The Art of the Metaobject Protocol

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow

The MIT Press
Cambridge, Massachusetts
London, England

Acknowledgments

The work described here is synthetic in nature, bringing together techniques and insights from several banches of computer science. It has also been essentially collaborative; we have had the pleasure to meet and work with people from a number of communities.

The largest of these communities has been the users of our prototype CLOS implementation, PCL. By using and experimenting with early metaobject protocols, these people provided a fertile ground for the ideas behind this work to grow. Their enthu- siasm provided energy for this work and their successes provided the insight. Everyone who ever used the metaobject protocol facilities of PCL contributed to this work, but certain people deserve particular mention. Ken Anderson, Jim Kempf, Andreas Paepcke and Mike Thome were experts on different aspects of Metaobject Protocol use; their contributions to the Common-Loops mailing list helped many others learn to use the CLOS Metaobject Protocol. Richard Harris, Yasuhiko Kiuchi and Luis Rodriguez made significant contributions to the design and maintenance of PCL, a task that was also helped by each of the Common Lisp vendors who suggested implementation-specific ways to enhance PCL performance. The project also benefited from a number of users who had the courage and conviction to attempt large projects with early versions of the CLOS Metaobject Protocol: Ken Anderson, James Bennett, John Collins, Angela Dappert-Farquhar, Neil Goldman, Warren Harris, Reed Hastings, Chuck Irvine, Jim Kempf, Joshua Lubell, Yoshihiro Masuda, Phillip McBride, Steven Nicoud, Greg Nuyens, Andreas Paepcke, Peter Patel-Schneider, Dan Rabin, Doug Rand, George Robertson, Larry Rowe, Richard Shapiro and Arun Welch.

Electronic mail has been essential to the dialogue in this project. In particular, the CommonLoops mailing list has been the home of the PCL user community. Yasuhiko Kiuchi maintained this list for three years, overseeing its growth from less than onehundred to almost eight-hundred readers. More recently, Arun Welch has taken this list over, maintaining both it and the gateway to the comp.lang.clos newsgroup.

Masayuki Ida has been the primary force for establishing and nurturing a PCL and CLOS Metaobject Protocol user community in Japan. His success at mediating the cross-cultural language and stylistic differences has enabled the community to include the Japanese users.

The CLOS design community has also been involved in this project. The people involved in that effort, or in the larger Common Lisp standardization effort, have participated in the development of both the fundamental principles of metaobject protocol design presented in Part I and in the full CLOS Metaobject Protocol presented in Part II. They are: Kim Barrett, Eric Benson, Scott Cyphers, Harley Davis, Linda DeMichiel, Gary Drescher, Patrick Dussud, John Foderaro, Richard P. Gabriel, David Gray, Ken Kahn, Sonya Keene, Jim Kempf, Larry Masinter, David A. Moon, Andreas Paepcke, Chris Richardson, Alan Snyder, Guy Steele, Walter van Roggen, Dan Weinreb, Jon L White and Jan Zubkoff.

Work on reflection has been another source of inspiration guiding the design of the CLOS Metaobject Protocol and influencing its presentation. Smith's 3-Lisp had shown how the framework of a simple reflective interpreter can be used to clarify the relation between a language's reflective facilities and its implementation, by factoring out potentially distracting issues of circularity. Jim des Rivieres, who joined the project at a relatively late stage, used his experience with reflection to develop the Closette implementation and pedagogical structure on which Part I of the book is based. We would especially like to thank the following members of the reflection community, who have made contributions to the development of this work: Giuseppe Attardi, Pierre Cointe, Roman Cunis, Mike Dixon, Brian Foote, Nicolas Graube, John Lamping, Pattie Maes, Satoshi Matsuoka, Ramana Rao and Takuo Watanabe.

The manuscript for this book has taken shape over a period of years. In addition to the members of the CLOS community previously mentioned, we would like to thank the following people for their extensive feedback on various drafts: Hal Abelson, Pierre Cointe, Doug Cutting, Mike Dixon, Brian Foote, Volker Haarslev, Masayuki Ida, Yasuhiko Kiuchi, Wilf LaLonde, John Lamping, Stan Lanning, Yoshihiro Masuda, Satoshi Matsuoka, Ramana Rao, Brian Smith, Deborah Tatar, Dave Thomas, Takuo Watanabe, Mark Weiser and Peter Wegner.

Finally, there are our friends and colleagues at Xerox PARC, who have helped and supported us through this long project. We thank them for the help they have given us, and for making PARC such a stimulating and enjoyable place to work. In particular, we want to thank: Bob Bauer, Alan Bawden, Nora Boettcher, John Seely Brown, Doug Cutting, Johan de Kleer, Mike Dixon, Mimi Gardner, David Goldstone, Volker Haarslev, Ken Kahn, Yashiko Kiuchi, John Lamping, Stan Lanning, Susi Lilly, Larry Masinter, Ramana Rao, Jonathan Rees, George Robertson, Luis Rodriguez, Brian Smith, Mark Stefik, Deborah Tatar and Mark Weiser.

Introduction

Modern programming language design lives in tension between two apparently conflicting demands. On the one hand, high-level languages such as Scheme, Prolog, and ML incorporate significant advances in elegance and expressive power. On the other hand, many industrial programmers find these languages too "theoretical" or impractical for everyday use, and too inefficient. As a result, these languages are often used only in academic and research contexts, while the majority of the world's mainline programming is conducted in such languages as C and C++, distinguished instead for their efficiency and adaptability.

This book is about a new approach to programming language design, in which these two demands of elegance and efficiency are viewed as compatible, not conflicting. Our goal is the development of languages that are as clean as the purest theoretical designs, but that make no compromises on performance or control over implementation.

The way in which we have achieved elegance and efficiency jointly is to base language design on *metaobject protocols.* Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language.

Languages that incorporate metaobject protocols blur the distinction between language designer and language user. Traditionally, designers are expected to produce languages with well-defined, fixed behaviors (or "semantics"). Users are expected to treat these languages as immutable black-box abstractions, and to derive any needed flexibility or power from constructs built on top of them. This sharp division is thought to constitute an appropriate division of labor. Programming language design is viewed as a difficult, highly-specialized art, inappropriate for average users to engage in. It is also often assumed that a language design must be rigid in order to support portable implementations, efficient compilers, and the like.

The metaobject protocol approach, in contrast, is based on the idea that one can and should "open languages up," allowing users to adjust the design and implementation to suit their particular needs. In other words, users are encouraged to participate in the language design process. If handled properly, opening up the language design need not compromise program portability or implementation efficiency.

In a language based upon our metaobject protocols, the language implementation itself is structured as an object-oriented program. This allows us to exploit the power of object-oriented programming techniques to make the language implementation adjustable and flexible. In effect, the resulting implementation does not represent a single point in the overall space of language designs, but rather an entire region within that space.

The protocols followed by this object-oriented program serve two important functions. First, they are used by the designers to specify a distinguished point in that region, corre-

sponding to the language's default behavior and implementation. Second, they allow uters to create variant languages, using standard techniques of sub classing and specialization. In this way, users can select whatever point in the region of language designs best serves their needs.

Our development of the metaobject protocol approach has emerged hand-in-hand with our involvement, over the past several years, in the design of the Common Lisp Object System (CLOS) [CLtL, CLtLII, X3J13][1] CLOS is a high-level object-oriented language designed as part of the forthcoming ANSI Common Lisp standard. That project brought us face to face with many of the classic problems of high-level languages, including the need for compatibility with existing languages, special extensions for particular projects, and efficiency. Our effort to deal with these problems led us to develop the approach to language design based on metaobject protocols and to implement a first practical instance, a metaobject protocol for CLOS.

The purpose of this book is twofold. First, in Part I, we present metaobject protocol design by gradually deriving a simplified metaobject protocol for CLOS. Because we expect the notion of a metaobject protocol itself to evolve, we not only present the approach as we understand it today, but also point towards open issues and directions for further development.

In Part II we provide a detailed and complete description of a particular metaobject protocol we have designed for CLOS. This second CLOS metaobject protocol can be incorporated into production CLOS implementations and used for writing production-quality code.

The work reported in this book synthesizes a number of concerns and approaches from different parts of the computer science field. As a result we hope it will be of interest to a wide spectrum of the community:

- *Programming language designers* should benefit from our analysis of some of the problems users have with high-level languages. Some designers may be interested in adding a metaobject protocol to existing languages or designing a new language with a metaobject protocol.

  Augmenting a language with a metaobject protocol does not need to be a radical change. In many cases, problems with an existing language or implementation can be improved by gradually introducing metaobject protocol features. Also, compiler, debugger. and programming environment features can often be recast as metaobject protocols, with concomitant simplifications in their design, implementation, and documentation.
- *Programmers and software engineers* should be interested in our analysis of why high-level languages are often inadequate, and in our suggestions for how to improve them. Even if they are not working with a metaobject protocol, they may find that the analysis helps them conceptualize and address problems they are having with existing languages.

[1] Two of the authors (Bobrow and Kiczales) were members of the X3J13 subcommittee responsible for CLOS.

- People working with *object-oriented* languages will have two reasons to be interested in this work. First, since our approach relies extensively on object-oriented techniques, the book can simply be viewed as presenting a well-documented case-study within the object-oriented paradigm. Particular attention is given to issues of designing and documenting object-oriented protocols. Second, the book includes a discussion not only of the behavior of CLOS, but also of features that are characteristic of object-oriented programming languages in general. As well as examining how these features work, we focus on some of the issues involved in designing such languages.
- People interested in *reflection* will recognize that our approach also relies on the techniques of procedural reflection. We hope they will see what we have done as helping to bring reflection into wider practical use, by engineering reflective techniques to be robust, efficient, and easy to use.
- The *CLOS community*, being a cross-section of these other groups, will presumably share many of their interests. Furthermore, the book provides them with specific information on how to use, implement, and continue the development of the CLOS metaobject protocol.

The Problems We Faced

During the development of the CLOS standard, we realized that we were up against a number of fundamental problems. The prospective CLOS user community was already using a variety of object-oriented extensions to Lisp. They were committed to large bodies of existing code, which they needed to continue using and maintaining. Experience with these earlier extended languages had shown that various improvements would be desirable, but some of these involved incompatible changes. In fact the very notion of a single, standardized object-oriented extension to Common Lisp was an inherently incompatible change, since the set of earlier extensions were not compatible among themselves. We therefore faced a traditional dilemma: genuine needs of backward compatibility were fundamentally at odds with important goals of an improved design.

As is often the case, the situation was particularly aggravating because of an *essential compatibility* between the language being designed and each of the older ones—the fact that although they differed in surface details, they were all based, at a deeper level, on the same fundamental approach. Each, after all, was an object-oriented programming language with classes, instances, inheritance, methods, and generic functions[2] which, when called, automatically determine the appropriate method to run. From the broader perspective of the family of object-oriented programming languages, they differed only in how they interpreted

[2] In traditional object-oriented languages, the terminology used for a polymorphic operation is message. The terminology for invoking a such an operation is sending a message. In CLOS, the form taken by this functionality warrants the use 'of somewhat different terminology, which will be used throughout this book: the CLOS term for message is generic function, and the term for sending a message is calling or invoking a generic function. (Appendix A presents an introduction to CLOS.)

various aspects of object-oriented behavior: the syntax for calling generic functions; the rules for handling multiple inheritance; the rules of method lookup; etc.

Along with this first challenge, of compatibility, we faced a second one, of extensibility. As well as using a small number of major existing languages, many of the prospective CLOS users had also developed custom languages of their own. In many ways these, too, were essentially compatible with the major languages, and with the new language we wanted to design. But each had its own distinguishing characteristics, supporting a number of additional features, or implementing variant interpretations of basic object-oriented behavior. For example, some of these languages provided special mechanisms for representing the structure of instances. Others employed various special inheritance and method lookup mechanisms.

Furthermore, it turned out on examination that these added functionalities and variations in the base object-oriented model were neither arbitrary nor superfluous. Because they satisfied various application-specific requirements, these incremental differences allowed programs to be clear, easy to write, and straightforward to maintain. Without them, the users would have lost the advantage of programming in a high-level language: expressive power well-matched to the problem at hand.

It was clear, furthermore, that some prospective CLOS users would always be in a similar situation. *No matter what design was agreed upon,* there would be times when a given user, for entirely appropriate reasons, would need this or that variant of it. For example, while CLOS was being designed, it emerged that some users were interested in the support of persistent objects, of the sort that would be provided by an object-oriented database. Our challenge was to find a way to enable such users, who wanted something close to CLOS, to adapt it to fit their needs.

The third problem we faced was that of ensuring that programs written in CLOS would run efficiently.[3] Unfortunately, the very expressiveness of high-level languages makes this difficult. No single implementation strategy is likely to perform well on the full range of behaviors that user programs can be expected to exhibit.

Consider, for example, two different uses of CLOS classes. In the first, which might arise in a graphics application, instances represent screen positions, with two slots[4] for x and y coordinates. In such applications, it is safe to assume that slot access performance

[3] 3We call this familiar problem the paradox of high-level programming languages. On the one hand, the primary rationale for high-level programming languages is that they are more expressive-i.e., that they allow better formulations of what our programs are doing. On the other hand, it is widely agreed that programs written in high-level languages are usually less efficient than programs written in lower-level ones. The paradox arises from the fact that there must be things about the high-level programs that aren't being expressed more clearly, since otherwise compilers would be able to notice and exploit them, making those programs more, rather than less, efficient than their low-level counterparts. In on-going work on metaobject protocols, not discussed in this book, we are focusing directly on this issue. Our goal is to make it easier to make a piece of code simultaneously clear and fast in a high-level language than to make it fast in a low-level language.

[4] In CLOS the fields of an instance, which in many languages are called instance variables, are called slots.

is critical—the faster the better. In the second, which might arise in a blackboard system, instances can potentially have a large number of slots, but in practice any given instance only uses a small number of them. If there can be a very large number of these instances, the space taken up by the instances would be of critical concern, making it important not to waste space on all the unused slots.

Even though the *behavior* of both of these classes is well captured by the CLOS language, they would clearly benefit from different implementation strategies. As a result, an implementation that used a single strategy would at best perform well on only one. The situation is further exacerbated by the fact that the information needed to choose the best implementation strategy might be difficult or impossible for a compiler to extract from the program text, since it depends on dynamic behavior, such as how many instances will be created, or how often their slots will be accessed. This is why efficiency is a challenge: somehow, within the context of a single language design, different users should be given the specific implementation and performance profiles they need.

While these goals of compatibility, extensibility and efficiency at first seemed different, we eventually came to see them as instances of the same underlying problem, and were therefore able to address them within a common structural framework.

The underlying problem is a lack of fit: in each case, the basic language design fails to meet some particular need. Compatibility is an obvious example: as soon as the language is changed to incorporate a new feature, it fails to match up with existing bodies of code. But the other cases have the same structure. The graphics program is well served by one implementation of instance representation, but the sparsely populated slots case requires another.

The unavoidable conclusion is that no single language will ever be universally appropriate, no matter how clever its design. So we adopted a different solution. Rather than supplying the user with a fixed, single point in the space of all language designs and implementations, we would instead support a region of possible designs within that overall space. This is the essence of the metaobject protocol approach. In the case of CLOS, for example, instead of providing a single fixed language, with a single implementation strategy, the metaobject protocol extends a basic or "default" CLOS by providing a surrounding region of alternatives. Users are free to move to whatever point in that region best matches their particular requirements.

This strategy has two tangible benefits. First, relying on the metaobject protocol to deal with a wide range of users' concerns allows the base case—CLOS itself—to be simpler and more elegant. The very existence of the metaobject protocol, in other words, takes some pressure off the design of the base language, to its benefit. Second, the strategy of supporting a CLOS region, rather than a single CLOS point, enables us to solve all three of our original problems.

The compatibility problem is solved by ensuring that the behavior of each of the earlier languages lay within the scope of the newly supplied region. As we have already said, those

earlier languages were already incompatible with one other, and none was located at exactly the same point as basic CLOS. But, because of their essential similarity, we were able to delineate a coherent region of object-oriented languages that included them all. Users can select whichever language they prefer by adjusting default CLOS to the appropriate new point. Furthermore, they derive some additional benefits. Since the language behavior can be incrementally adjusted to any point in the region, not just to one or two pre-designated positions, users can gradually convert their programs from the old language to the new. And because different parts of a program can be assigned to different positions in the region, users can combine code written in different versions of the language within the same program.

The extensibility problem is solved in the same way. As long as the region includes the extended behavior the user wants, the default language can be simply adjusted to meet it. The efficiency problem can similarly be solved, so long as the user can readily alter the implementation strategy to suit each particular program or part thereof. In sum, if a region can be identified that is comprehensible enough for the user to understand and large enough to include the user's needs, and if it is easy for the user to incrementally adjust the default language behavior and implementation within that region, then our three goals can be met, and the investment in the basic language implementation preserved. The question remains, though, of how this can be done.

Metaobject Protocol Based Language Design

While designing a language—or language region—in this way departs significantly from traditional practice, it can be done while preserving the important qualities of existing design approaches. There are two critical enabling technologies: reflective techniques make it possible to open up a language's implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language's implementation and behavior to be locally and incrementally adjusted.

Reflective techniques [Smith 84, Maes&Nardi 88] allow the implementation to be exposed in a way that satisfies two important criteria. First, the access must be at an appropriately high level of abstraction, so that implementors retain enough freedom to exploit idiosyncrasies of their target platforms, and so that users aren't saddled with gratuitous (and non-portable) details. Second, that access must be effective, in the sense that adjustments must actually change the language behavior. These two properties are exactly what is provided by a reflective implementation model.[5]

[5] Reflection also solves any problems of circularity that arise when the language used to implement the protocol is the same as the language implemented by the prot,ocol. This merging of languages, which is often convenient, is adopted in CLOS, and the metaobject protocols presented in both Part I and Part IIcontain a number of such circularities. As in any reflective system, however, they can easily be discharged, as explained in Appendix C. It is important to note, however, that self-referentiality is not essential to the basic notion of a metaobject protocol. In on-going work that extends the ideas presented in this book, we are adding a metaobject protocol to Scheme, but we are using CLOS (not Scheme) as the language for

What reflection on its own doesn't provide, however, is flexibility, incrementality, or ease of use. This is where object-oriented techniques come into their own. These techniques work by (i) defining a set of object types and operations on them, which can support not just a single behavior, but a space or region of behaviors-this is commonly called a protocol; (ii) defining a default behavior, a single point in the region, in terms of the protocol-this is the role of the default classes and methods; and (iii) making it possible to effect incremental adjustments from the default behavior to other points in the region—this is the role of inheritance and specialization.

These techniques, applied to the design and implementation of a programming language itself, are exactly what our strategy requires. First, the basic elements of the programming language–classes, methods and generic functions-are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of metaobjects. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects-a metaobject protocol. Third, for each kind of metaobject, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol. In this way, metaobject protocols, by supplementing the base language, provide control over the language's behavior, and therefore provide the user with the ability to select any point within the region of languages around the one specified by the default classes. In the CLOS metaobject protocol, for example, the rules used to determine the implementation of instances are controlled by a small number of generic functions. This makes it possible to change those rules by defining a new kind of class, as a subclass of the default, and by giving it specialized methods on those generic functions. By doing this, the user is making an *incremental* adjustment in the language. Most aspects of both its behavior and implementation remain unchanged, with just the instance representation strategy being adjusted.

In this way, by combining these two techniques into an integrated protocol, we are able to meet a number of important design criteria:

- *Robustness:* moving the language around in the region to suit one program shouldn't have an adverse effect on other programs or on the system as a whole;
- *Abstraction:* in order to adjust the language, the user should not have to know the complete details of the language implementation;
- *Ease of use:* adjusting the language must be natural and straightforward, and the resulting languages must themselves be easy to use; and
- *Efficiency:* providing the flexibility of a surrounding region should not undermine the performance of the default language, nor curtail the implementor's ability to exploit idiosyncrasies of target architectures to improve performance of the entire region (in fact we retain our goal of having programs written in a language augmented with a metaobject protocol be more, not less efficient than programs written in a traditional language).

---

expressing adjustments, so issues of self-reference don't arise. The two criteria discussed in the main text remain of central importance, and reflective techniques can still be used to support them.

Our conclusion is that a synthetic combination of object-oriented and reflective techniques, applied under existing software engineering considerations, make possible a new approach to programming language design, one that meets a wider set of design criteria than have been met before. Doing so is the art of metaobject protocol design, the subject of this book.

Structure of the Book

The remainder of the book is divided in two parts. The first part presents metaobject protocol design. The second part gives a detailed specification of a metaobject protocol for CLOS.

In Part I, metaobject protocol design is presented as a narrated derivation of a metaobject protocol for CLOS. We begin with a (simplified) CLOS sans metaobject protocol, and gradually derive one for it. The derivation is driven by examples of the kinds of problems metaobject protocols can solve. This approach allows us to give attention not just to how metaobject protocols work and are implemented, but also to the process of analyzing user needs, and of incorporating those needs into the design of a protocol. In effect, metaobject protocol design requires determining the size, shape, and dimensions of the region to be provided. In the early stages of the derivation, the focus is on the basic motivation ani approach of metaobject protocol design. In later stages, attention shifts to problems of ease of use and efficiency.

Throughout, we will work with actual code for a simplified implementation of CLOS, and as we develop it, its metaobject protocol. This will give the reader an opportunity to gain some practical experience with the evolving design. In the same vein, we have included a number of exercises, addressing important concerns and open issues—we encourage all readers at least to read them, if not actually to work them through. Solutions to those that can be answered with code are included in Appendix B; others require more discursive replies, from short essays to moderate sized term projects.

The presentation throughout this first part presumes a familiarity with Common Lisp and CLOS. Readers who are unfamiliar with CLOS, but familiar with other object-oriented languages, will find an introduction in Appendix A. Those who are not familiar with object-oriented programming can find an excellent introduction to both it and CLOS in [Keene 89].

Chapter 1 lays the groundwork by presenting a simplified subset of CLOS and a simple implementation of it. This CLOS subset and implementation will be the basis of all of Part I, so we encourage even those familiar with CLOS to read this chapter.

Chapter 2 begins the derivation of the metaobject protocol by looking at the needs of users writing browsers and other program analysis tools. We will develop a variety of *introspective* protocols, which make it possible to analyze the structure and definition of a program.

Chapter 3 continues the derivation of the metaobject protocol with examples of compatibility, extensibility and performance needs that require adjusting the default language behavior. This ability to "step in" or intercede in the behavior of the system will be provided by a set of *intercessory* protocols.

In Chapter 4, we continue to develop intercessory protocols. But, in this chapter, our focus is the problem of designing protocols that are efficient and easy to use. We discuss various protocol design considerations and techniques.

Part II presents a detailed specification of a metaobject protocol for CLOS. The specification is divided into two chapters, in a manner similar to the specification of CLOS itself [X3J13]. Chapter 5 presents basic terminology and concepts; Chapter 6 describes each function, generic function, and method in the protocol.

Readers interested in designing metaobject protocols for other languages will find that this part not only fills in specific technical details, but also conveys additional information about the overall nature of our design approach. For CLOS users and implementors, however, the primary interest of Part II will be as a specification of a complete metaobject protocol for CLOS. This protocol is not offered as a proposed standard, but as a basis for experimentation, which may subsequently lead to the development of a revised and standardized CLOS metaobject protocol. There is evidence that this is already underway; many Common Lisp vendors are already implementing metaobject protocols based on the one presented here.

I THE DESIGN AND IMPLEMENTATION OF METAOBJECT PROTOCOLS

1 How CLOS is Implemented

We will present metaobject protocol design in stages, by following the development of a simplified metaobject protocol for a subset of CLOS. Our technique will be to progressively design and implement a metaobject protocol for this language, motivating each step with an example of the kind of problem users have with CLOS. Each example will be resolved by showing how it is handled by the newly developed portion of the protocol.

Metaobject protocol design requires an understanding both of the language behavior (in this case CLOS), and of the common architecture of that language's implementations. The first task therefore, addressed in this chapter, is to present the architecture of CLOS implementations in the form of Closette, a simple CLOS interpreter.[1] Closette is the "everyman" of CLOS implementations—despite the simplifications, it is representative of the architecture of all CLOS implementations.[2]

Although understanding implementation architecture is important, it is vital to distinguish the implementation from the documented language. A useful metaphor for making this separation comes from the theatre. We can think of the documented language as being *on-stage.* Users, which we think of as the *audience,* only get to see this on-stage behavior. The internal parts of the implementation are *backstage:* they support what happens on-stage, but the audience doesn't get to see them. Finally, implementors are the producers: they get to see what happens both on and offstage, and they are the ones responsible for putting on the show.

In presenting Closette, we will be showing the essential structure of what can be found backstage in any CLOS implementation. We will see how this backstage structure supports the on-stage language behavior. This will be useful in later chapters as we design the metaobject protocol, because it will let us think about what information waiting in the wings might be useful to the user, and what possibilities there are for implementors to expose that information.

Throughout this part of the book, presentations are based on working code. This will allow readers to try the examples, work through the exercises, and try alternative approaches. In fact, we recommend this (see Appendix D for the complete code).

---

[1] To support the later addition of a metaobject protocol, we have structured Closette as an object-oriented program. In this case, the object-oriented language is CLOS itself. (We assume that the reader is comfortable enough with the idea of metacircular interpreters to trust that the manifest circularities can eventually be resolved; we defer discussion of these circularities until Appendix C.)

[2] The simplifications in Closette are for pedagogical purposes only. The most significant is that it is an interpreter rather than compiler-based implementation-that is, we have reduced complexity by neglecting performance. In addition, most error-checking code has been omitted.

It will be assumed that the reader is familiar with Lisp and has some familiarity with CLOS programming. Those with a background in other object-oriented programming languages, such as Smalltalk or C++, can acquaint themselves with CLOS by reading Appendix A. Those who are not familiar with object-oriented programming can find an excellent introduction to both it and CLOS in [Keene 89].

1.1 A Subset of CLOS

In the interests of pedagogy and (relative) brevity, we have chosen to work with a simplified subset of CLOS. All the essential features of full CLOS are included: *classes,* which inherit structure and behavior from one or more other classes; *instances of classes,* which are created, initialized, and manipulated; *generic functions,* whose behavior depends on the classes of the arguments supplied to them; and *methods* which define the class-specific behavior and operations of generic functions. The major restrictions of the simplified dialect include:

No class redefinition. Full CLOS allows the definition of a class to be changed; the changes are propagated to its subclasses and to extant instances. The subset does not allow classes to be redefined.

No method redefinition. Full CLOS allows methods to be redefined, with the new definition completely replacing the old one. The subset does not allow methods to be redefined. (For convenience, the working code in Appendix D does support method redefinition.)

No forward-referenced superclasses. Full CLOS allows classes to be referenced before they are defined. One class can be defined in terms of another before the second has been defined. These forward references are not permitted in the subset. Explicit generic function definitions. Full CLOS allows the definition of a generic function to be inferred from the method definitions. The subset requires that a generic function be explicitly introduced with a defgeneric form before any methods are defined on it.

Standard method combination only. Full CLOS provides a powerful mechanism for user control of method combination. The subset defines only simple "demon" combination (primary, before-, and after-methods).

No `eql` specializers. Full CLOS allows methods to be specialized not only to classes, but also to individual objects. The subset restricts method specialization to classes.

No slots with `:class` allocation. Full CLOS supports slots allocated in each instance of a class and slots which are shared across all of them. The subset defines only per-instance slots.

Types and classes not fully integrated. Full CLOS closely integrates Common Lisp types and CLOS classes. It is possible to define methods specialized to primitive classes (e.g., `symbol`) and structure classes (defined with `defstruct`). The subset provides classes for the primitive Common Lisp types but not for structure classes.

Minimal syntactic sugar. A number of convenience macros and special forms are not included in the subset. These include: `with-slots`, `generic-function`, `generic-flet` and `generic-labels`.

1.2 The Basic Backstage Structures

In its simplest terms, a CLOS program consists of `defclass`, `defgeneric`, and `defmethod` forms mixed in with other more traditional Common Lisp forms. Executing these forms defines the program's classes, generic functions and methods.

Backstage, execution of these forms creates internal representations of the classes, generic functions, and methods, recording the information provided in their definitions. The implementation uses the information stored in the internal representation of a class to create instances of that class and to access their slots. Information stored in the internal representation of a generic function and its methods is used to invoke the generic function.

To make things concrete, consider the following example CLOS program:

```
(defclass rectangle ()
  ((height :initform 0.0 :initarg :height)
   (width :initform 0.0 :initarg :width)))

(defclass color-mixin ()
  ((cyan    :initform a :initarg :cyan)
   (magenta :initform a :initarg :magenta)
   (yellow  :initform a :initarg :yellow)))

(defclass color-rectangle (color-mixin rectangle)
  ((clearp :initform (y-or-n-p "But is it transparent?")
           :initarg :clearp :accessor clearp)))

(defgeneric paint (x))

(defmethod paint ((x rectangle))                ;Method #1
  (vertical-stroke (slot-value x 'height)
                   (slot-value x 'width)))

(defmethod paint :before ((x color-mixin))      ;Method #2
  (set-brush-color (slot-value x 'cyan)
                   (slot-value x 'magenta)
                   (slot-value x 'yellow)))
```

```
(defmethod paint ((x color-rectangle))        ;Method #3
  (unless (clearp x) (call-next-method)))

(setq door
      (make-instance 'color-rectangle
        :width 38 :height 84 :cyan 60 :yellow 55 :clearp nil))
```

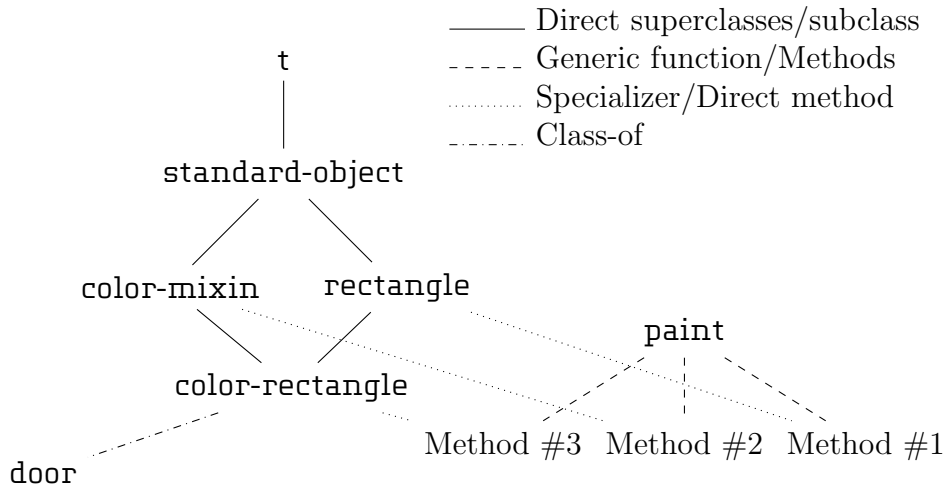

Figure 1.1. A First Gilmpse Backstage.

These definitions cause several new internal objects to be created and connected as shown in Figure 1.1. They can be divided into two groups: classes and instances, and functions and methods. In the former group, each class object is connected to its direct (i.e., immediate) superclasses and subclasses. The instance `door` (which is the only on-stage object in the figure) is connected to its class, but not vice versa. In the latter group, the generic function `paint` is connected to its methods, and conversely the methods are connected to their generic function. Connecting the two groups are bidirectional links between methods and the classes they are specialized to.

Unlike instances of `color-rectangle`, such as the one `door` is bound to, which represent rectangles, the backstage objects, such as the one corresponding to the class `rectangle`, represent elements of the program. These backstage objects are called metaobjects because they represent the program rather than the program's domain. In fact, everything that goes on inside the implementation is considered to be at the "meta" level with respect to the program; i.e., *about* the program itself, rather than about whatever the program happens to be about. Of course, if you weren't peering into the insides of an implementation you might not even notice that there were such metaobjects being backstage they are normally hidden from the user.

Name:	`color-rectangle`
Direct superclasses:	`(color-mixinrectangle)`
Direct slots:	{`clearp`}
Class precedence list:	`(color-rectangle color-mixin rectangle standard-object t)`
Effective slots:	{`clearp`, `cyan`, `height`, `magenta`, `width`, `yellow`}
Direct subclasses:	none
Direct methods:	{`paint` method #3}

Figure 1.2. The metaobject for the class `color-rectangle`.

CLOS implementations divide the execution of defining forms (`defclass`, `defgeneric`, and `defmethod`) and the processing of metaobjects into a three layer structure—somewhat reminiscent of furniture construction:

- The macro-expansion layer that provides a thin veneer of syntactic sugar that the user gets to seej e.g., the `defclass` macro.
- The glue layer that maps names to the metaobjects; e.g., the function `find-class`, which looks up a class metaobject given its name.
- The lowest layer that provides all the support, and traffics directly in first-class metaobjects. This is where the behavior of classes, instances, generic-functions, and methods is implemented. (Our metaobject protocols will end up being concentrated in this layer.)

Given this overall picture, the remainder of the chapter will fill in the details by working through Closette, dealing in turn with the following issues:

- How classes are represented. (Section 1.3)
- How objects are printed. (Section **??**)
- How instances are represented, initialized, and accessed. (Section **??**)
- How generic functions are represented. (Section **??**)
- How methods are represented. (Section **??**)
- What happens when a generic function is called. (Section **??**)

1.3 Representing Classes

The CLOS user defines classes with the `defclass` macro. It is natural, therefore, to start by examining how `defclass` is implemented. Although a lot of machinery will be introduced, remember that the definitions other than the `defclass` macro are internal to the implementation—they are hidden backstage.

The term *class metaobject* is used for the backstage structure that represents the classes the user defines with `defclass`. For example, the class metaobject corresponding to the class named `color-rectangle` contains the information shown in Figure 1.2. In general, this information will include:

- Fields from the `defclass` form; e.g., the class's name (`color-rectangle`), direct super classes (`color-mixin` and `rectangle`), and slot specifications ({`clearp`...}).
- Information that is derived or inherited; e.g., a list of all of the class's superclasses in order of precedence and the full set of slots including those inherited from superclasses.
- Backlinks to the class's direct subclasses and links to methods that include the class among the method's specializers.

As our first step in the construction of Closette, the class `standard-class` is defined to centralize the description of what class metaobjects look like. Instances of `standard-class` represent individual classes; or, to say it another way, class metaobjects are instances of `standard-class`. Here is the definition:

```
(defclass standard-class ()
  ((name :initarg :name
         :accessor class-name)
   (direct-superclasses :initarg :direct-superclasses
                        :accessor class-direct-superclasses)
   (direct-slots :accessor class-direct-slots)
   (class-precedence-list :accessor class-precedence-list)
   (effective-slots :accessor class-slots)
   (direct-subclasses :initform ()
                      :accessor class-direct-subclasses)
   (direct-methods :initform ()
                   :accessor class-direct-methods)))
```

Note that throughout this part of the book, all code that is part of the Closette implementation will be marked with a backstage door this way.

Each of the slots has been given accessor functions; these will be used consistently to access the slots of metaobjects (rather than employing `slot-value`).

1.3.1 The `defclass` Macro

In Closette's three layered implementation structure, the job of the defclass macro (macro-expansion layer) is to parse the class definition and convert it into a call to `ensure-class` (glue layer).

The implementation of `defclass` is organized so that the bulk of the macro-expansion work is carried out by `canonicalize-`... procedures:

```
(defmacro defclass (name direct-superclasses direct-slots &rest options)
  `(ensure-class' ,name
    :direct-superclasses ,(canonicalize-direct-superclasses
                           direct-superclasses)
    :direct-slots ,(canonicalize-direct-slots
                    direct-slots)
```

```
,@(canonicalize-defclass-options options)))
```

1.3.2 Direct Superclasses

Bibliography

[Barstow et al. 84] Barstow, David, Howard Shrobe, and Eric Sandewall (eds.) *Interactive Programming Environments,* McGraw-Hill, New York, 1984.

[Bobrow et al. 87] Bobrow, Daniel G., David S. Fogelsong, and Mark S. Miller "Definition Groups: Making Sources Into First-Class Objects," in Bruce Shriver and Peter Wegner (eds.) Research Directions in Object-Oriented Programming, MIT Press, 1987, 129-46.

[Bobrow&Stefik 83] Bobrow, Daniel G. and Mark Stefik *The Loops Manual,* Intelligent Systems Laboratory, Xerox PARC, 1983.

[Cannon 82] Cannon, Howard I. "Flavors: A Non-Hierarchical Approach to Object-Oriented Programming," 1982.

[CLtL] Steele, Guy *Common Lisp: The Language,* Digital Press, 1984.

[CLtLII] Steele, Guy *Common Lisp: The Language, Second Edition,* Digital Press, 1990.

[Keene 89] Keene, Sonya E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS,* Addison-Wesley, 1989.

[Maes&Nardi 88] Maes, Pattie and Daniele Nardi (eds.) *Meta-Level Architectures and Reflection,* North-Holland, 1988.

[Smith 84] Smith, Brian C. "Reflection and Semantics in LISP," Proceedings 11^{th} Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, January 1984, 23-35.

[X3J13] Bobrow, Daniel G. Linda G. Demichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon *Common Lisp Object System Specification,* X3J13 Document 88-002R, June 1988; appears in Lisp and Symbolic Computation 1, 3/4, January 1989, 245-394, and as Chapter 28 of [CLtLII], 770-864.