embedding.

Assuming that an embedding of a replicated version of a pattern diagram in a program diagram presents a valid witness of the occurrence of the pattern in the program, we can next ask what would be the consequence of such a choice of formalization.

It is not hard to see that the above choice will make the relation *Patocn* recursive. Given a pattern π and a program p the number of nodes in p gives an upper bound on the number of times a node or substructure in the pattern can be replicated. This limit determines a finite collection of replicated versions of π. For each of these versions we subsequently can test whether it can be embedded in p.

At the same time our approach makes it clear that the relation *Patocn* will be complex. The problem whether one graph can be embedded in another is known to be an *NP*-complete problem.

In fact there exists a very simple argument to show that (under minimal assumptions on what pattern occurrence should denote) the set *Patocn* is at least *NP*-hard.

For this purpose I take the freedom of introducing a pattern myself. It is a *negative* pattern [17] I wouldn't recommend to invoke for enhancing reuse; quite to the contrary it is a design pattern system designers should keep out of their products.

MESS OF ORDER k; Structural

Motivation
When modeling the interaction between k different classes of agents in a system, these agents require channels in order to reach out to each other. Designers easily are lured into subclassing the basic `Agent` class into k subclasses, providing each subclass with a reference to each of the others. The resulting class diagram containing k subclasses, each referring to each other, is very inflexible and its use should therefore be discouraged.

Structure
I omit here the self evident drawing of a class with k subclasses all referring to each other
...
Consequences
A program where the Mess of order k pattern is used becomes highly unflexible and its potential for reuse is severely diminished. This is particularly useful in case the resulting system subsequently is delivered following the 374-th Ferengi Rule of Acquisition: *Never deliver the Source Code; Only deliver the Cloned Image* [24].
...
Known Uses
Rumor goes that indeed this pattern has been used, but nobody ever dared to claim credits for it.
...

According to our embedding strategy finding an occurrence of **MESS OF ORDER** k in a program amounts to locating k classes in the program's object diagram which all inherit from a common superclass and all have references to each other. This problem is easily seen to be a special case of the **Clique** problem which is known to be NP-complete.

The pattern **MESS OF ORDER** k introduced here includes the parameter k, which should be instantiated before a concrete pattern is realised. The NP-hardness is maintained when we write down the value of k using *unary* notation: k is represented by writing k strokes next to each other. Evidently we might represent k in binary or decimal notation as well. So we must consider the fact that Design Patterns may refer to numbers in their description,and these numbers must be represented in some way.

This observation points towards yet another source of potential hardness. If we allow ourselves to invoke numbers in pattern descriptions, and these numbers are written in binary or decimal representation, these patterns when expanded into concrete graphs may become exponentially large and so may the programs where these patterns occur. This is evident for the degenerate pattern which just requires the presence of k different classes in some program.

However, by the absence of syntactic constraints on pattern descriptions, super exponential expansion becomes possible as well. In the above degenerate pattern we may describe the number k *any way we like*, as long as we describe a unique number. We may write down any code which eventually expands into a number. so we could invent some fancy numbering system for very large integers, or, more generally, use a fixed encoding by means of some recursive function f: the number $f(k)$ is represented by the pair consisting of (an encoding of) the program for f and the argument k.

However, under the above freedom the smallest program containing a pattern of length

t may require size $R(t - C)$ where C is some absolute constant and R denotes the so-called *Rado-function*, $R(t)$ being the largest number computed by some terminating t-state Turing Machine working on an initially empty input. It is known that $R(t)$ grows faster than every total recursive function [23, 25], which shows by our characterization result in section 4 that *Patrn* will become non-recursive.

It is unlikely that the pattern community is interested in these degenerate patterns, but the message remains: *Complete Freedom, and Absence of Syntactic Constraints on Pattern Description may result in Undecidability.*

Aside from the undecidability provoked by the absence of linguistic constraints on Pattern descriptions, it remains very likely that determining the presence of simple patterns in systems becomes at least *NP*-hard, since our example requires only minimal assumptions on the expressive power for our language. This however shouldn't be taken too serious; *NP*-hardness is something we encounter wherever we go [12].

6 The Best of Both Worlds

Having arrived at the last section of this paper I return to the question what we, researchers in fundamental computer science, can say about the linguistic status of Design Patterns. No statement or judgement can be definite since the results all are subject to assumptions.

Instead my observations should be understood to be a challenge to the Patterns Community: *Just state how you want your world to be, and we can point out for you the consequences of your choices.*

Below I will sketch a number of possibilities how the realm of Design Patterns may be structured. It remains to the Patterns Community to decide whether they want to accept the relevant assumptions or want to deny them, but the consequences of such a choice should be clear. Their answer will also determine on what basis they will communicate with the other schools in Computer and Information Science.

the Realm of Anarchy The first world is the world which results if my first assumption is rejected: the set *Prag* is not required to be decidable, and neither are *Patrn* and *Patocn*.

In this world complete freedom is available but its price is total anarchy. It means that there doesn't exist a commonly agreed basis for determining whether someone's argument for a candidate pattern to arise in a program holds or doesn't hold. Arguments and meta-arguments will drag on forever over the internet.

As long as we don't make a clear distinction between Design Patterns and Non-Design Patterns, this world may be closer to the actual world than I feel comfortable with. Therefore I restrict myself in these and further considerations to Design Patterns only.

It should be evident that from this world communication with more traditional established schools in Computer and Technology Science is hard if not impossible.

the Empire of Unconstrained Pragmatics In the second world the set *Prag* is decidable but *Patocn* and *Patrn* are not.

In this world the rules of the pragmatics game have been established, but the games themselves may consume all the time of the world. A candidate pattern indeed may be a true pattern but proving it may require either an unrealistically large program or some unmanageable certificate (or both). In this world the Design Pattern community speaks a definitive well structured common language, albeit one with practically unlimited expressive power.

the Kingdom of Constrained Pragmatics In the third world both *Prag* and *Patocn* are decidable, but *Patrn* is not.

In this world occurrence of patterns within a program can be established, but the structure of Design Patterns itself is so expressive that they can refer to unrealistically large programs. The use of patterns can be understood but there is no limitation on what they can express.

the Federation of Prudence In the fourth world all sets *Prag*, *Patocn* and *Patrn* are decidable.

In this world there exists a fair degree of proportionality between the patterns and the smallest programs which realize these patterns. This state of affairs, however, does require limitations on linguistic expressiveness.

the Republic of Lost Control A curious intermediate world is obtained by requiring that *Prag* and *Patrn* are decidable whereas *Patocn* is not.

In this world patterns can be realised in moderately small programs, and these occurrences can be certified using bounded arguments. These patterns also may be realised in much large programs and for those larger programs the certification process becomes unmanageable.

Degenerate Worlds The world where *Patrn* and *Patocn* are decidable whereas *Prag* is not represents a degenerate situation; in this world it seems that an incorrect notion of witness argument is used, since it follows from the decidability assumptions that an alternative decidable set of witness certificates exist. It is a world where one has both arguments which are decisive and arguments which remain open to debate forever.

Similar observations can be made about the remaining two worlds where only one of *Patrn* and *Patocn* is decidable and *Prag* is undecidable.

Aside from these decidability issues we might look into the various possibilities with respect to the complexity of these sets once they are found to be decidable.

It seems reasonable to assume that the set *Prag* is not only decidable but in fact easy to decide (say in Polynomial time). We can always reach this state of affairs if we are willing to expand the size of our witness arguments.

For the set *Patocn* we have seen evidence that this set is at least *NP*-hard. The set may turn out to be harder as well; there seems to reside a sufficient amount of intensionality involved in design patterns that known techniques for establishing *PSPACE*-hardness can be used, but I leave this for future research.

The hardness of the set *Patrn* itself (assuming that it is decidable) depends on its linguistic expressivity. The Pattern Community should determine what expressive power they need, and then we may address this issue.

In my introduction I mentioned the possibility of a *Calculus of Design Patterns*. So far we only looked at individual patterns, programs and witness certificates. However, programs populate a programming language which is enriched by rules on how to compose programs using operations like merge, union or sequential or parallel composition. Similarly we may consider the possibility of composing patterns by operations on more elementary ones. It then becomes a problem to invent corresponding operators for combining witness statements in order to show that some composed pattern arises in some combination of programs.

In formula: we like to invent groups of operators $\triangle, *, \sharp$ with the property that we can infer $w_1 \triangle w_2 \models \pi_1 * \pi_2 \in p_1 \sharp p_2$ from the assumption that $w_1 \models \pi_1 \in p_1$ and $w_2 \models \pi_2 \in p_2$. Solving this problem will provide us with the basis for a Calculus of Design Patterns, and a justification for the study of Design patterns as formal objects.

Ackowledgements

My understanding of the various issues involving Patterns has been greatly enhanced by the interaction with the students in the Design Pattern seminar (Serge Barth, Eric de Haan, H. Lieuw A Joeng, Remco Leguijt, Coen Visser, Mark Wielaard, and Bjarke Wedemeijer), which was organized in the winter term of 1997 in my department. I am grateful for the feedback, criticism and comments given by Hendrica van Emde Boas - Lubsen, Theo Janssen, Leen Torenvliet and Johan van Benthem, including the attempts to rebut my theories.

References

[1] Aho, A.V., Hopcroft, J.E. & Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley (1974)

[2] Alexander, Ch. et. al., *A Pattern Language; Towns, Buildings, Construction*, Oxford Univ. Press (1977).

[3] Alexander, Ch., *The Timeless Way of Building*, Oxford Univ. Press (1979).

[4] Auer, K. & Beck, K., *Lazy Optimization: Patterns for Efficient Smalltalk Programming*, in [29], pp. 19–42.

[5] Bentley, J., Programming Pearls, Addison Wesley (1986)

[6] Bentley, J., More Programming Pearls; Confessions of a Coder, Addison Wesley (1988)

[7] Cook, S.A., *The complexity of theorem proving procedures*, Proc. 3rd Ann. ACM Symp. Theory of Computing, 1971, 151–158

[8] Dillard, J.M., *Star Trek; First Contact*, Pocket Books (1997)

[9] Davis, M.D., *Computability and Unsolvability*, McGraw-Hill (1958)

[10] Gabriel, R.P., *Patterns of Software; Tales from the Software Community*, Oxford Univ. Press (1996)

[11] Gamma, E., Helm, R., Johnson, R. & Vlissides, J., *Design Patterns; Elements of Reusable Object-Oriented Software*, Addison Wesley (1995)

[12] Garey, M.R. & Johnson, D.S, *Computers and Intractability; a Guide to the Theory of NP-completeness*, Freeman (1979)

[13] Greibach, S.A., Formal Languages; Origins and Directions, Ann. Hist. Comput. 3 (1981), 14–41

[14] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley (1979)

[15] Johnson, D.S., *A Catalogue of Complexity Classes*, in J. van Leeuwen (ed.), Handbook of theoretical computer science, Vol. A, North Holland Publ. Comp. (1990), pp. 67–161.

[16] Karp, R.E., *Reducibility among Combinatorial Problems*, in Miller, R.E., & Thatcher, J.W. (eds.), Complexity of Computer Computations, Plenum (1972), 85–103

[17] Kim, J.J., & Benner, K.V., *Implementation Patterns for the Observer Pattern*, in [29], pp. 75–86

[18] Kleene, S.C., *Introduction to Metamathematics*, van Nostrand (1952)

[19] Knuth, D.E., *The Art of Computer Programming, Vol. 1; Fundamental Algorithms*, Addison Wesley (1968)

[20] Knuth, D.E., *Literate Programming*, CSLI Lecture Notes 27, Stanford, (1992)

[21] Levin, L.A., *Universal sorting Problems*, Problems of Information Transmission 9 (1973) 265–266.

[22] Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall (1981)

[23] Li, Ming & Vitányi, P.M.B., *An Introduction to Kolmogorov Complexity and its Applications*, Springer (1993)

[24] Quark, *The Ferengi Rules of Acquisition*, I.S. Behr (ed.), Pocket Books (1995)

[25] Rado, T., *On non-computable functions*, Bell System Tech. J. XX, 1962, 877–884

[26] Rogers, H. Jr, *Theory of Recursive Functions and Effective Computability*, Mc Graw-Hill Series in Higher Mathematics (1967)

[27] Turing, A.M., *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Math. Soc. ser. 2, 42 (1936) 230-265

[28] van Emde Boas, P. *Machine models and simulations*, in J. van Leeuwen (ed.), Handbook of theoretical computer science, Vol. A, North Holland Publ. Comp. (1990), pp. 1–66

[29] Vlissides, J.M., Coplien, J.O. & Kerth, N.l. (eds.), *Pattern Languages of Program Design 2*, Addison Wesley (1996)