32 Conclusion: The Master Programmer

Chapter Objectives This chapter provides a summary and discussion of the primary idioms and design patterns presented in our book.

Chapter Contents

32.1 Language Paradigm-Based Abstractions and Idioms

In the Introduction to this book, we stated that we wanted to do more than simply demonstrate the implementation of key AI algorithms in some of the major languages used in the field. We also wanted to explore the ways that the problems we try to solve, the programming languages we create to help in their solution, and the patterns and idioms that arise in the practice of AI programming have shaped each other. We will conclude with a few observations on these themes.

More than anything else, the history of programming languages is a history of increasingly powerful, ever more diverse abstraction mechanisms. Lisp, the oldest of the languages we have explored, remains one of the most dramatic examples of this progression. Although procedural in nature, Lisp was arguably the first to abstract procedural programming from such patterns as explicit branching, common memory blocks, parameter passing by reference, pointer arithmetic, global scoping of functions and variables, and other structures that more or less reflect the underlying machine architecture. By adopting a model based on the theory of recursive functions, Lisp provides programmers with a cleaner semantics, including recursive control structures, principled variable scoping mechanisms, and a variety of structures for implementing symbolic data structures.

Like Lisp, Prolog bases its abstraction on a mathematical theory: in this case, formal logic and resolution theorem proving. This allows Prolog to abstract out procedural semantics almost completely (the left to right handling of goals and such pragmatic mechanisms as the cut are necessary exceptions). The result is a declarative semantics that allows programmers to view programs as sets of constraints on problem solutions. Also, because grammars naturally take the form of rules, Prolog has not only proven its value in natural language processing applications, as well as a tool for manipulating formal languages, such as compilers or interpreters.

Drawing in part on the lessons of these earlier languages, object-oriented languages, such as Java, offer an extremely rich set of abstractions that support the idea of organizing even the most ordinary program as a model

of its application domain. These abstractions include class definitions, inheritance, abstract classes, interfaces, packages, overriding of methods, and generic collections. In particular, it is interesting to note the close historical relationship between Lisp and the development of object languages. Although Smalltalk was the first "pure" object-oriented language, it was closely followed by many object-oriented Lisp dialects. This relationship is natural, since Lisp laid a foundation for object-orientation through such features as the ability to manipulate functions as s-expressions, and the control over evaluation it gives the programmer. Java has continued this development, and is particularly notable for providing powerful software engineering support through development environments such as Eclipse, and the large number of packages it provides for user data structures, network programming, user interface implementation, web-based implementation, Artificial Intelligence, and other aspects of application development.

In addition to – or perhaps because of – their underlying semantic models, all these languages support more general forms of abstraction. The organization of programs around abstract data types, "bundles" of data structures and operations on them, is a common device used by good programmers – no matter what language they are using. Meta-linguistic abstraction is another technique that is particularly important to Artificial Intelligence programming. The complexity of AI problems clearly requires powerful forms of problem decomposition, but the ill-formed nature of many research problems defies such common techniques as top-down decomposition. Meta-linguistic abstraction addresses this conundrum by enabling programmers to design languages that are tailored to solving specific problems. It tames hard problems by abstracting their key features into a meta language, rather than decomposing them into parts. The general search algorithms, expert system shells, learning frameworks, semantic networks, and other techniques illustrated in this book are all examples of meta-linguistic abstraction.

This diversity of abstraction mechanisms across languages underlies a central theme of this book: the relationship between programming languages and the idioms of their use. Each language suggests a set of natural ways of achieving common programming tasks. These are refined through practice and shared throughout the programmer community through examples, mentoring, conferences, books, and all the mechanisms through which any language idiom spreads. Lisp's use of lists and CAR/CDR recursion to construct complex data structures is one of that language's central idioms; indeed, it is almost emblematic of the language. Similarly, the use of rule ordering in Prolog, with non-recursive terminating statements preceding recursive rules appearing throughout Prolog programs is on of that language's key idioms. Object-oriented languages rely upon a particularly rich set of idioms and underscore the importance of understanding and using them properly.

Java, for example, adopted the C programming language syntax to improve its learnability and readability (whether or not this was good idea continues to be passionately debated). It would be possible for a programmer to write Java programs that consisted of a single class with a static main method

that called additional main methods in the class. This program might function correctly, but it would hardly be considered a good java program. Instead, quality Java programs distribute their functionality over relatively large numbers of class definitions, organized into hierarchies by inheritance, interface definitions, method overloading, etc. The goal is to reflect the structure of the problem in the implementation of its solution. This not only brings into focus the use of programming languages to sharpen our thinking by building epistemological models of a problem domain, but also supports communication among developers and with customers by letting people draw on their understanding of the domain.

There are many reasons for the importance of idioms to good programming. Perhaps the most obvious is that the idiomatic patterns of language use have evolved to help with the various activities in the software lifecycle, from program design through maintenance. Adhering to them is important to gaining the full benefits of the language. For example, our hypothetical "Java written as C" program would lack the maintainability of a well-written Java program.

A further reason for adhering to accepted language idioms is for communication. As we will discuss below, software development (at least once we move beyond toy programs) is a fundamentally social activity. It is not enough for our programs to be correct. We also want other programmers to be able to read them, understand the reasons we wrote the program as we did, and ultimately modify our code without adding bugs due to a misunderstanding of our original intent.

Throughout the book, we have tried to communicate these idioms, and suggested that mastering them, along with the traditional algorithms, data structures, and languages, is an essential component of programming skill.

32.2 Programming as a Tool for Exploring Problem Domains

Idioms are also bound up – along with the related concept of design patterns, also discussed below – with an idea we introduced in the book's introduction: programming languages as tools for thinking. In the early stages of learning to program, the greatest challenges facing the student are in translating a software requirement, usually a homework assignment, into a program that works correctly. As we move into professional-level research or software development, this changes. We are seldom given clear, stable problem statements; rather, our job is to interpret a vague customer need or research goal and project it into a program that meets our needs. The languages we have addressed in this book are the product of many person-decades of theoretical development, experience, and insight. They are not only tools for programming computers, but also for refining our understanding of problems and their solution.

Illustrating this idea of programming languages as tools for thinking has been one of our primary goals in writing this book. Lisp is the oldest, and still one of the best, examples of this. The s-expression syntax is ideally suited for constructing symbolic data structures, and, along with the basic cons/car/cdr operations, provides an elegant foundation for structures as

diverse as lists, trees, frames, networks, and other types of knowledge representation common to Artificial Intelligence. A search of early AI literature shows the power of s-expressions as both a basis for symbolic computing and for communication of theoretical ideas: numerous articles on knowledge representation, learning, reasoning, and other topics use s-expressions to state theoretical ideas as natural science uses algebra.

Prolog continues this tradition with its use of logical representation and declarative semantics. Logic is the classic "tool for thinking," giving a mathematical foundation to the disciplines of clarity, validity, and proof. Subtler is the idea of declarative semantics, of stating constraints on a problem solution independently of the procedural steps used to realize those constraints. This brings a number of benefits. Prolog programs are notoriously concise, since the mechanisms of procedural computing are abstracted out of the logical statement of problem constraints. This concision helps give clear formulation to the complex problems faced in AI programming. Natural language understanding programs are the most obvious example of this, but we also call the reader's attention to the relative ease of writing meta-interpreters in Prolog. This discipline of meta-linguistic abstraction is a quintessential way a language assists in our thinking about hard problems.

Java's core disciplines of encapsulation, inheritance, and method extension also reflect a heritage of AI thinking. As a tool for thinking, Java brings these powerful disciplines to problem decomposition and representation, metalinguistic abstraction, incremental prototyping, and other forms of problem solving. An interesting example of the subtle influence object-oriented programming has on our thinking can be found in comparing the declarative semantics of Prolog with the static structure of an object-oriented program.

Although we have no "hard" data to prove this, our work as both engineers and teachers has convinced us that the more experienced a Java programmer becomes, the more classes and interfaces we find in their programs. Novice programmers seem to favor fewer classes with longer methods, most likely because they lack the rich language of idioms and patterns used by skilled object-oriented designers. Breaking a program down into a larger number of objects brings several obvious benefits, including ease of debugging and validating code, and enhanced reuse. Another benefit of this is a shift of program semantics from procedural code to the static structure of objects and relations in the class structure. For example, a well-designed class hierarchy with the use of overloaded methods can eliminate many if-then tests in the program: the class "knows" which method to use without an explicit test. For this reason, Java programmers frown on the use of operators like `instanceof` to test explicitly for class membership: the object should exploit inheritance to call the proper method rather than use such tests.

The analogy of this to Prolog's declarative semantics is useful: both techniques move program semantics from dynamic execution to static structure. The static structure of objects or assertions can be understood by inspection of code, rather than by stepping through executions. It can be

analyzed and verified in terms of things and relations, rather than the complexities of analyzing the many paths a program can take through its execution. And, it enhances the use of the programming language as a tool for stating theoretical ideas: as a tool for thinking.

32.3 Programming as a Social Activity

As programming has matured as a discipline, we have also come to recognize that teams usually write complex software, rather than a single genius laboring in isolation. Both authors work in research institutions, and are acutely aware that the complexity of the problems modern computer science tackles makes the lone genius the exception, rather than the rule. The most dramatic example of this is open-source software, which is built by numerous programmers laboring around the world. To support this, we must recognize that we are writing programs as much to be read by other engineers as to be executed on a computer.

Software Engineering and AI

This social dimension of programming is most strongly evident in the discipline of software engineering. We feel it unfortunate that many textbooks on software engineering emphasize the formal aspects of documentation, program design, source code control and versioning, testing, prototyping, release management, and similar engineering practices, and downplay the basic source of their value: to insure efficient, clear communication across a software development team.

Both of this book's authors work in research institutions, and have encountered the mindset that research programming does not require the same levels of engineering as applications development. Although research programming may not involve the need for tutorials, user manuals and other artifacts of importance to commercial software, we should not forget that the goal of software engineering is to insure communication. Research teams require this kind of coordination as much as commercial development groups. In our own practice, we have found considerable success with a communications-focused approach to software engineering, treating documentation, tests, versioning, and other artifacts as tools to communicate with our team and the larger community. Thinking of software engineering in these terms allows us to take a "lightweight" approach that emphasizes the use of software engineering techniques for communication and coordination within the research team. We urge the programmer to see their own software engineering skills in this light.

Prototyping

Prototyping is an example of a software engineering practice that has its roots in the demands of research, and that has found its way into commercial development. In the early days, software engineering seemed to aim at "getting it right the first time" through careful specification and validation of requirements. This is seldom possible in research environments where the complexity and novelty of problems and the use of programming as a tool for thinking precludes such perfection. Interestingly, as applications development has moved into interactive domains that must blend into the complex communication acts of human communities, the goal of "getting it right the first time" has been rejected in favor of a prototyping approach.

We urge the reader to look at the patterns and techniques presented in this book as tools for building programs quickly and in ways that make their semantics clear – as tools for prototyping. Metalinguistic abstraction is the most obvious example of this. In building complex, knowledge-based systems, the separation of inference engine and knowledge illustrated in many examples of this book allows the programmer to focus on representing problem-specific knowledge in the development process.

Similarly, in object-oriented programming, the mechanisms of interfaces, class inheritance, method extension, encapsulation, and similar techniques provide a powerful set of tools for prototyping. Although often thought of as tools for writing reusable software, they give a guiding structure to prototyping. "Thin-line" prototyping is a technique that draws on these object-oriented mechanisms. A thin-line prototype is one that implements all major components of a system, although initially with limited complexity. For example, assume an implementation of an expert-system in a complex network environment. A thin-line prototype would include all parts of the system to test communication, interaction, etc., but with limited functionality. The expert system may only have enough rules to solve a few example problems; the network communications may only implement enough messages to test the efficiency of communications; the user interface may only consist of enough screens to solve an initial problem set, and so on.

The power of thin-line prototypes is that they test the overall architecture of the program without requiring a complete implementation. Once this is done and evaluated for efficiency and robustness by engineers and for usability and correctness by end users, we can continue development with a focused, easily managed cycle of adding functionality, testing it, and planning. In our experience, most AI programs are built this way.

Reuse

It would be nearly impossible to write a book on programming without a discussion of an idea that has become something of a holy grail to modern software development: code reuse. Both in industry and academia, programmers are under pressure, not only to build useful, reliable software, but also to produce useful, reusable components as a by-product of that effort. In aiming for this goal, we should be aware of two subtleties.

The first is that reusable software components rarely appear as by-products of a problem-specific programming effort. The reason is that reuse, by definition, requires that components be designed, implemented, and tested for the general case. Unless the programmer steps back from the problem at hand to define general use cases for a component, and designs, builds, tests, and documents to the general cases, it is unlikely the component will be useful to other projects. We have built a number of reusable components, and all of them have their roots in this effort to define and build to the general case.

The second thing we should consider is that actual components should not be the only focus of software reuse. Considerable value can be found in reusing ideas: the idioms and patterns that we have demonstrated in this book. These are almost the definition of skill and mastery in a programmer, and can rightly be seen as the core of design and reuse.

32.4 Final Thoughts

It has been our goal to give the reader an understanding of, not only the power and beauty of the programming languages Prolog, Lisp, and Java, but also of the intellectual depth involved in mastering them. This mastery involves the languages syntax and semantics, the understanding of its idioms of use, and the ability to project those idioms into the patterns of design and implementation that define a well-written program.

In approaching this goal, we have focused on common problems in Artificial Intelligence programming, and reasoned our way through their solution, letting the idioms of language use and the patterns of program organization emerge from that process. The power of idioms, patterns, and other forms of engineering mastery is in their application, and they can have as many realizations, as many implementations as there are problems that they may fit. We hope our method and its execution in this book have helped the student understand the deeper reasons, the more nuanced habits of thinking and perception, behind these patterns. This is, to paraphrase Einstein, less a matter of knowledge than of imagination.

We hope this book has added some fuel to the fires of our readers' imaginations.