Open Implementations and Metaobject Protocols

Gregor Kiczales, Andreas Paepcke

Notes from an earlier tutorial that shows how to use OO programming techniques to support Open Implementation of programming languages. Many of the concepts in here apply to other kinds of open implementation as well.

NOTE: THIS DOCUMENT CONTAINS WORK IN PROGRESS.

Open Implementations

and Metaobject Protocols

Gregor Kiczales

Andreas Paepcke

Contents

Chapter 1 Introduction

Executive Summary

A Simple Premise

Substrate systems—like programming languages, operating systems and object systems—that are flexible and tailorable are more useful than ones that are not.

Being open to programmer inspection and adjustment means they can meet a far wider range of needs.

This means applications will be less complex, because programmers can fix some of the problems themselves.

The work presented in this book is based on the observation that much of the complexity in current application programs comes from the application programmer having to ‘code around’ deficiencies in the underlying operating system, programming language, object system or other substrate software. This happens when the application programmer would like a specific feature or behavior[1] that the substrate does not provide.

1. Throughout, we use the term 'behavior', instead of 'semantics', to refer to the “computation” or activity that results from executing a program. There are three reasons for this choice. First, as a word of ordinary English, 'behavior' is more accurate and descriptive of what we have in mind. Second, the term 'semantics' has quite different connotations in some fields adjacent to computer science, such as artificial intelligence and cognitive science, which can lead to confusion. Third, in philosophy and logic, 'semantics' is primarily used to name a relatively abstract relationship between a symbol system and the (usually external) domain that its symbols are <about>. In the case of an ordinary program, such as an elevator controller, this would include the (non-effective) relation between the data structure representing the number of people on the elevator, for example, and the actual people taking the ride. Similarly, in an object system, a philosopher would be most likely to use the term “semantics” to refer to the relation between the objects inside the system, such as an object for flight-133 in an airline reservation system, and the real-world flight to which that object was intended to correspond. Since we believe that computer science would do well to pay more direct attention to abstract relations of this sort -- i.e., to relations between particular programs and their real-world subject domains -- we are at least tempted to use the term 'semantics' broadly enough to include such relations within its scope.

But this book does not propose that substrate software should just "have more features to please more programmers." Instead, it suggests that substrates should be open in a way that allows programmers access to and control over the substrate's implementation in a way that allows the programmer to tailor the substrate to the needs of a particular application. This is called *open implementation*.

Naming conventions.[1]

1.

1. Noun-noun compounds should be styled open when their elements are stressed independently; they may be styled closed only when stressed as a single word (as with "database", "mastermind", "dogsled"; contrast "data flow", "master key", and "dog biscuit"'.)
Open compounds are hyphenated when used as modifiers before a noun:

master-key impressions data-flow diagram
dog-biscuit packaging base-interface specifications

2. Words formed with the prefixes pre-, intra-, extra-, super-, sub-, ultra-, meta-, etc. are usually spelled solid, except in the following cases:
a. Second element capitalized or a figure: anti-Semitic, pre-1945
b. Second element more than one word: non-English-speaking
c. To distinguish homonyms: un-ionized, re-present
d. Optionally, when both the prefix and a noun stem bear independent stress (anti-art, anti-collision), particularly when the closed form would create a distracting sequence of vowels (as in meta-analysis, supra-auricular, meso-appendix, meta-arthritic; but either meta-igneous or metaigneous)

Topics Addressed (1/4)

Won't that make substrate systems

- harder to design?
- harder to implement?
- more prone to misuse?

These problems can be overcome, and the reduced complexity of applications makes it worthwhile.

While many would agree that inflexible substrate software does cause problems for application programmers, the thought of open implementation raises concerns, especially since it seems that the design and implementation of closed substrates is already difficult enough. Questions arise such as: Won't substrates with open implementations require designers to anticipate all the possible extensions a programmer could want? Does open implementation violate modularity and encapsulation? How will the implementor be able to make an open implementation robust? Will this be such a powerful tool that application programmers will be able to "machine gun themselves in the foot?"

This book shows how these concerns can be addressed, and how the reduction in application complexity more than makes up for the additional care required in substrate design.

Topics Addressed (2/4)

Processes and technologies supporting the design and use of open implementations of programming languages, object systems, operating systems, databases and other substrate software.

One reason why the effort is worthwhile is that each substrate tends to support many applications. In addition, the techniques introduced here will be applicable to a wide variety of substrate systems.

Topics Addressed (3/4)

But isn't that just OOP?

OOP is one technology that helps. But more is involved:

- A methodology
- Metaobject protocols: one way of using OOP
- Non-OOP techniques: partial evaluation, reflection...

But isn't an object-oriented implementation language all you need for this? Well, not quite. Several other aspects are indispensable as well, and they will be addressed:

- A design process and a way of thinking about the design of substrates with more flexible implementations.
- An effective strategy for partitioning the substrate functionality, for organizing the interactions between the system components, and for enabling intervention without making substrate usage cumbersome and slow.
- A variety of supporting technologies that are independent of object-oriented programming.

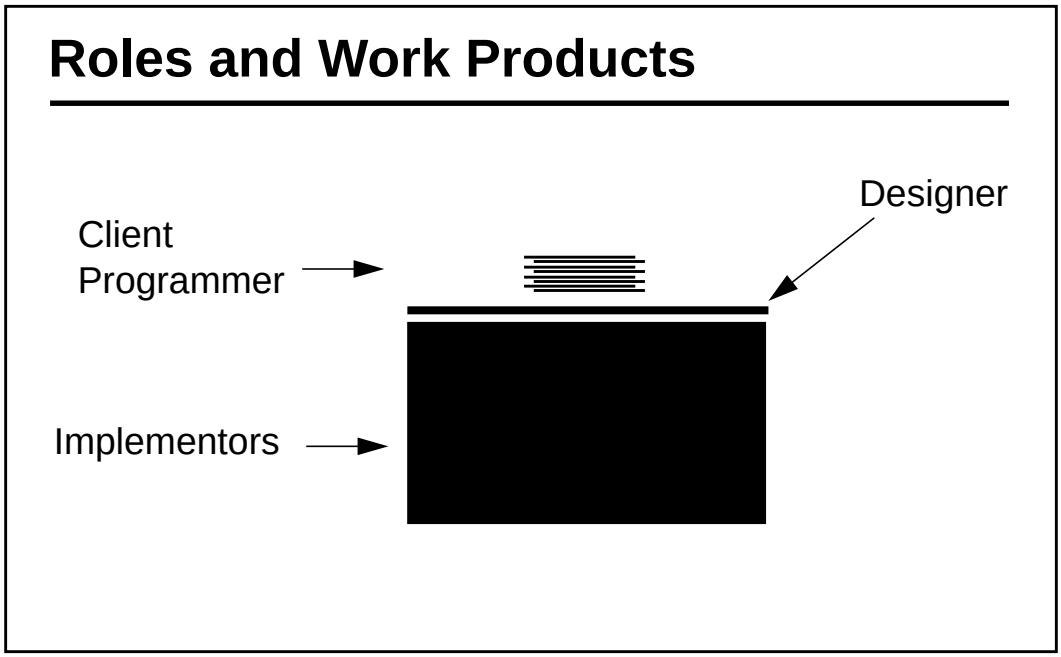
Topics Addressed (4/4)

The primary example will be a simple object system. The collection of techniques will be used to open its implementation and make it more flexible for a client programmer.

But the intuitions, processes and techniques that will be discussed are applicable in other areas, such as operating systems, window systems, databases and other kinds of software systems.

The most detailed parts of this book use an object system as an example substrate to make tailorable. In this case, that object system happens to be bound to a C-like language, but that is coincidental. It might instead be used to serve as the data model of an object-oriented database. The language as such is not of concern here. Instead, the focus will be on the object system: its classes, methods, inheritance behavior, etc.

Example substrates other than object systems, such as operating systems or distributed computing, will be mentioned occasionally, and the reader is invited to draw parallels to them as the tutorial moves along. The technologies that will be introduced are not limited to object systems, but address issues of reusability and flexibility in general.



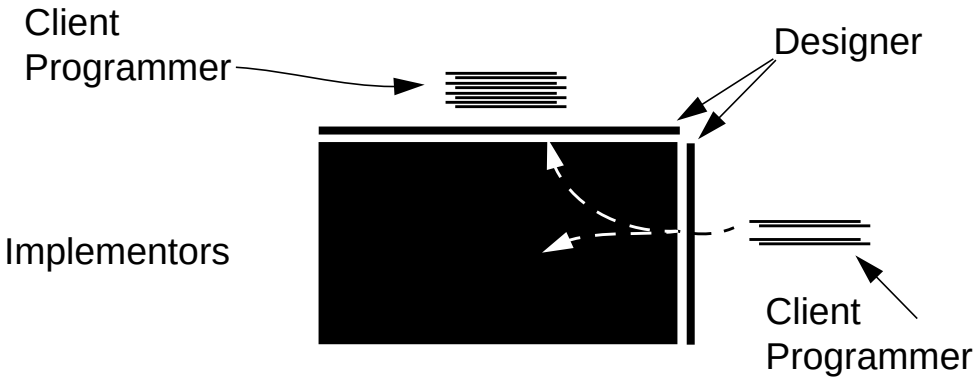
The open implementations approach involves a reapportioning of responsibility among the parties involved with substrate software: the designer, the implementors, and the programmer who builds on top of it.

As such, a number of issues will be on the table for examination, including the roles of the designer, implementors and programmer, as well as their work products, the documented design, the implementation, and the programs that use the substrate system.

The figure shows a traditional black-box substrate with an interface for client programmers to program to—with all the limitations the interface design and the particular implementation places upon him.

As the story of open implementation unfolds in this book, a second interface will be added on the side of the black box.

Increased Programmer Access



A new kind of interface allows the client programmer to access and modify implementation and design.

It is specified by the same designer who specifies the traditional, base level interface. The client programmer still programs mainly to the base interface, but he can also write separate programs to the new interface to make the substrate implementation or its base level interface more suitable to his particular needs. This in turn simplifies his programs.

The client programmer thereby gains some of the power and responsibility traditionally reserved for the designer and implementors. The challenge to this approach is that the substrate needs to stay robust and efficient.

Separation of concerns?

Getting Down to Work

Every object system is inadequate somewhere.

As you read, try to think about your own experiences and frustrations.

As a preview of the issues the book addresses, the following pages present examples of the kinds of problems programmers currently have with closed object systems. These examples are presented in the form of a simplified dialog between a programmer on the one hand, and the implementors on the other. This kind of dialog is a driving force throughout the book.

Programmer Questions (1/4)

…and the Responses They Get

Dear Implementors,

How can I find out which classes inherit from two given classes? Or which method will run in a given case? Or which classes define a given slot name?

Dear Programmer,

A special feature of our implementation is that it includes a nice graphical browser which can be used to explore these kinds of questions.

Here is an example where the programmer simply wants access to...

This is a common enough request in the domain of object systems that many implementations include some built-in tools for exploring these questions. But as this example suggests, unless these facilities are part of the documented standard, any application programmer use of them will not be portable across implementations. Typically, this functionality is left out because it is too complex to burden all implementors with.

Programmer Questions (2/4)

…and the Responses They Get

Dear Implementors,

What is the name of the procedure for copying objects? I can't find it in the manual.

Dear Programmer,
There is no copy procedure because the appropriate type of copying is often context-sensitive. Sometimes a shallow copy is needed, sometimes a deep copy. Often a combination is the right choice. Because there is no 'one right way' to copy objects, the designers chose not to include such a mechanism.

In this example, the programmer is asking for a simple feature which he is surprised to discover has not been provided. He learns that the object system designer did not include it because there are too many different "reasonable" behaviors for copying objects.

This is particularly frustrating since most programmers have a particular behavior in mind, but they find it awkward or impossible to implement without access to implementation internals. Without an open implementation, the object system implementors can either arbitrarily decide to use one of the copy semantics, thereby making the system more amenable to a few users, but invoking a false sense of generality, or they can leave the functionality out altogether, making the system equally unpleasant for all users.

Notice that the programmer's question is directed at the implementors, even though it is more about the design than the implementation. This happens in practice because most programmers do not have access to the designers of the systems they use—only the vendor or, at best, the implementors do.

Programmer Questions (3/4)

…and the Responses They Get

Dear Implementors,

Is there a hook or callback that is always called when a slot is altered? I would like to have one slot in my objects be updated whenever any of the other slots is set. I plan to use this to implement a history mechanism.

Dear Programmer,

There is no such feature in the system. It would be hard to implement, so it probably isn't really what you want.

This is another example where the programmer wants an additional feature, although arguably a trickier one than the copy functionality. But because such powerful callbacks have been difficult to manage and implement efficiently, ***

Programmer Questions (4/4)

…and the Responses They Get

Dear Fellow Programmers,

Is there an implementation of TinyObjects that has an option that makes it do Eiffel-style inheritance?

Dear Programmer,

No. Even if they did, using such an implementation-specific feature would make your code non-portable. It probably isn't what you want.

This one ***

This brings up the question of whether this person should simply use Eiffel. But that is not a solution, because even though there might be a single major mismatch of a system to one's needs, a wholesale switch to an alternative is not necessarily the best, or even a possible answer.

Client Programmer Frustration

Functionality
- missing
- wrong

Performance
- space
- time

***Need example for performance tuning.

If a programmer likes nothing about a substrate, he should just use another alternative. But what the substrate designer (and even more often the vendors) hears is “Gee, I like most of your system. But there is this one thing that is really mismatched to my needs.”

These examples showed several categories of changes to implementations that programmers tend to ask for. More examples will come up in the chapters ahead and will categorized to a much finer granularity.

The Designer's Dilemma

The designer of closed implementations must choose functionality that is...

- general-purpose enough,
- clean enough,
- and efficient enough...

...to go into the system that **all users must share**.

The answers often make sense from the designers' and implementors' points of view if their systems are monolithically constructed and are used by a wide audience for many purposes.

The problem is that if the application programmer really needs a piece of functionality, he will implement it somehow. The result are user programs cluttered with partial, often sub-optimal solutions that are replicated in variations throughout other applications.

The Implementors' Dilemma

The implementors of closed implementations must choose implementation strategies that are...

- general-purpose enough,
- clean enough,
- and efficient enough...

...to go into the system that **all users must share**.

The answers often make sense from the designers' and implementors' points of view if their systems are monolithically constructed and are used by a wide audience for many purposes.

The problem is that if the application programmer really needs a piece of functionality, he will implement it somehow. The result are user programs cluttered with partial, often sub-optimal solutions that are replicated in variations throughout other applications.

A Long-Standing Tension...

Designers and implementors of closed substrates must choose:

General-purpose enough for all ⟷ Special needs of an individual

System tailorability fundamentally dissolves that tension. It allows substrates to be lean, yet able to meet special needs of even small classes of programs.

Restating the approach of this book in somewhat different terms: No person using a substrate is a "general-purpose" programmer. Many times application programmers have legitimate special needs.

Open Implementations

Dissolve the Tension...

...by giving the client programmer principled access to the implementation. This enables him to inspect, extend and modify the implementation to affect functionality or performance.

Given such access, the programmer will be able to solve many of his problems himself.

If the designer does a good job of making substrate system implementations open, the client programmer will often be able to solve his own problems by first adjusting the substrate to suit his needs. Then he can focus on his application, constructing it on top of the modified substrate that is exactly appropriate to him.[1]

This is the basic tenet of the material presented here. While this may sound obvious to some, it has not been pervasive theory and practice in computer science. This book must therefore show just what is meant by “principled”.

In particular, it needs to address concerns, such as these:

- Will tailorability result in a constantly changing interface?
- Can substrates still be safe?
- Can they be efficient, or will implementors be over-constrained?

A first step towards answering these questions will be to be more differentiated about what it means to have an open implementation.

1. *Open implementations* should really be called *open systems*. But that term was taken.

Three Kinds of Opening

- Introspection—access to implementation state
- Invocation—access to internal functionality
- Intercession—changes to behavior or implementation strategy to improve performance

Three kinds of implementation opening will be discussed, each serving a different purpose. The first, introspection, enables the client programmer to work with abstractions of selected implementation state in his programs. This helps answer questions such as the one about inheritance structure on page 11. The challenge will be to do this without compromising the benefits of encapsulation.

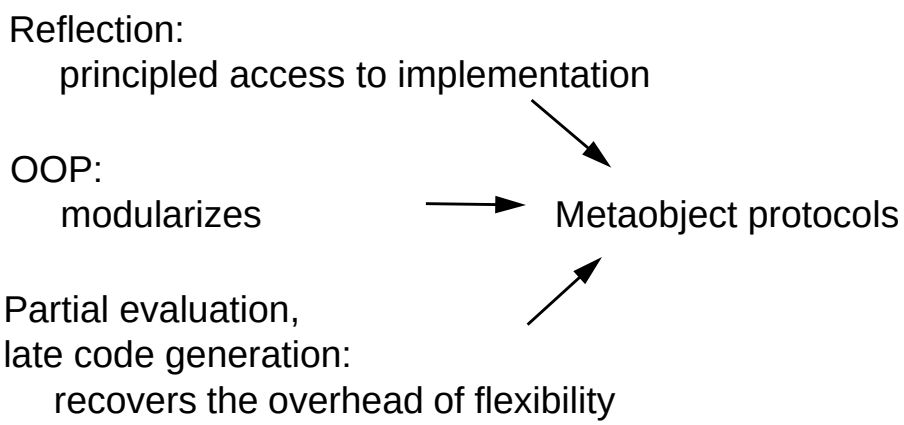
The second kind of opening is the ability for client programmers to invoke substrate functionality directly, without going through the official substrate interface. This is often useful when the interface is 'optimized' for cases other than what the client programmer needs.

The third kind of opening, finally, is to let programmers add or change substrate behavior, or to adjust implementation strategies to improve performance for particular applications. The missing copy functionality of page 12, or the callback of page 13 are examples for this.

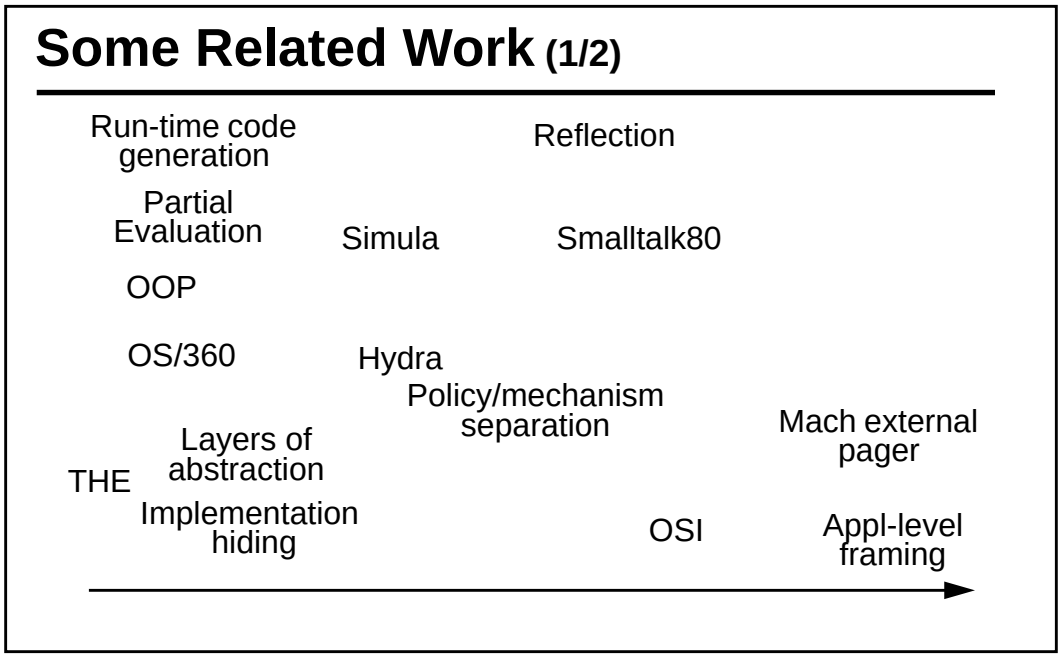
How will all this be accomplished? Fortunately, appropriate technologies exist today.

Metaobject Protocols

A Synthetic Idea to Achieve Open Implementations

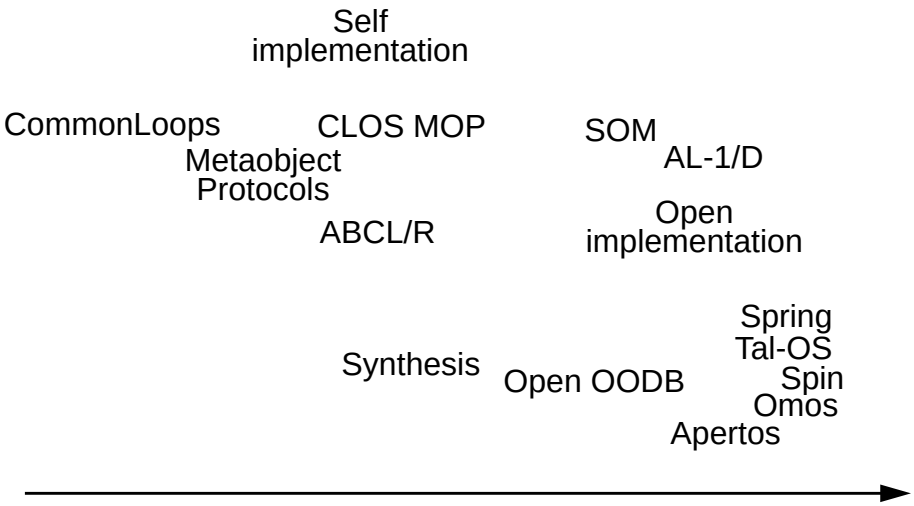


Metaobject protocols, the central tool used in this book, is a synthetic capability that draws from a variety of existing technologies. Each of these technologies is enriched by ongoing research. Except for the basics of object-oriented programming, all necessary material around these technologies will be introduced as it comes into play throughout the book.



There is a long history that led up to the idea of open implementations and metaobject protocols in particular. Important notions, such as run-time code generation, partial evaluation, object-oriented programming, layers of abstraction or implementation hiding spawned threads of development in programming languages, as well as in the networking and operating systems communities. This figure, together with the following one shows a very rough layout of the notions and example systems inspired by these early thoughts and associated prototypes. At the top, the language thread shows how object-oriented programming was followed by *Smalltalk-80* which already contained reflective elements in its metaclasses. In the operating systems area (lower part of the figure), *Hydra*'s policy / mechanism separation was later followed by the *Mach* operating system with its replaceable paging mechanism. Analogously for networking, ISO's open systems interconnection reference model (OSI) clearly followed an abstraction layering approach, and it has dominated thinking in the network community for a long time. But the network community has realized that layering has limitations, and the concepts of application-layer framing (of network data) and integrated layer processing (collapsing the layer implementation for specific purposes) are network equivalents of approaches exemplified for the area of object models in this book.

Some Related Work (2/2)

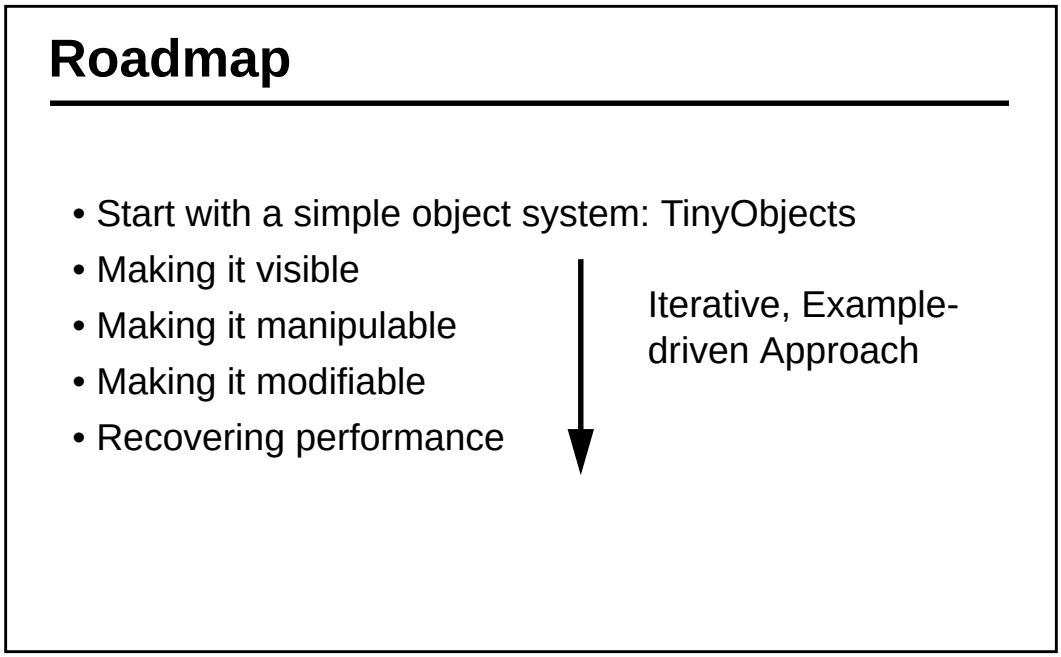


Continuing with the development of languages, metaobject protocols began to be used in *CLOS* to manage the complexities introduced by reflective architectures. The use of reflection, meanwhile, was broadened in *ABCL/R* to cover problems of distribution.

Similarly to the networking community, it became clear that abstraction brought inefficiencies which needed to be offset by advanced implementation techniques that reached back to the ideas of partial evaluation and run-time code generation. The implementation of the *Self* language is notable in this context, as are the efforts around the *Synthesis* operating system.

With *Open OODB*, the database community entered the search for open implementations as well. Open implementations, then, is the confluence of several threads in a long history, fed by intellectual insights (abstraction, implementation hiding, reflection), by technologies (partial evaluation, run-time code generation, object-oriented programming) and by a series of prototypes and products that attempted to bring all these together.

Please refer to the bibliography at the end of the book for relevant references.



So much for history and the context of this book. It tells the story of open implementations by using an object system as an example.

The following chapter presents the design of a traditional (black-box) object system called TinyObjects. It will be the example used throughout the book. Chapter 3 through Chapter 7 work through a series of programmer complaints, gradually opening the implementation to address them.

TinyObjects is a running system which will be introduced in Chapter 2[1]. It uses a C++-like syntax to make it easy to follow. But TinyObjects is not C++. This is because C++—or any other full-fledged language—would be more complex than desirable for teaching purposes. On the other hand, TinyObjects is complex enough that the important issues of a real-life system can be examined.

In most cases, real code will be used for the examples. The use of real code can have the disadvantage of being somewhat more complex and less to the point than pseudo code. The advantage, however, is that it can convey a sense of the actual effort involved in the various examples.

1. An emulator for it is written in C++.

In order to successfully introduce the concepts, the reader is asked to suspend questions of substrate performance until Chapter 7. That chapter is exclusively dedicated to the problem of regaining performance lost by the techniques introduced in earlier chapters.

Chapter 2

TinyObjects: A Simple Object System

This chapter introduces a simple object system called TinyObjects. It is not intended to be full-scale programming tool, but was instead designed to be a simple "everyman" of object-oriented programming facilities, useful for teaching purposes. There is nothing special about it, it is the common core of C++, Smalltalk, CLOS and Objective-C, wrapped up in a C-like syntax.

The remainder of the chapter presents TinyObjects in three ways. First, the next slide briefly summarizes the features of TinyObjects in the terminology to be used throughout the book. Second, the following four slides compare TinyObjects with other common object-oriented languages. Then, the remainder of this chapter presents the syntax and behavior of specific aspects of TinyObjects in more detail.

As TinyObjects is being presented, be critical of it. Think about how useful it would be to you. Is it missing features you think you might need? Do the features it has work the way you would like? Make note of the problems you discover. These will be useful later in designing, evaluating and improving the open implementation of TinyObjects that subsequent chapters will develop.

TinyObjects

A Very Simple Object Model

- Class/instance model (not prototypes)
- Objects have slots
- Generic functions and methods
- CallNextMethod
- Multiple inheritance
- Dynamic type checking and no static typing
- Everything is a pointer
- Automatic Garbage Collection

TinyObjects has a class / instance model, meaning that the programmer first defines classes, and can then create objects that are instances of those classes. The class specifies the slots it's instances possess (instance variables in Smalltalk, data members in C++).

Generic functions are TinyObjects' equivalent of "messages." Methods provide the code that implements generic functions for specific classes. Methods specialized to subclasses override those for superclasses, but a method can use **`callNextMethod`** to run the superclass method. TinyObjects supports multiple inheritance.

For simplicity, "everything is a pointer" in TinyObjects. This means there is no need to deal with de-referencing pointers or finding object addresses. All parameter passing is by reference and all return values are pointers. TinyObjects also includes an automatic garbage collector, since that helps simplify the code we show.

Finally, because TinyObjects is embedded in C it includes a number of basic C features, such as variables, assignment, conditionals and the like.

Differences from C++

- Much simpler
- Dynamic type checking
- Everything is a pointer
- Dynamic dispatch only (virtuals)
- No public/private
- Automatic disambiguation of name conflicts
- Automatic garbage collection

C+: comparison with TinyObjects:TinyObjects; C++, comparison

As is the case in languages like Lisp and Smalltalk, all values manipulated by TinyObjects programs are dynamically typed objects. C++ programs, in contrast, manipulate both object and non-object values, and rely primarily on static typing.

While C++ distinguishes between virtual and non-virtual functions, TinyObjects supports only what it calls *generic functions*, which have functionality similar to virtual functions, but which are called using ordinary C function call syntax.

The C++ facilities for enforced encapsulation are missing from TinyObjects. Protection of data members (slots) is by convention only. Programmers are expected to read and write slots only through accessor generic functions.

Another difference has to do with how TinyObjects resolves like-named slots and like-named methods when classes inherit from each other. Whereas C++ requires programmers to explicitly specify which class's data, or function member they wish to access, TinyObjects automatically merges like-named slots (see page 40) and automatically orders like-named methods (see page 50). (This aspect will be revisited in this book.)

Differences from Smalltalk

- Multiple inheritance
- Object initialization mechanism is different
- No meta-classes (to begin with)
- Function syntax instead of messages

Although TinyObjects is a compiled language with mostly C-like syntax and control structures, in many respects it is similar to Smalltalk. One big difference is support for multiple inheritance, with conflicts resolved by a CLOS-like class precedence mechanism.

In addition, TinyObjects has a general scheme for automatically initializing objects as they are created.

Another big difference is that whereas Smalltalk includes some meta-object protocol functionality, TinyObjects, as introduced in this chapter, includes no metaobject protocol. (That will be added in later chapters.)

TinyObjects:Smalltalk, comparison;Smalltalk:comparison with TinyObjects

Differences from CLOS

- Much simpler
- Class specializers only
- No multimethods
- Object initialization
- No method combination
- No slot options
- Classes, methods and GFs are not first-class
- No redefinition (classes or methods)

Those who know CLOS will recognize that TinyObjects is much simpler, and missing many useful CLOS features.

TinyObjects supports only classical method dispatch, so it does not support multi-methods or **`eql`** specializers. It also has no declarative method combination. It does, however, have **`callNextMethod`**.

TinyObjects has only instance slots and no class slots. It does not have slot initializers, slot initargs, or automatically generated slot accessors.

Also, at least initially, TinyObjects is more C-like than CLOS-like, in that classes, generic functions and methods are not first class. Also, as mentioned in the comparison with Smalltalk, TinyObjects differs from CLOS in that it starts with no meta-object protocol.

CLOS:comparison with TinyObjects;TinyObjects:CLOS, comparison

Differences from Objective-C

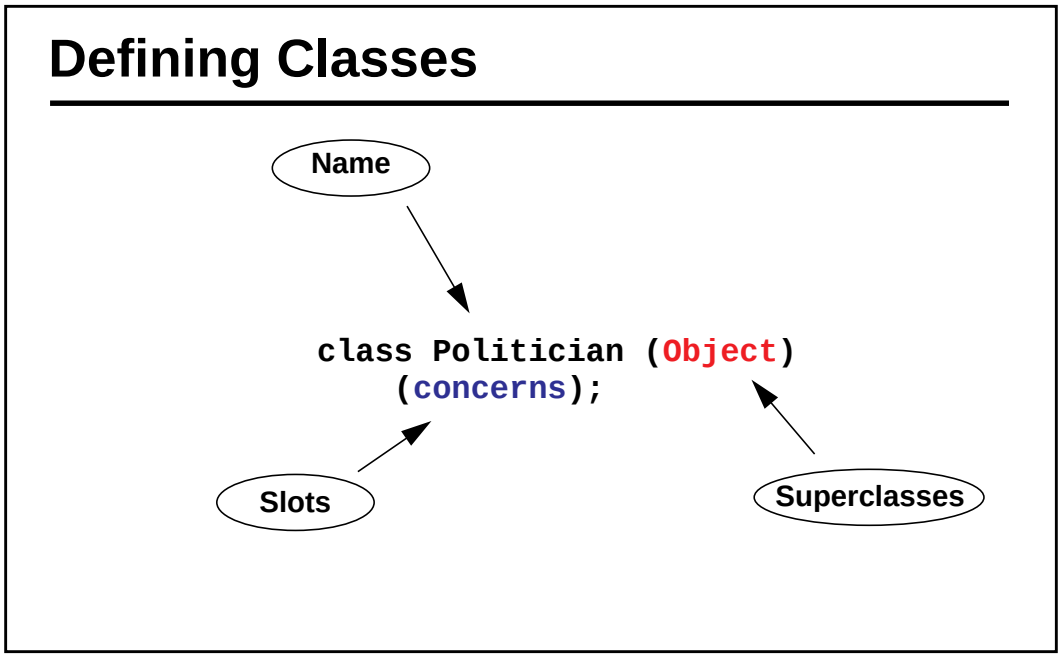
- Multiple inheritance
- Everything is an object
- Object initialization mechanism
- Function syntax, instead of messages

Like Objective-C, TinyObjects is a compiled language based on C, with a dynamically-typed object system. Unlike Objective-C, where objects are an addition to C's statically-typed values, TinyObjects' object system is a replacement for C's type system. All values are objects, and even built-in classes like integers and strings can have new methods defined on them.

TinyObjects uses ordinary function call syntax to invoke methods on an object, rather than Objective-C's special message send syntax. TinyObjects also supports multiple inheritance.

In addition, TinyObjects has a general scheme for automatically initializing objects as they are created.

TinyObjects:Objective-C, comparison;Objective-C:comparison with TinyObjects



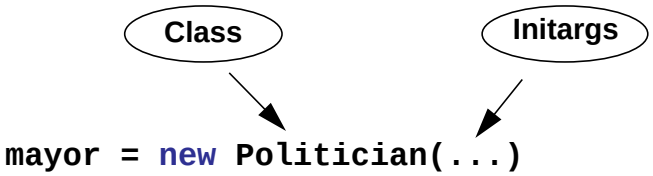
The remainder of the chapter presents the syntax and behavior of TinyObjects in more detail.

This slide shows how a new class is defined. The definition consists of four parts: The keyword **`class`**, the name of the new class, a list of superclasses and a list of slot names. Note the plural, superclasses—TinyObjects supports multiple inheritance. (Examples of defining a class with multiple superclasses will come up shortly.)

Throughout this book we will use a naming convention in which class names begin with an uppercase letter, and each subsequent word in the name is also capitalized. Function, variable and slot names will begin with a lowercase letter, and each subsequent word in the name will be capitalized.

Class:definition;Class:TinyObjects in; Multiple inheritance:specification; Slot:definition;Superclass:definition; TinyObjects:Class definition; Multiple inheritance:TinyObjects in; Subclass:definition

Making Objects

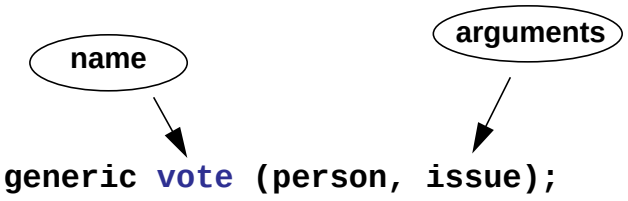


Objects are created using the **new** primitive. The syntax is the **new** keyword followed by the class name and a (possibly empty) list of initialization arguments. The **new** primitive returns (a pointer to) an initialized object. In this example, that new object is then bound to the variable **mayor**.

The syntax and processing of the initialization arguments—which will subsequently be abbreviated to *initargs*—is class-specific. More detail on this will be provided shortly.

TinyObjects:Object creation

Defining Generic Functions



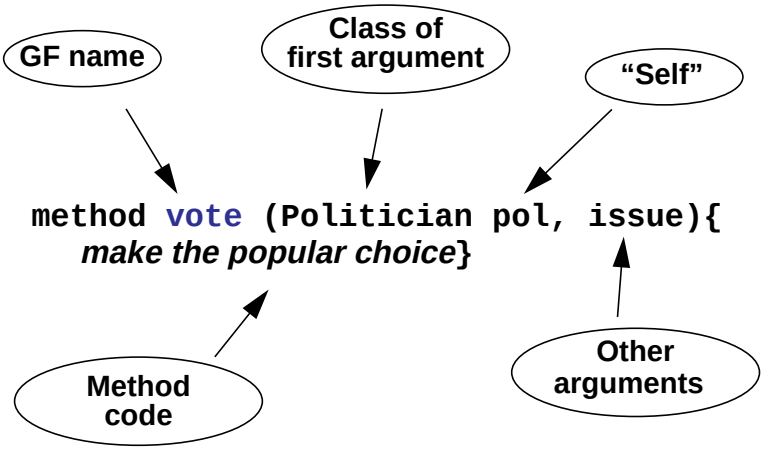
TinyObjects provides generic functions, which have functionality similar to that of C++ virtual functions or Smalltalk messages.

The figure shows the definition of a generic function that takes two arguments. The names of the arguments are in no way significant to TinyObjects, although good programming style dictates using meaningful names. Once methods are added, the class of the first argument (**`person`**) will determine the method that is run.

Generic function:definition;Generic function:TinyObjects in

TinyObjects:Generic function definition

Defining Methods



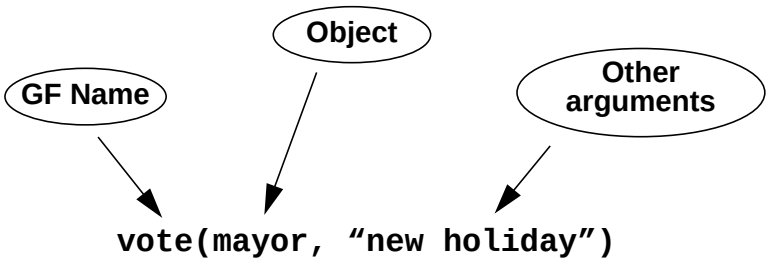
Methods provide the behavior of generic functions. A generic function can have multiple methods, specialized to specific classes. When a generic function is called, it checks the class of its first argument, and selects from among its methods the one that most specifically matches that class. (This will be covered in more detail in the remainder of the chapter.)

A method definition contains four components: the keyword **`method`**, the name of the generic function to which the method belongs, an argument list, and a body of code. The argument list must contain as many arguments as are specified in the generic function definition. The first argument is special in that the name of its class must also be specified. (This is the class to which the new method is specialized.)

The **`vote`** method above is said to be "specialized to the **`Politician`** class." Note that in TinyObjects the object to which the method is applied appears as an explicit parameter (the first), rather than being an implicit ("self") parameter as it is in some languages.

Method:definition;Method:TinyObjects in;TinyObjects:Method definition

Calling a Generic Function



Generic functions are called using normal C function call syntax. Method dispatch is driven by the first argument only (in this case **`mayor`**).

Assuming only the method and class definitions shown so far, this call to **`vote`** will run the method on the previous page. If on the other hand, **`vote`** were called on an object of the class **`Object`**, the only superclass of **`Politician`**, no method would be applicable and an error would be signalled.

Generic function:invocation;Method:invocation

Polymorphism

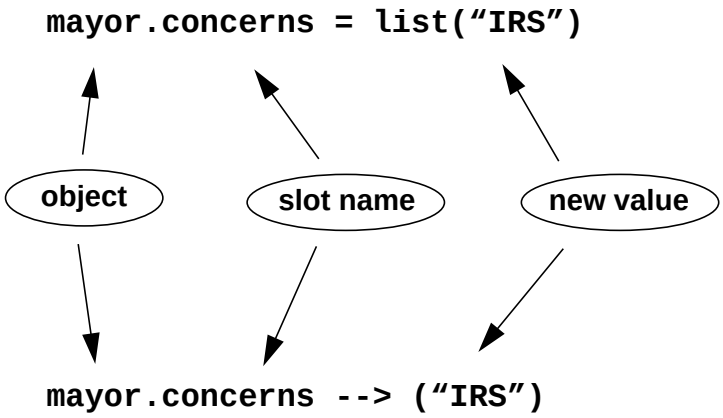
```
class Manager (Object) ();
method vote (Manager who, issue) {
      make cost-effective choice}

ceo = new Manager()



vote(mayor, “new holiday”) --> “YES”
vote(ceo,   “new holiday”) --> “NO”
```

This is an example where more than one method is defined on a single generic function. First, a new class of person, **Manager**, is defined. A method on **vote**, specialized to **Manager** is defined as well. The two calls to **vote** show the effect of having the two different methods—politicians seek to be popular with the workers, and managers seek to be popular with the stockholders.

At the bottom of the slide, as a convention in this book, an arrow “-->” indicates the return value of a procedure, function, or language primitive.

Method:invocation

Accessing Slots



In TinyObjects, each object stores values for a set of slots specified by its class. For example, the class **`Politician`** defines one slot—**`concerns`**. (It inherits no slots from its superclass **`Object.)`** So, **`Politician`** objects each end up with a single slot, **`concerns`**.

At the lowest level, slots can be read and written using the dot notation shown on the slide.

Note that on this slide we are making a simple use of a list data structure similar to that found in many data structure libraries. (For more about this data structure see page 73 and page 74).

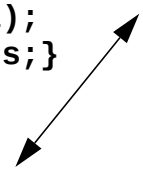
Question: You've said that TinyObjects has a multiple inheritance model. What happens if two different superclasses define a slot with the same name?

Answer: In this situation, TinyObjects works like CLOS and Smalltalk, not like C++. That is, the resulting objects end up with only a single slot with that name. While this simple solution is often satisfactory, there are cases where it is inappropriate (e.g., the diamond example mentioned on page 75 and page 77). This may be something to add to your list of behaviors that an open implementation of TinyObjects should allow programmers to adjust for themselves.

TinyObjects:Slot access;Slot:accessing

Initialization of Objects

```
initializer Politician (pol, concerns) {
   initialize Object (pol);
   pol.concerns = concerns;}
```



```
aPol = new Politician(list("hi tech"))

aPol.concerns  -->  ("hi tech")
```

Object initialization is accomplished through a built-in mechanism that allows each class to have its own *initializer*. These are special pieces of code that are called automatically as part of object creation.

In the example above, the initializer accepts two arguments. The first is the actual new object, the second is the first of the initargs supplied to **new**.

Like many initializers, this one simply runs the superclass initializer and then stores the value of the **concerns** initarg in a slot. It is important to note though that the body of an initializer can include any arbitrary code. (Examples of more elaborate initializers will come up shortly.)

Initializers define the initargs that a class accepts, so from the perspective of instantiators of the **Politician** class, the effect of this initializer is to say that when creating politicians, the first and only initarg should be the concerns. Similarly, note that the class **Object** is not expecting any initargs. Only the object itself is passed in the call to the initializer for **Object**.

TinyObjects:Object initialization;Object:initialization

Encapsulation

Reader and Writer Generic Functions and Methods

```
generic concerns (pol);
generic setConcerns (pol, concerns);


method concerns (Politician pol) {
   return pol.concerns;}

method setConcerns (Politician pol, concerns) {
   pol.concerns = concerns;}
```

When programming in TinyObjects, a stylistic convention is used which discourages clients from accessing the slots of an object directly. Instead, generic functions and appropriate methods for accessing the state of an object are defined, even when the state being accessed is stored directly as the value of a slot. This encapsulation convention provides better modularity, and allows the programmer to change the internal representation of objects without changing their interface.

Note that to save space, in later examples the explicit definition of such state accessing generic functions and methods, which we refer to as *readers* and *writers,* will often be omitted and simply assumed to exist.

Exercise: Some languages, including C++, provide an explicit mechanism to enforce this slot access discipline. For reasons of simplicity, we have chosen to leave this out of TinyObjects. Was this the right choice? How can you be sure? Or is this an example of a design decision that is impossible to get just right for all prospective programmers?

TinyObjects:Encapsulation

Using Readers and Writers

```
concerns(mayor) --> (“IRS”)

setConcerns(mayor, list(“ethics committee”))

concerns(mayor) --> (“ethics committee”)
```

Reader and writer generic functions are called just like any other generic functions.

The Complete Program (1/5)

Classes

```
class Politician (Object) (concerns);

class Manager (Object) ();

class Elected (Object) (margin);

class Senator (Elected, Politician) (state);
```

The next five pages present all the code for the voting program, including some new code not shown before. Along the way, several important points about polymorphic programs in TinyObjects are emphasized. Once all the code has been presented the chapter concludes with several pages that discuss the full behavior of method dispatch in the presence of multiple inheritance.

This slide shows all the class definitions, including two new classes, **`Elected`** and **`Senator`**. The class **`Elected`** is a *mixin*-class, not designed to be instantiated on its own.[1] The class **`Senator`** is an instantiable class and, as shown, **`Senator`**s are elected Politicians.

Class:definitio;TinyObjects:Class definition; <$startrange>Examples:TinyObjects basics; Polymorphism<start>; Multiple inheritance:TinyObjects in

1. The use of mixin-classes is a stylistic convention popular in some OO communities. As used in this book, the mixin-classes are not in any way syntactically distinguished, although adding such a feature is a good thing to consider when you read Chapter 5.

The Complete Program (2/5)

Readers and Writers

```
generic concerns (pol);

method concerns(Polictician.pol) {
   return pol.concerns;}
```

-
-
-

Similar definitions for margin, state and set concerns

At the bottom of the figure are three readers and one writer. The concerns of a politician can be read and written, but the margin of an elected politician and the state of a senator are fixed at the time those objects are created. They can only be read, and may not be modified or written afterwards.

The Complete Program (3/5)

Initialization

```
initializer Politician (pol, concerns){
   initialize Object (pol);
   pol.concerns = concerns;}

initializer Elected (pol, margin){
   initialize Object (pol);
   pol.margin = margin;}

initializer Senator (pol, concerns,
                     margin, state){
   initialize Elected (pol, margin);
   initialize Politician (pol, concerns);
```

This slide shows the initializers for all the classes that have one. Note that a class is not required to have an initializer. Again, from a programmer's perspective, these initializers mandate the format of the initargs list when instances of each class are created. So, for example, creating a senator requires three initargs: the concerns, the margin of victory, and the state.

It is the responsibility of the initializer of a subclass to make sure that the superclass initializer(s) are called. As shown, the usual convention for doing this is that the superclass initializers are called first, and then the class-specific initialization is performed.[1]

Object:initialization

1. Note also that the initializer for the class **`object`** will end up being run twice. This is true of the initializer for any class inherited along multiple paths. This means that the code for any initializer must be idempotent. (See page 75 for more about such "diamond" cases.)

The Complete Program (4/5)

Generic Function and Methods

```
generic vote(person, issue);

method vote (Politician pol, issue) {
      make the popular choice}

method vote (Manager who, issue) {

method vote ((who Manager) issue) {
      make cost-effective choice}
```

Here is the definition of the generic function **vote** and the two methods on it that we've already seen.

Generic function:applicability

The Complete Program (5/5)

More Methods

```
method vote (Elected pol, issue) {
   if (margin(pol) < 10)
      return agreeWith(findLobby(issue));
   else
      return callNextMethod();}

method vote (Senator pol, issue) {
   if (isIn(issue, concerns(pol))) {
      expound(randomCitation());
      return vote(pol, issue);}
   else
      return callNextMethod();}
```

Here are two more methods on the generic function **vote**, that define special behavior for the classes **Elected** and **Senator**.

The effect of the first method is to make the politician cave in to the appropriate lobby if their margin of victory was narrow (the first branch of the **if**). If their margin was 10 or more, they just defer to the next most specific method by invoking **callNextMethod**.

Senators also defer to the next most specific method, unless the issue is one of their special concerns; in that case they filibuster (enter the infinitely recursive call to **vote**).

<$endrange>Examples:TinyObjects basics;Polymorphism<end>

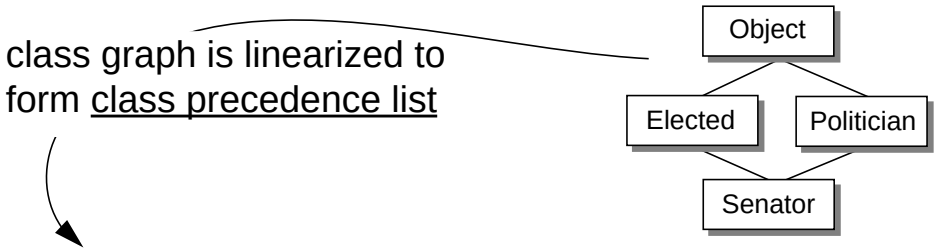
Making Senators

```
senCa1 = new Senator("California",
                     10,
                     list("hot tub quota"))

senCa2 = new Senator("California",
                     3,
                     list("ozone hole",
                          "prop wash shortage"))
```

This code creates two senators from California, each with their particular margin of victory and list of concerns. Again, the number and meaning of the initargs supplied to **new** is defined on a class-specific basis by the relevant initializer, in this case by the initializer for the class **Senator**.

Using this complete example, the next several pages conclude the chapter by presenting the full functionality of method lookup and overriding in the presence of multiple inheritance.

Method Dispatch (1/4)



(Senator Elected Politician Object)

This figure introduces the concept of *class precedence list* which plays a central role in method dispatch. The class precedence list (CPL) of a class is a linearization of its superclasses. The details of the linearization are not crucial for our purposes. The essential points of the linearization are that: (i) subclasses always precede their superclasses, and, (ii) the order that classes appear in the superclasses field of **`class`** is preserved.

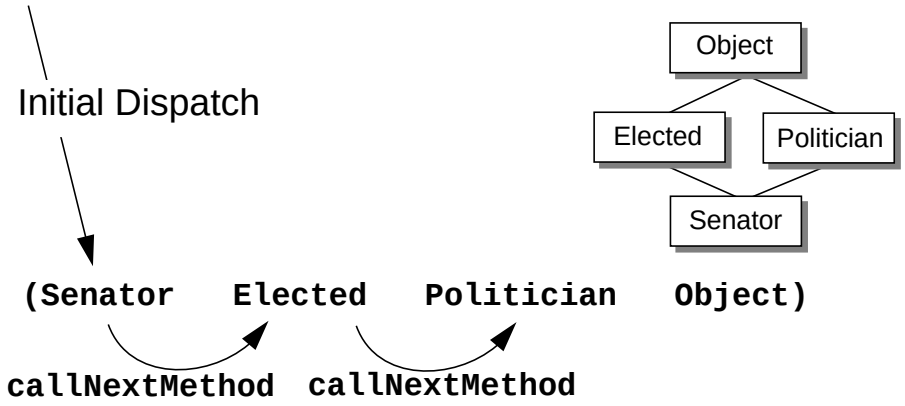
Method dispatch and **`callNextMethod`** rely on the CPL. When a generic function is called, it first obtains the CPL of the class of the first argument, and then selects from among the generic function's methods the one specialized to the class that appears *earliest* in the CPL. If **`callNextMethod`** is invoked in the body of that method, the method specialized to the class that appears next in the CPL is run, and so on.

To understand this better, consider the detailed examples on the next pages.

Class precedence list

Method Dispatch (2/4)

```
vote(senCa1, "suntan lotion requirement")
```



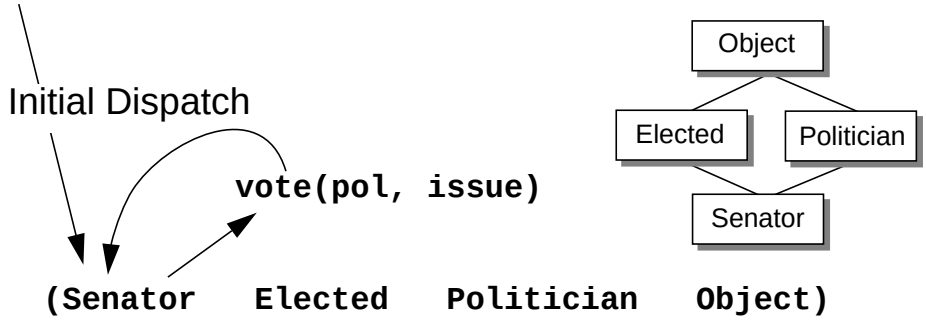
When the generic function call shown above is executed, the first method that runs is the one specialized to **`Senator`** (bottom of page 47). Since suntan lotion is not one of the senator's special concerns, that method invokes **`callNextMethod`**, which then runs the method specialized to **`Elected`** (top of page 46). When that method invokes **`callNextMethod,`** it runs the method specialized to **`Politician`** (page 46) which makes the popular choice and returns "no."

Notice the importance of having **`callNextMethod`** follow the class precedence list of the first argument to the generic function. The second invocation of **`callNextMethod`** ends up running the method specialized to **`Politician`**, even though **`Politician`** is not a superclass of **`Elected`**. This behavior is what makes multiple inheritance and mixin-classes practical to use in TinyObjects.

Method:invocation

Method Dispatch (3/4)

```
vote(senCa1, "hot tub quota")
```



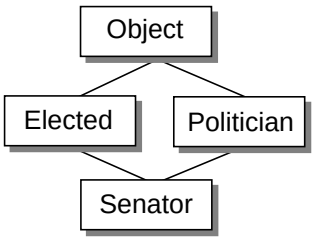
As a second example, consider the case where **senCa1** votes on a hot tub quota which is one of their concerns. In this case, the bottom method on page 47 just calls **vote** recursively, causing a filibuster.

It is important to stress that not every class must have a method for every generic function. For example, if there was no method on **vote** specialized to **Senator**, method dispatch would jump to the next class in the CPL (e.g., **Elected**). If on the other hand there was no method specialized to **Elected** (but there was for **Senator** and **Politician**, then **callNext-Method** from the method specialized to **Senator** would go to the method specialized to **Politician**, skipping **Elected**.

Method Dispatch (4/4)

```
vote(senCa2, "high-rise
             restrictions")
```

Initial Dispatch




```
(Senator   Elected   Politician   Object)
```

```
callNextMethod
```

Finally, when someone like **`senCa2`**, with the low margin of victory, votes on an issue that is not one of their concerns, the method specialized to **`Elected`** decides to agree with an appropriate lobby.

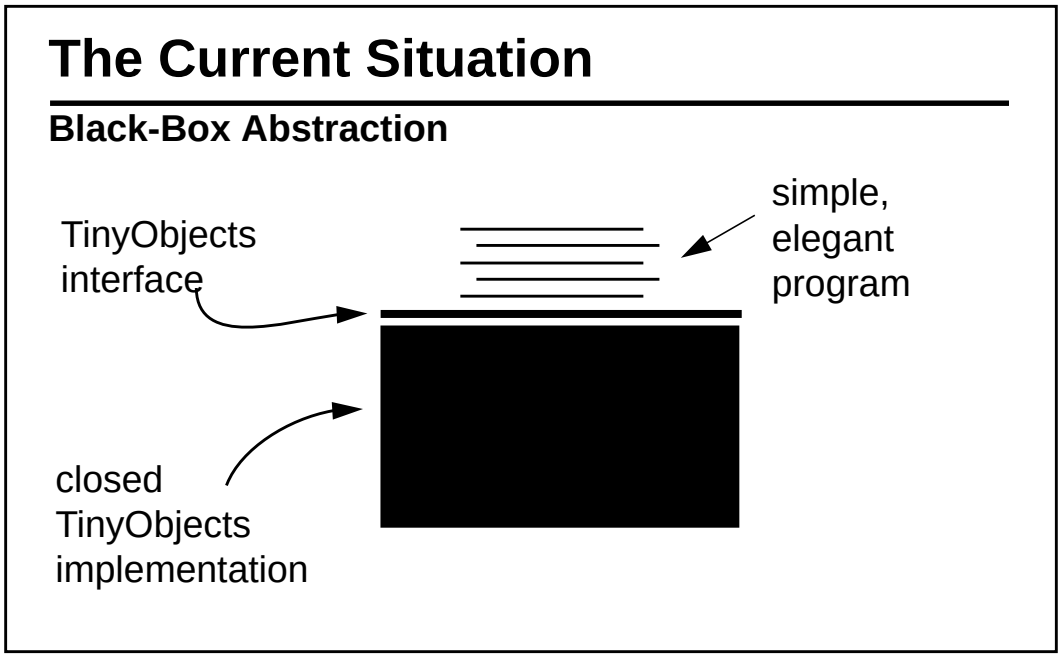
This concludes the introduction to TinyObjects. This is all the preparation needed to start showing how a meta-object protocol can be used to open the implementation of a substrate like this, thereby making it more flexible and useful for the programmer.

Chapter 3 But I Wish I Knew…

In this chapter we will observe the first of several design cycle iterations that open the implementation of Tiny CLOS. This first iteration will result in an implementation with introspective capabilities, which allow programmers to examine the state of the system, but do not allow any modifications. Subsequent iterations will...

This book uses an iterative and example-driven style throughout. In fact, this approach is important to the design of any real open implementation.

If done right, at the end of this cycle, a whole class of shortcomings requiring the newly accessible functionality can be addressed by the programmer, without further involvement of the substrate designer or implementors.



TinyObjects is, like most of its cousins, a “black-box” design. It presents the programmer with a simple narrow interface through which instructions, such as class and method definitions are passed. No aspect of the object system’s implementation is visible outside of the black box. Programmers write code on top of this interface, with no knowledge of the implementation below them.

This chapter will show that the totality of information hiding traditionally associated with black-box abstractions is not always an advantage. It will also show that the notion of black-box abstraction can be expanded to allo w some useful “looking inside the box” and how this can be done without compromising the important maintainability, robustness and portability goals of the approach.

Black-box:abstraction

Flow of the Following Material

Successive examples of programmers problems working with TinyObjects elicit responses from the designer that progressively open the implementation.

The actual messages from programmers are made up, but they are representative of real problems seen with real object-oriented systems.

The material from here to page 78 shows the first complete iteration through an open implementation design cycle:

- Programmer complaints prompt the designer to consider changing the substrate design (page 58).
- The designer analyzes the complaints to get a more general sense of what programmers really need (page 59).
- The designer XXX. (page 62 to page 66).
- The designer develops a clean abstraction to such information and mechanisms, thereby opening the implementation as needed—in a disciplined fashion (page 67).
- The designer tests the abstraction (page 69 to page 77).
- The designer revisits the programmer complaints to ensure they have been adequately dealt with, thereby repeating the previous five steps (page 79 to page 77).
- The designer ensures coherence of the resulting extended system and the existence of efficient implementation algorithms (page 89 and page 84). Design cycle

I Wish...

A TinyObjects programmer writes:

> For my new quality assurance program I need to write programs that have access to information about what subclasses a class has, which slots are inherited from where, which methods are specialized on which class, and so on.
>
> How can I get this information?

This slide shows a message from a programmer asking for functionality that the basic TinyObjects doesn't support.

The slide shows just one message, but since the designer remembers many similar messages, she decides to see if there is some way she could improve TinyObjects to help solve these kinds of problems.

In working with these (or any other) programmer requests, the substrate designer faces a dual challenge. One the one hand, she wants to respect the particulars of each request. On the other hand, she wants to back away a bit from those particulars so that she can develop a more general understanding of the need and so design more generally useful functionality, that can satisfy all the requests.

<$startrange>Examples:browsers; <$startrange>Slot:reification; <$startrange>Subclass:reification; Reification:class

What Is This Programmer Asking For?

- Superclass/subclass information
- Slots defined on each class
- Methods specialized to each class

A representation of the inheritance structure of their program.

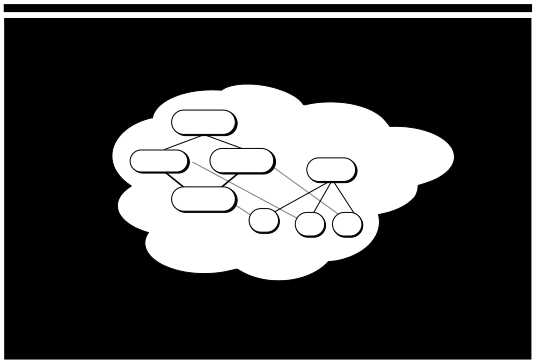
In this case, that stepping back and getting a more general sense is relatively straightforward. In essence, the programmer is asking for access to a representation of the basic inheritance structure of his programs.

But note that requests from the programmer will not always be this clear. Because the programmer is trained to respect abstraction and not to ask for what is under the covers, and because the system designer is trained to hide information, communication surrounding this kind of request can be murky. It can take quite some time to get to a clear understanding of what a programmer needs.

<$startrange>Class:reification; Class:linkage with methods, generic functions and subclasses; Generic function: linkage with classes and methods; Method:linkage with classes and generic functions; Subclass:linkage with superclasses

Inherent Program Representation

Looking for a representation of the inheritance structure of the program.



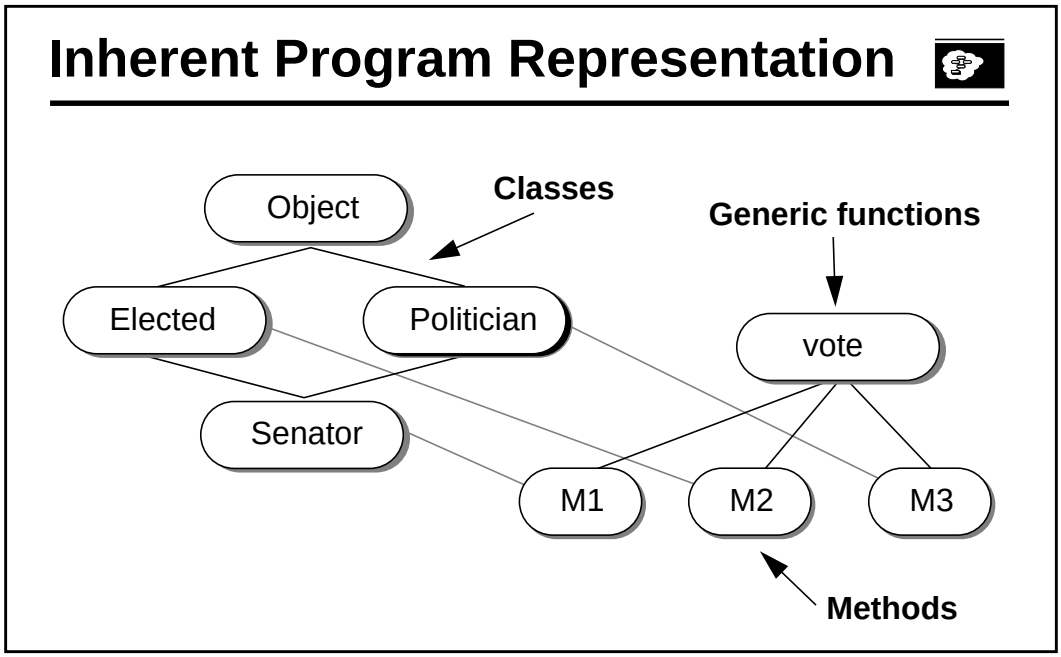
At this point the designer's job is to understand whether this is a reasonable request and what it might mean for TinyObjects to provide such functionality. To do this, the designer now shifts gears a bit and starts thinking about what it would mean, from the TinyObjects implementor's perspective, to provide such functionality. The question is, would it be reasonable for implementations to maintain and export, in some form, a representation of the program?

In thinking about this, the designer is going to rely on her sense of the inherent structure of Tiny Objects implementations --- exposing that to the programmer in a principled way is what it means to design an Open Implementation of Tiny Objects. She will however be careful not to allow the details of particular implementations to end up being visible to the programmer.

The notion of a system's inherent implementation structure is a subtle one, and it requires some time to become comfortable with. As with most other design guidelines, the best way to learn it is from a series of examples, rather than an abstract definition. This kind of example, together with dis cussion is what this book presents.

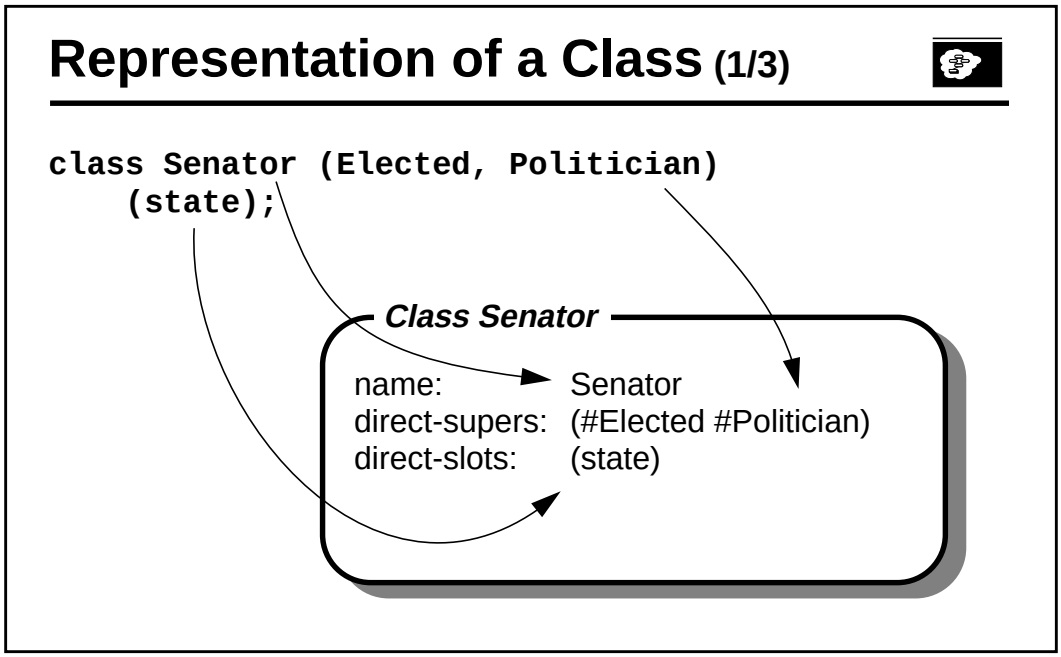
Note the cloud icon at the top right of the slide. We will use this icon throughout the book to mark slides that show the designer's thoughts about the inherent structure of the implementation.

Black-box:partial opening;Cloud icon; Icons:cloud



This figure goes into more detail inside the cloud. The designer is now thinking about what such a program representation might be like. Checking her speculation to be sure it fits within the inherent structure of Tiny Objects implementations, the designer is satisfied she is within reason since any TinyObjects complier inherently has to have a representation of the program in order to compile it. This check is important for two reasons: First it suggests that such functionality could be easy to implement. Second, it reinforces her sense that this is a reasonable thing for programmers to want access to.

For people who are not object system designers or implementors, the last paragraph may be a bit surprising. But object systems are not magic. In the same way that a window system has representations of windows, or an accounting system has representations of accounts, a TinyObjects implementation will have representations of classes, methods and generic functions.

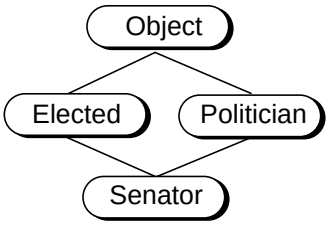


The designer now goes on to consider in more detail the desired program representation should look likeThe box with the rounded corners shows an elaboration of one of the boxes on page 62. The notation “#xxx” (sharp followed by anything) is a compact visual way of indicating a structure passed around by an implementation.)

As she is drawing these pictures, the designer is aware that not all implementations will use record-like structures such as these to maintain this information. But she is confident that, whatever form it takes, a representation of each class, including this information taken from the class definition, is an inherent aspect of implemeting Tiny Objects.

Sign convention

Representation of a Class (2/3)



Class Senator

name:	Senator
direct-supers:	(#Elected #Politician)
direct-slots:	(state)
CPL:	(#Senator #Elected #Politician #Object)
slots:	(state margin concerns)

The class precedence list (CPL) and complete set of slots are computed and stored

The designer now reasons that such class descriptions should also contain various kinds of derived information, such as:

- The *class precedence list* (CPL), which is a collection of all the superclasses, direct and indirect, ordered to reflect the relative specificity of methods. (See page 51, if you wish to review CPLs
- A complete list of all the class's slots, those defined directly in the class as well as those inherited from superclasses.

Representation of a Class (3/3)

```
class Freshman (Senator) ();
```

Class Senator

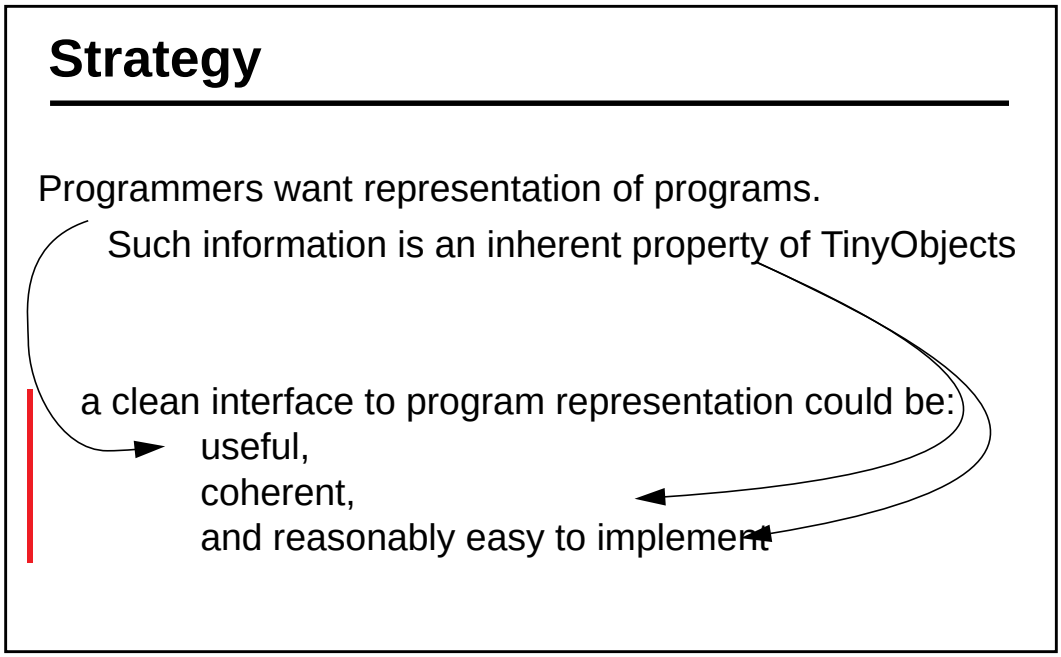
name:	Senator
direct-supers:	(#Elected #Politician)
direct-slots:	(state)
CPL:	(#Senator #Elected) #Politician #Object)
slots:	(state margin concerns)
direct-subs	(#Freshman)

The designer also believes that the representation of a class should point to its subclasses as well as its superclasses.

Question: It isn't clear to me that all TinyObjects implementations would actually maintain pointers from superclasses to subclasses. Is this really a valid assumption?

Answer: While it is true that some implementations might not maintain such pointers, many implementations do, if only for the purpose of supporting debugging

This question brings up an important point about what "the inherent structure of the implementation" means. Specifically, is is a broader concept than what existing implementations necessarily do. It is more like what it is reasonable to think of them as doing, and reasonable to require them to do. Keeping pointers from a class to its subclasses may not be something-closed Tiny Objects implementations do, but it is within the inherent structure of Tiny Objects implementations. If this concerns you, be sure to consider it when you do the exercise on page 94.



So far, two steps of the design cycle have been completed: (i) the object system designer has gotten a good sense of what additional functionality the programmer needs to address his problems, and (ii) with an eye towards providing this functionality, the designer has thought carefully about the inherent structure of TinyObjects and what would make sense to provide.

In this case, she can conclude relatively easily that providing such functionality appears to be useful, coherent and implementable. Her conclusion comes easily in this case because both the programmer complaints were so clear, and the degree to which this functionality is an inherent part of Tiny Objects implementation is so clear.

Access to Data about Classes

Implementations will provide objects to describe each class in a program. These objects support a small set of readers.

Get class objects:

from class name: `@findClass("Senator")`
from object: `@classOf(senCa1)`

Readers:

```
@className@classCPL
@classDirectSupers  @classSlots
@classDirectSlots
@classDirectSubs
```

<IndexCode>@ <Default Para Font>Sign convention;Scroll icon; Icons:scroll; Metaobject;Class:metaobjects;Class:accessor functions

At this point she needs to design an interface that is clean and elegant, and that provides appropriate insulation between the programmer and the implementation.

What is needed is a simple data abstraction, so the designer decides to present that in the form of objects, using the encapsulation properties of Object-Orientation to help get a clean abstraction. Her decision to use objects is for two reasons: it is natural, the programmers by definition are already comfortable with objects.

This slide summarizes the data abstraction developed by the designer. It works by making available, on a per-class basis, an object that provides access to important information about that class.

All the accessors begin with an "@." This is simply a naming convention about which more will be said soon. To save space, the return values of these accessors are not specified in detail here, but they return the "natural" values. That is, @classDirectSupers returns a list of class description objects. Similarly, @classDirectSlots returns a list of slot names as strings. The issues that arise when writing a more detailed specification of these accessors will be addressed later.

It is important to note that the material on this slide represents a proposed extension to the Tiny Objects specification. This fact is marked by the small scroll icon in the corner. If the designer decides to go ahead and publish this, all Tiny Objects implementations will then be required to support this new functionality. Before publishing it then, the designer will be careful to test it, and assure herself that it is indeed useful and implementable.

A NEW SLIDE GOES HERE COMPARING THE AMORPHOUS CLOUD ICON TO THE SCROLL ICON IN MEANING

It is crucial to distinguish between the meanings of slides with a cloud and slides with a scroll. Slides with a cloud reflect the designer's initial thinking about the inherent structure of Tiny Objects implementations and how her design might work. Slides with a scroll reflect a real proposed design. Do not, for example, let a slide with a cloud, such as the one on page 64, lead to an assumption that a "documented[1] class description object will have a slot named **`directSlots`**." Slides with a scroll are all that will documented—they are all the programmer will be able to count on. The one on page 67, for instance, says that the only way to get the direct slots of a class is to call **`@classDirectSlots`**.

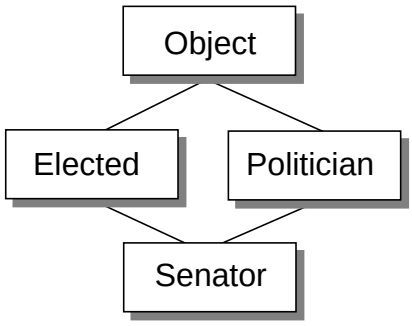
At this point the designer is making good progress towards addressing this first set of programmer requests. But, she still has to ensure that her proposed design is both truly implementable and truly useful. She decides to assess its usefulness first, by trying to code up solutions to the programmer problems that have been posed.

1. The terms *documented* and *specified* will be used interchangeably.

Finding All Subclasses (1/3)

Testing the new facilities

`@classDirectSubs`
returns down links



`@classDirectSupers`
returns up links

The first programmer problem she turns to concerns finding the subclasses and superclasses of a class. For example, the programmer might want to find out the transitive closure of a class's subclasses. This figure shows the available interface and functionality from the programmer's perspective. Again, it is crucial to understand that the designer is now working from the programmer's perspective. She has to understand whether the functionality she has designed will solve the programmer's problems. The fact that this slide is using the new functionality is reflected by the small mop icon in the upper right. (The mop is because we'll shortly be saying that this new functionality is a metaobject protocol (MOP).)

Using the new functionality, it is straightforward to obtain the complete list of subclasses of **`Object`** by walking down the tree, calling **`@classDirectSubs`** for each node to get to the next level down.

Subclass:finding; Mop icon; Icons:mop

Finding All Subclasses (2/3)

A First Use of Class Readers

```
function @allSubs (c) {
   var result = list();
   @walkSubs(c, result);
   return result;}

function @walkSubs (c, result) {
   extend(result, c);
   foreach(sub, @classDirectSubs(c))
     @walkSubs(sub, result);}
```

Note:
This accepts and returns class description objects.

This slide shows the actual code the designer believes the programmer should write to implement the tree walk.[1] Mostly, it looks like any other tree walking program. It recursively walks down the class graph, collecting each class as it goes. In the interest of brevity, code that removes duplicates from the returned list of classes has been left out. Senator will therefore appear twice in the result of calling @allSubs with the class Object.

This slide makes use of several more of the built-in list operators mentioned in Chapter 2. The statement **`extend(result,c)`** adds the class description object **`c`** to the end of the list **`result`**. The **`foreach`** assigns successive elements of the list returned by **`@classDirectSubs`** to the variable **`sub`** and executes its body. When the list is exhausted, the loop terminates. For mor information on these see page 72.

TinyObjects:<IndexCode>extend<Default Para Font> operator; Class:visual representation conventions

1. We assume that the designer has mocked up an implementation of the new functionality so that she can actually run her test programs, as opposed to just write and think about them.

Finding All Subclasses (3/3)

```
@allSubs(@findClass("Politician"))
   -->  (#Politician #Senator)


function @allClasses () {
   return @allSubs(@findClass("Object"));}


@allClasses()  -->  (#Object #Politician
                     #Elected #Senator  ...)
```

A simple test of the **@allSubs** function shows that it works as desired.

By taking advantage of the fact that the class **Object** is at the top of the class hierarchy in every TinyObjects program, the programmer could build another useful function, **@allClasses**, which returns every class currently loaded, simply by calling **@allSubs**.

Exercise: As written above, the function **@allSubs** accepts and returns class description objects. But this can be inconvenient in some cases. Write a similar function that accepts and returns class names.

Summary of List Operations (1/2)

```
letters = list("A", "B", "C")-->   ("A" "B" "C")

first(letters) -->  "A"

 isIn("B", letters) -->  true

 var labelValuePairs = list(list("A", 1),
                            list("B", 2),
                            list("C", "Blue"))

 lookup("A", labelValuePairs) --> 1
```

This slide and the next provide a concise summary of the list data structure being used throughout the book. The behavior of each of the operations should be clear from the examples.'The fact that TinyObjects includes and automatic garbage collector is most evident in code that manipulates lists. There are no calls to **free** before list a becomes unused.

TinyObjects:<IndexCode>first<Default Para Font> operator; TinyObjects:<IndexCode>isIn<Default Para Font> operator; TinyObjects:<IndexCode>list<Default Para Font> operator; TinyObjects:<IndexCode>lookup<Default Para Font> operator;

<IndexCode>first<Default Para Font> operator; <IndexCode>isIn<Default Para Font> operator; <IndexCode>list<Default Para Font> operator; <IndexCode>lookup<Default Para Font> operator

Summary of List Operations (2/2)

extend appends an item to the end of a list.
union merges a list onto the end of a list.

```
var result = list();
 foreach(river, (list("Nile", "Amazon")))
  extend(result, direction(river));
 return result; --> (("N" "S") ("E" "W"))

 var result = list();
 foreach(river, (list("Nile", "Amazon")))
  union(result, direction(river));
 return result; --> ("N" "S" "E" "W")
```

Assume in the code above, that **direction(river)** returns a list of directions, such as **("N" "S")**. Since **extend(list,element)** appends **element** onto the end of **list,** the result of the first loop is a list of lists. On the other hand, since. **Union(list,sublist)**, merges the new **sub-list** onto the end of **list**.The result of the second loop is a list of directions.

<IndexCode>Union<Default Para Font> operator;TinyObjects:<IndexCode>Union<Default Para Font> operator; <$endrange>Subclass:finding; <IndexCode>extend<Default Para Font> operator

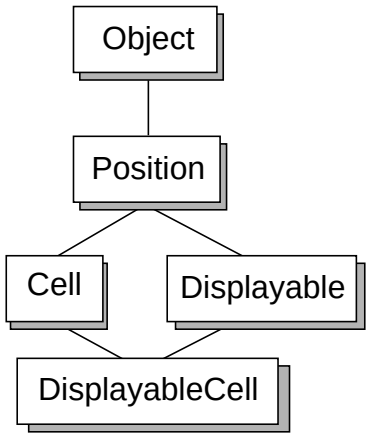
How Much Multiple Inheritance?

Which Classes Have Multiple Direct Supers?

```
function @classesWithMI () {
 var result = list();
 foreach (c, @allClasses())
    if (length(@classDirectSupers(c)) > 1)
      extend(result, c);
 return result;}
```

Continuing to experiment with the newly designed functionality, here is another program the designer believes the programmer might want to write. It gathers a list of all classes that have multiple direct superclasses. It works quite simply by iterating through a list of all the classes, collecting those that have more than one superclass into a result list.

Exercise: Some cases of multiple inheritance are potentially more problematic than others. In particular, the 'diamond' case is one of the most troublesome. As shown to the right, a diamond happens when a class has two routes to a superclass other than **`Object`**. In the figure, there is a diamond from **`DisplayableCell`** to **`Position`**. Write a program to find all diamonds. Your program should return a list of diamonds where each diamond is itself a list of the root and leaf node of the diamond.



Multiple inheritance:finding in program

Slot Genealogy (1/2)

Where Did Slots in a Class Come From?

```
@slotOrigin("Senator", "margin")
   --> "Elected"

function @slotOrigin (className, slotName) {
  var c = @findClass(className);
  foreach (ancestor, @classCPL(c))
   if (isIn(slotName,
            @classDirectSlots(ancestor)))
     return @className(ancestor);
  return false;}
```

Another of the quality assurance programmer's questions had to do with slot inheritance. The question is, for a given slot in a given class, which class was that slot actually defined in? There are different forms this sort of functionality can take. The function shown here returns the name of the first superclass that defines the slot.

Again, this is a simple function. It works by iterating through the class's ancestors (the class precedence list), looking for a class that includes the slot name in its list of direct slots.

<$startrange>Slot:genealogy, finding

Slot Genealogy (2/2)

Finding *all* ancestors:

```
@slotGenealogy("Senator", "margin")
   -->  ("Elected")

function @slotGenealogy(className, slotName) {
   var result = list();
   var c = @findClass(className);
   foreach (ancestor, @classCpl(c))
     if (isIn(slotName,
              @classDirectSlots(ancestor)))
       extend(result, @className(ancestor));
 return result;}
```

Here is a somewhat different version of this facility that returns the names of all the ancestors that define this slot name.

Question: How does one decide whether a function should accept a class name or. a class description object?

Answer: In general, it is convenient to have interactive functions take class names. Functions usually called by other functions should take class objects, because they can then skip the step of obtaining the object from the name.

Exercise: The concept of diamonds of inheritance was presented in the exercise on page 69. Edit the program you wrote for that exercise to report only diamonds in which the class at the top of the diamond defines slots.

<$endrange>Examples:browsers; <$endrange>Slot:genealogy, finding

Anything Else to Try?

At this point the designer is beginning to feel confident that the functionality she has designed does indeed address the programmer requests. To be sure, she will of course try more tests.

Exercise: Ask yourself what tests would you try if you were the designer?

Exercise: Code up functions to compute each of the following: What is the average depth of leaf classes? How many classes do or don't define slots? Which slot names are used more than once?

Recap

A clean abstraction of the inherent representation of Tiny Objects programs provides the programmer with a powerful tool for addressing certain kinds of problems.

Stepping back one can summarize by saying that the designer felt that programmers could make good use of having access to the inherent representation of their program, so she developed a clean abstraction of that representation and has begun to test it. So far, her testing has show that her proposed design is indeed useful and works well.

Question: Wouldn't a good program browser be able to solve many of these programmer questions? Why introduce this functionality instead?

Answer: It is true that a good browser could solve some of the programmer questions that have come up. But note that an important property of the new functionality the designer is developing is that it is programmable. This means that the application programmer can use it to write programs which answer questions that are very specific to his particular needs and that are unlikely to have been anticipated and included in a general-purpose browser. So, while it might be a good idea for implementors to provide a browser in addition to these facilities, even a good graphical browser is not a replacement for them. Their programmable nature makes them more open-ended and able to serve a wider range of programmer needs.

Also note that, because the new functions are a documented part of TinyObjects, programs that use them will be portable, whereas most browsers tend to be specific to a particular implementation or platform.

<$endrange>Class:reification; <$endrange>Slot:reification; <$endrange>Subclass:reification; Reification:class

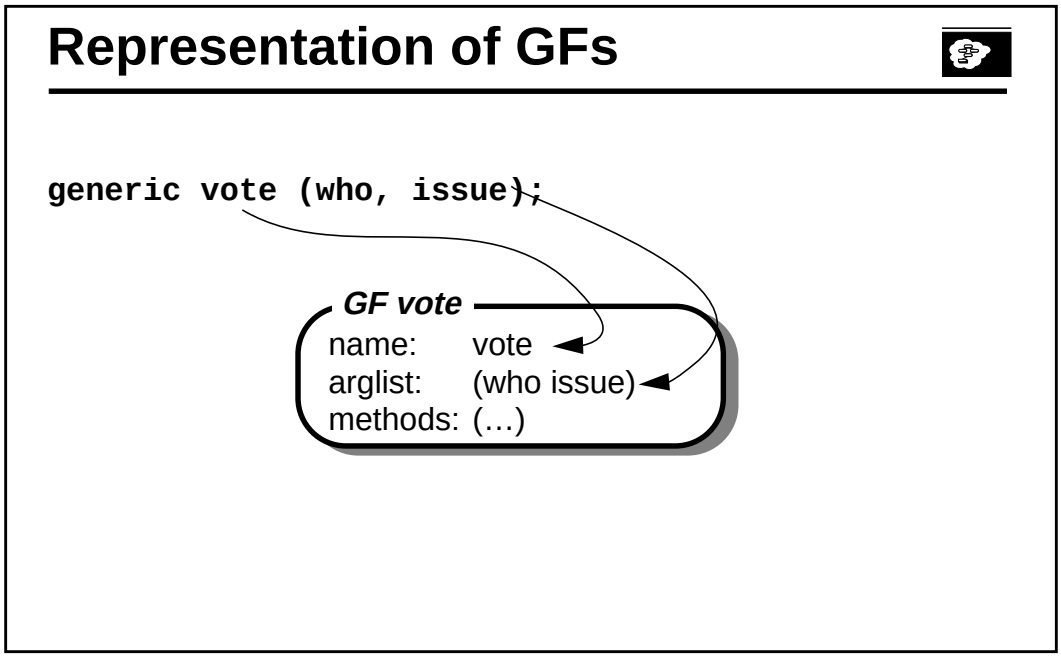
I Want to Know…

... whether a given generic function is applicable to objects of a given class.

We will now watch as the designer turns back to the programmer requests, and runs into an issue that her proposed design clearly does not address. In response to this problem she will design an analogous interface for generic functions and methods.

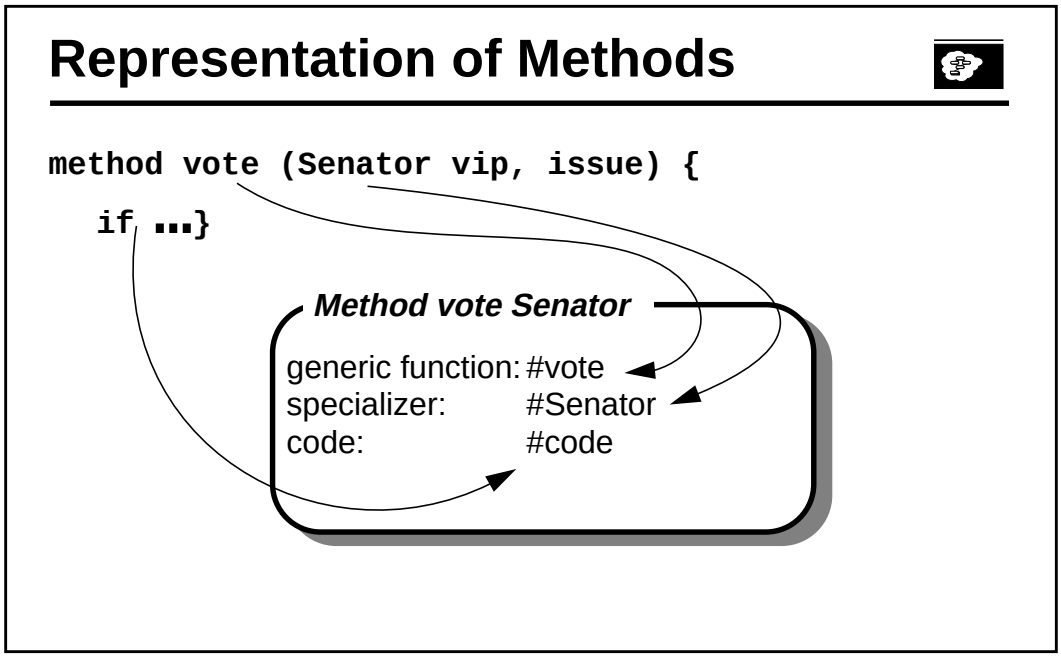
Once again, the designer's first step in response to this problem is to step back from its particular details to see what the programmer really wants; and once again, it is straightforward, he wants information about the generic functions and methods in his program. She sees this request as requiring an extension of the functionality she has developed for classes, and so she approaches the problem that way.

<$startrange>Examples:generic function applicability test



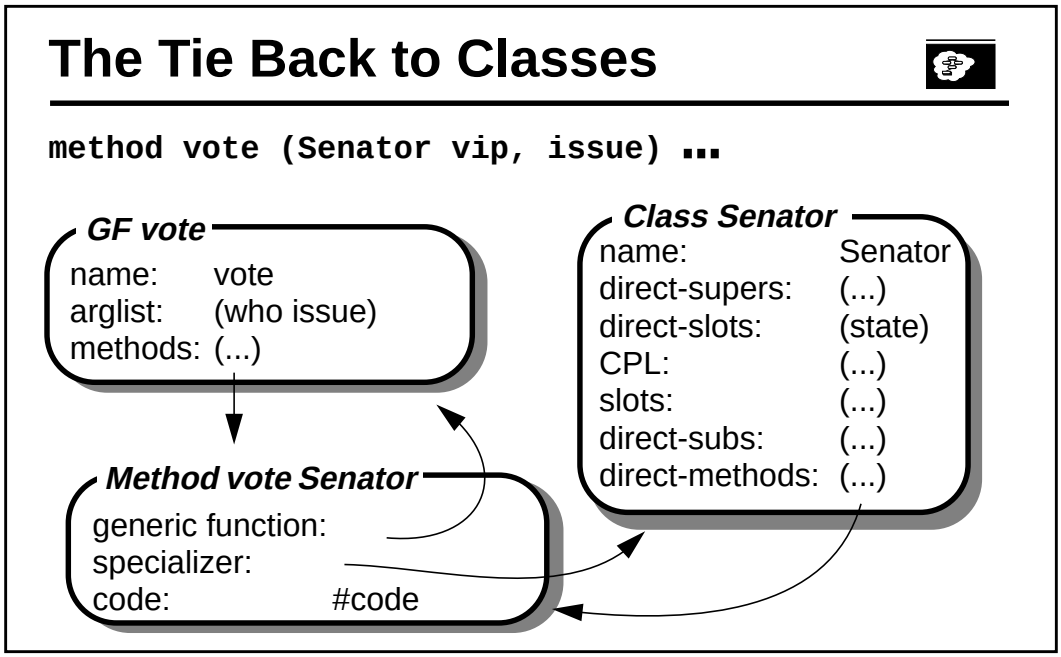
The designer's next step, again, is to understand what kind of relevant information in inherent to TinyObjects will inherently maintain. The black box with cloud marker on this slide indicates that once again, the object system designer is thinking about what inherent aspects of Tiny Objects implementation are relevant to this programmer problem. As shown, she believes there should be a representation of generic functions, similar to that for classes, that includes information from the generic function definition, as well as a list of methods defined on the generic function.

Generic function:information maintained for; <$startrange>Generic function reification



And that a similar representation of methods should include information from the method, as well as a connection to the generic function to which the method is attached and the class to which the the method is specialized.

Method:information maintained for; <$startrange>Method:reification;<$startrange>Reification:method



She concludes that classes, methods and generic functions are thus all interconnected as shown above.

Generic function: linkage with classes and methods; Method:linkage with classes and generic functions; Class:linkage with methods, generic functions and subclasses

Repeat the Strategy

Since information about GFs and methods is inherently managed by all implementations, the object system designer can use the encapsulation properties of OOP to:

Provide the programmer with a clean, standardized interface to that information, at only a small cost to object system implementors.

The programmer can navigate among classes, methods and GFs, examining important aspects of his programs.

Having once again worked out a sense of what functionality should be provided to the programmer, and been careful to ground that in the inherent structure of Tiny Objects implementation, the designer must now proceeds once again to design an abstract interface to that inherent implementation structure.

Readers for GF and Method Info

Implementations will provide objects that describe each GF and method. These objects support a small set of readers:

Get GF objects *(from the GF name)*: **`@findGf("vote")`**

Get method objects *(from a class)*: **`@classDirectMethods`**

Readers:

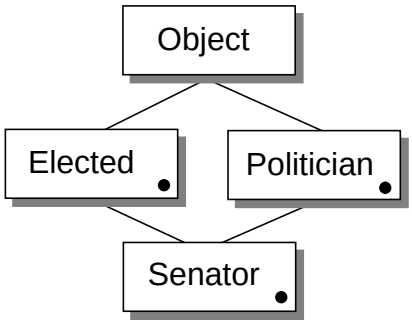
`@gfName`	**`@methodGf`**
`@gfArglist`	**`@methodSpecializer`**
`@gfMethods`	**`@methodFunction`**

Here, then, is the new piece of proposed design.

Just as with the class accessors on page 67, to save space we will simply say these accessors return the "natural" values, e.g., that **`@gfName`** returns a string, and **`@gfMethods`** returns a list of method description objects.

Metaobject; Generic function:reification; Method:reification; Reification:method; Generic function:accessor functions; Generic function:metaobjects; Method:accessor functions; Method:metaobjects;@classDirectMethods

Is a Given GF Applicable?




```
@isGfApplicable("vote",
                "Senator")
   -->  true

@methodGenealogy("vote",
                 "Senator")
   -->  ("Senator"
         "Elected"
         "Politician")
```

• denotes **vote** Methods

Having expanded her proposed design to include information about generic functions and methods, the designer now resumes testing it, starting with the problem from page 79 that prompted this expansion of the design. The problem was that the programmer wanted to know when a paricular generic function was applicable, that is whether there will be a method to run if the generic function is called with an object of a given class.

The figure above shows two different versions of this functionality the programmer might want to implement. The first simply says whether there is a method applicable when a generic function is called with an argument of a particular class. The second returns a list of the applicable methods.

As the designer now puts on the programmer's hat and tries to implement this functionality, remember that she must think only int terms of the proposed extensions to Tiny Objects. That is, she can only think about what is on slides with a scroll, not slides with a black box and cloud.

GF Applicability

```
function @isGfApplicable (gfName, className) {
 var gf = @findGf(gfName);
 var c = @findClass(className);
 foreach (m, @gfMethods(gf)) {
    var spec = @methodSpecializer(m);
    if (@isSubclass(c,spec))
      return true;}
 return false;}

function @isSubclass(c1,c2)
   return isIn(c2,@classCpl(c1));
```

One approach to program **@isGfApplicable** is to use **@gfMethods** to obtain the full set of methods for the generic function, and then to check whether the specializer of any of those methods is either the class in question or one of its superclasses.

The code above implements **@isGfApplicable** this way. It uses the **@isSubclass** helper function to test whether its first argument is a subclass of its second argument.

Exercise: The test for subclass relationship above returns **true** if **c1** and **c2** are the same. That is appropriate for this test, since methods defined on a class are applicable to that class, not just its subclasses. But, it still might be useful to have a version of **@isSubclass** that returned **false** if f **c1** and **c2** are the same. Write such a function, called **@isStrictSubclass**.

Method Genealogy

```
function @methodGenealogy (gfName, className) {
 var result = list();
 var gf = @findGf(gfName);
 var c = @findClass(className);
 foreach (m, @gfMethods(gf)) {
    var specl = @methodSpecializer(m);
    if (@isSubclass(c, specl))
      extend(result, @className(specl));}
 return result;}
```

The code on this slide implements the alternative version of the generic function applicability test --- it returns a list of all the methods applicable when a generic function is called on an argument of a specific class.

Exercise: One problem with this implementation of **`@methodGenealogy`** is that the order in which it returns the classes in is not necessarily the order of applicability of the methods. That is, it is not the same as the order in the class precedence list of the class named **`className`**. Write a new version of this function which always returns the classes in class precedence list order. Note that there are two ways to write this. One is to simply sort the result of the existing function, and the other is to arrive at the methods "the other way," that is by going in from the classes in the CPL rather than from the generic function.

<$endrange>Examples:generic function applicability test; Method:genealogy,finding

Anything Else to Try?

At this point, there are many other tests the designer would try, to continue to assure herself that the proposed design is indeed useful, elegant, and easy to use. Some of these might cause her to go back and modify or extend the design, others will just confirm that it works well.

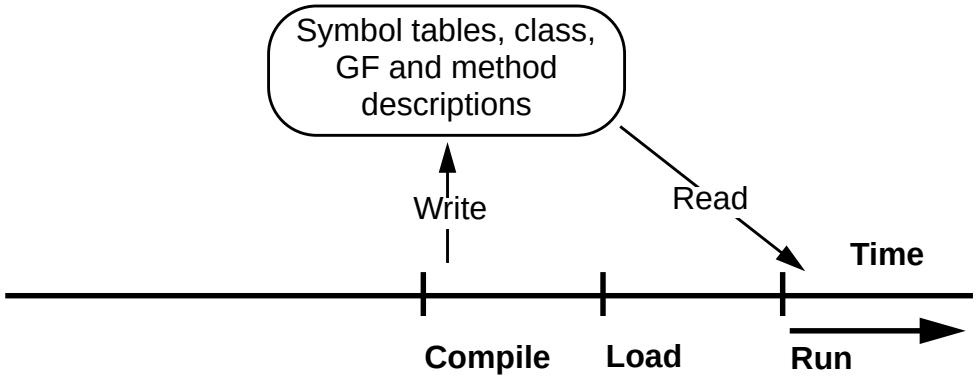
Exercise: Test the proposed design by writing a procedure that finds all the generic functions in a program. Use that procedure to write another one that computes the average number of methods on all generic functions.

Exercise: Where is most of the "functionality" (methods) defined in your object-oriented programs? Where is most of the "structure" (slots) defined? Root classes or leaf classes? Write a set of procedures to help you answer these questions. What can you learn about the modularity of your programs this way? About the degree to which you have followed a well-established design methodology? About the design methodology itself?

Exercise: Currently there is no way for the programmer to find out about a class's initializer. They can't, for example, write programs to see if a class has an initializer, or to see how many argument's a class's initializer accepts. Taking on the role of the designer, address this problem. What design would you come up with? What tests did you use?

Implementation Strategy

Existing debugger implementation techniques support this.



Having tested the proposed design for power and usability, the designer must now address the second aspect of testing -- to be sure that it is also implementable. She is already reasonably confident that the new functionality is implementable. The fact that her design is rooted in a sense of the inherent structure of TinyObjects gives her that confidence.

But she still must make certain that the new functionality is truly implementable. In this case there are two key questions: (i) how to make the compile time information available at runtime, and (ii), does her data abstraction provide sufficient implementation encapsulation.

Making such compile-time information available to the programmer at runtime is not a fundamentally new problem, since this is not so different from existing compilers that provide symbol tables for debugging at runtime. Providing descriptions of classes, generic functions and methods is just going one step further. In fact, a good TinyObjects implementation would no doubt already provide many of these facilities to support its debugging environment. In order to save space in the runtime image, the information could even be stored in a file or a database and retrieved on demand at runtime, just like symbol tables.

Encapsulation (1/2)

```
@className
@classDirectSupers
   •
   •
   •
```

```
class @Class (Object)
    (name,
     directSupers,
     directSlots,
     classPrecedenceList,
     slots,
     directSubs,
     directMethods);
```

With respect to the degree to which her design provides good encapsulation, the designer is also quite satisfied. The issue here is that she wants the design to be sufficiently abstract that different implementors have the leeway they need to implement this functionality in the way that works best for them.

Her design has one quite natural implementation in terms of objects with one slot for each property, as shown in this slide and the next. But there are many other possible approaches she feels the implementor could take. She is confident about this because, in documenting the functionality that has been added, she has been careful not to specify:

- how this information about classes is really stored in any particular implementation, or
- whether these externally visible class description objects share structure with any internal structures of the implementation.

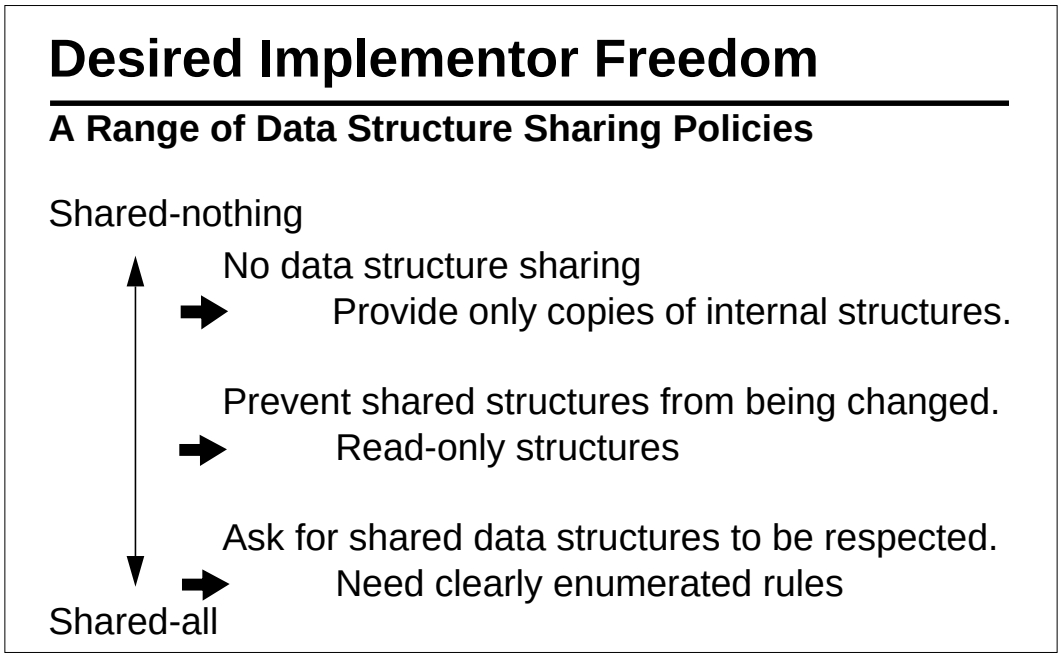
All she has documented is that the new functions exist, and what values they return when called.

One Possible Implementation (2/2)

```
function @className (c) {
   return c.name}

function @classDirectSupers (c) {
   return c.directSupers}
```

Exercise: Take a moment to think about the design and how well it encapsulates the implementation. For example, consider that some implementors might use a seven column table to store information about classes: one row for each class, and one column for each of the salient properties of a class. Or, some implementations might group the information into objects, but keep the actual properties in a different format. Or, some implementations might want to keep the information in a relational database. Does the proposed design provide adequate encapsulation for those internal implementations? Can you think of an internal structure that a Tiny Objects implementor might reasonably want to use that the proposed design does not adequately encapsulate?

At this point the designer becomes aware of encapsulation issues that her design does not yet address: What is the status of the values the proposed readers return? Is it possible for the programmer to do damage to the implementation (or other parts of their program) by mishandling those values? For example, what might happen if the programmer uses **`extend`** to modify a list returned by **`@classDirectSupers`**?



This is an instance of a general problem in designing a data abstraction --- the designer must make a decision as to whether the implementor can pass shared structure across the interface. On the one hand, passing shared structure can be more efficient (i.e. **`@classDirectSupers`** need not create a new list each time it is called). On the other hand, passing shared structure can introduce robustness problems (i.e. if the programmer mutates the list returned by **`@classDirectSupers`**).

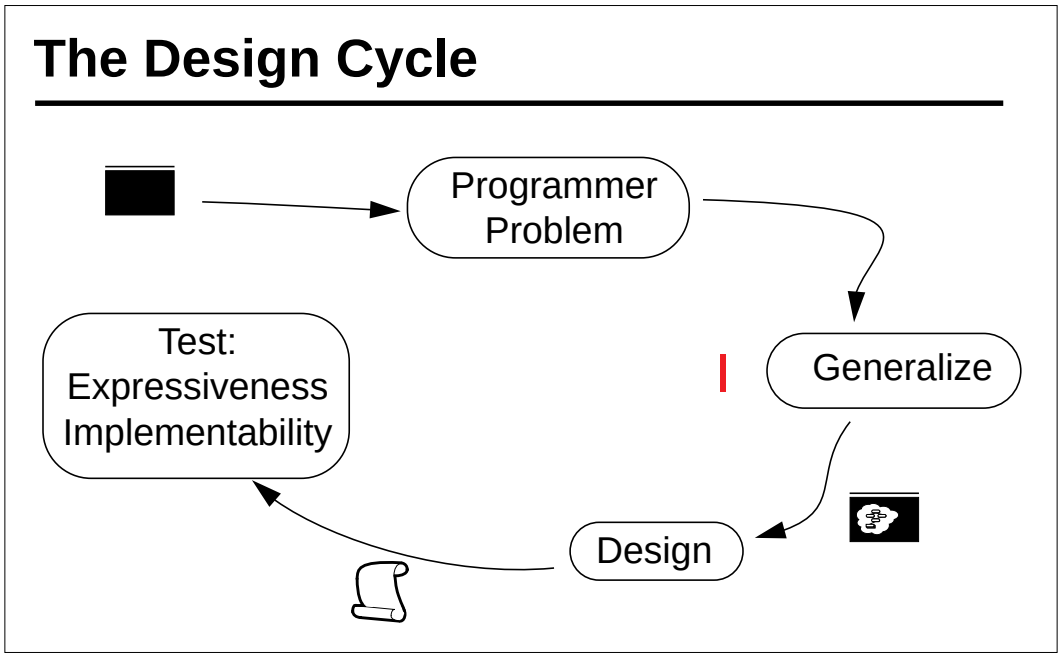
Rules for Using Readers

For all the class, gf and method readers, the programmer must:

- Not modify results obtained from the reader in any way.
- Not count on the values returned by successive calls to a reader to be an identical datastructure (i.e. two identical calls to **`@classSlots`** may return different lists, even though the will have the same elements).

The designer must take a stance about what sub-range of the sharing spectrum the implementor will be free to choose. The designer must decide, because the sharing policy can, in various ways, manifest itself to the programmer, and so it must be covered in the specification.

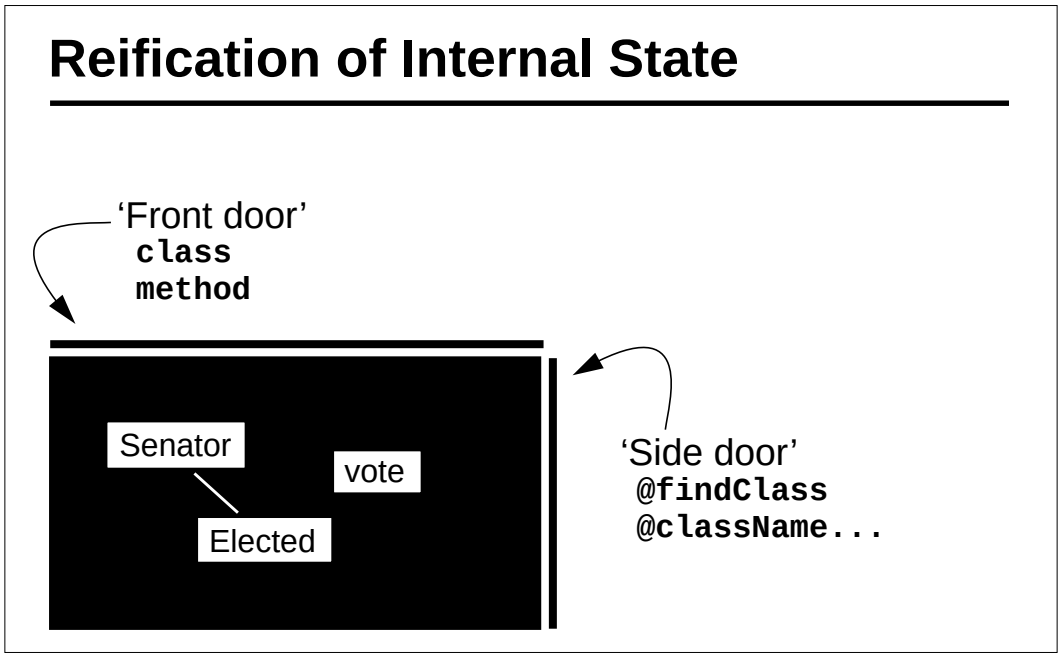
In this case, the designer chooses an extrremely conservative stance. She will allow the implementor great leeway, by writing the documentation in a way that prepares the programmer to expect that values returned by the readers might be shared or copied. She does this by adding the two basic rules shown on this slide to her proposed design.



Satisfied now that her design is useful, powerful, usable and implemented, the designer would proceed to actually publish it. Thus, the whole cycle we have observed in this chapter can be seen on this slide...

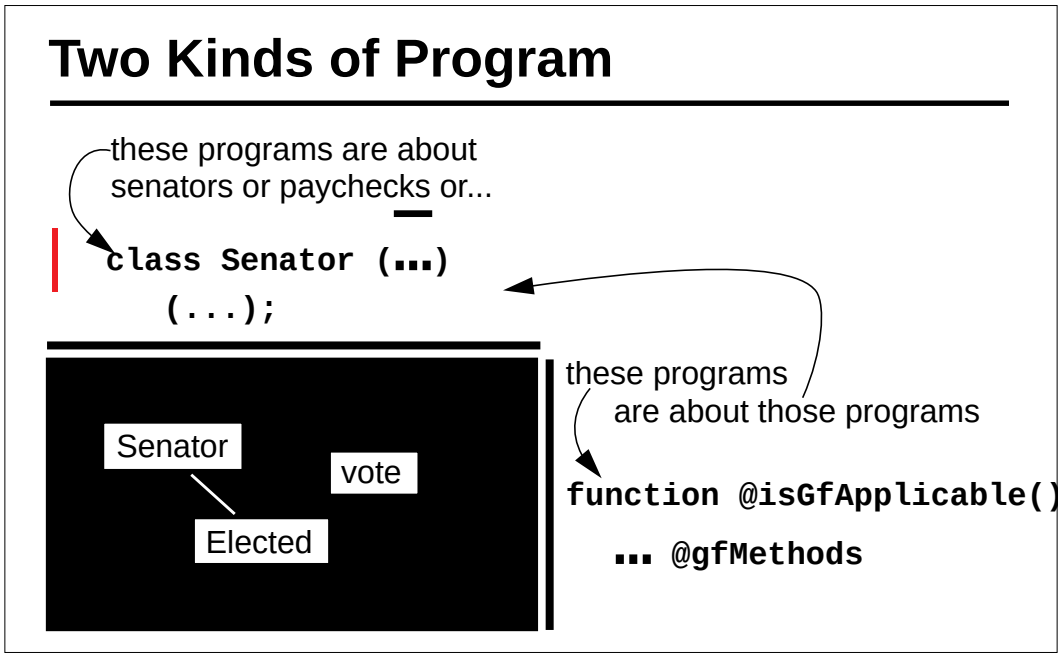
Of course, in a real design situation, the process would not be nearly so neat and tidy. There would be more instances of going back and fundamentally altering the design...

Also keep in mind that although this book presents the open implementation idea in terms of opening up the design of existing systems, there is no reason...



We now step back from observing the design process to make some general observations about the nature and the new functionality the designer has been developing. The perspective and terminology we will present is important because it provides a principled distinction between the new functionality and the original closed Tiny Objects. This distinction in turn will enable a principled separation between the new and old interfaces, and between the parts of the programmer's program that use the new and old interfaces. This will help to simplify the design as well as the user programs.

First notice that the Tiny Objects box is still mostly black, and that the original interface has not changed at all. Any old TinyObjects program will still run. What the object system designer has done is to *reify* some of the box's inherent internal information—the class, generic function and method descriptions—and to provide a clean "side door" interface to them. The term reify means to regard or treat an abstraction as if it had material existence. This is just what the designer has done with the class, generic function and method descriptions. Whereas previously they were an abstract or implicit part of the implementation that could not be manipulated, the programmer can now grab hold of them as concrete objects. Reification;Reflection

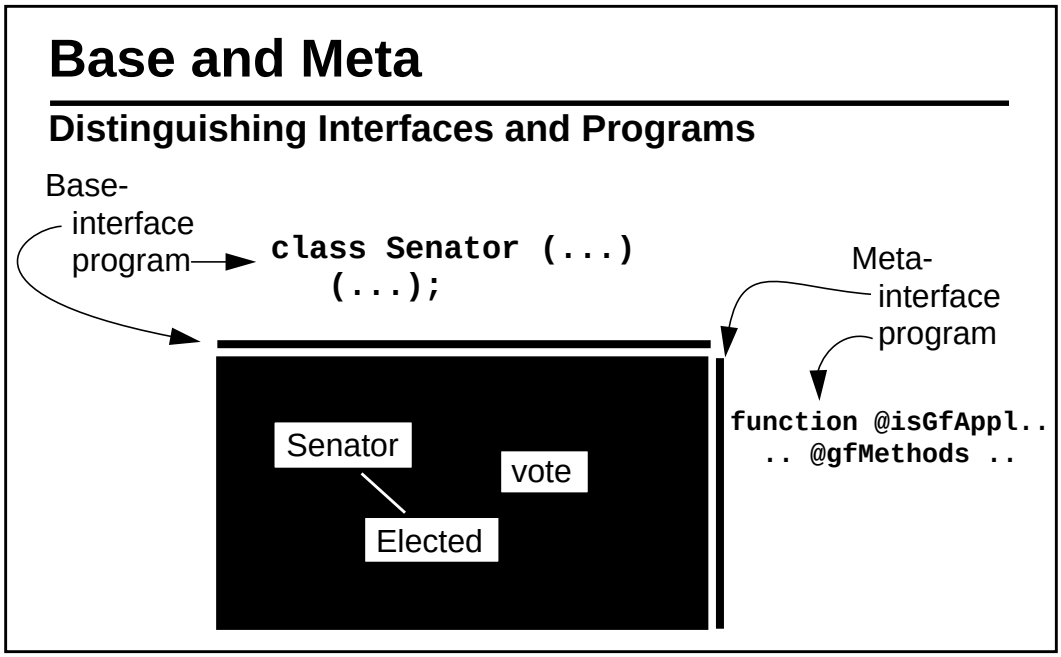


To draw out the difference between the old and new functionalities, it is important to focus on the kinds of programs that use each one.

Programs written using the original TinyObjects interface are concerned with senators and voting and the like. (Other programs will have paychecks or spreadsheets or some other domain as their basic subject matter).

On the other hand, programs written using the second interface are concerned with other programs, in particular the programs written using the first interface.

<$startrange>Base interface; <$startrange>Meta interface



To reflect this difference, the first kind of interface and program are called the *base-interface* and *base-program*, since they are about the basic subject matter for which the programmer is writing code (e.g., senators and their voting behavior). The second interface and program are called the *meta-interface* and *meta-program*, since they are one level of subject matter removed—they are about base programs and their interactions with the substrate.

Another way of thinking about this is that the objects manipulated by the first kind of program represent senators (or whatever the basic subject matter might be) whereas the objects manipulated by the second kind of program represent elements of the first kind of program.

Black-box:icon;<$endrange>Base interface;<$endrange>Black-box:partial opening;<$endrange>Meta interface

The Use of “@”

Names of meta-level functions begin with an “@” sign.

With this terminology in hand, we can now go back and add a piece of documentation to the specification of the new functionality, to explain the meaning of the “@” prefix naming convention.

<IndexCode>@ <Default Para Font> Sign convention; Metaclass; Metaobject protocol

Terminology

Objects that are the reification of inherent implementation components are called **metaobjects:**

class metaobjects represent classes
GF metaobjects represent generic functions
method metaobjects represent methods

The documented interface to the metaobject is called a **metaobject protocol**.

The classes of metaobjects are called **metaclasses**.

Class:metaobjects; Reification:method; Generic function:metaobjects; Reification:class

The particular meta-interface being developed here is object-oriented in that the data structures it exposes are objects. In such an object-oriented meta-interface, the objects that are the reification of inherent implementation components are called *metaobjects*, and the meta-interface itself is called a metaobject protocol. We use the term protocol, rather than interface because it better fits with the kinds of capabilities that will be introduced in later chapters.

So far, the designer has said nothing about the class of the metaobjects, but because that is coming up in the next chapter, we also introduce the term metaclass here. The class of any kind of metaobject is called a *metaclass*. While traditionally, the term “metaclass” has been used only for the class of class metaobjects, that restriction is best thought of as a historical accident. The more general use presented here works better for the more powerful metaobject protocols we will be presenting.

Note that the *object* in metaobject protocol indicates that the meta-interface is object-oriented. It is not a statement about base-interface. So, for example, one could have a metaobject protocol for a non-object-oriented programming language, or for a substrate that wasn’t a programming language at all (i.e. file systems, virtual memory systems, relational data-

bases etc). Similarly, one could have a non-object-oriented meta-interface for an object-oriented base-interface (although that might be less likely).

Metaobject protocol:designer

Introspective Protocol

An *introspective* meta-protocol allows examination of inherent implementation structure, mediated through an appropriate abstraction layer.

An introspective metaobject protocol consists of mechanisms for obtaining metaobjects and readers to examine their state.

The metaobject protocol developed so far is an introspective protocol. It provides the programmer with a principled way to examine selected implementation state, such as information about the names of a class's slots, or of its superclasses. The protocol is principled in the sense that it allows access to this information without forcing implementations to expose the internal data structures they actually use to represent it.

Any kind of meta protocol could be introspective in that sense. Metaobject protocols in particular work by requiring implementations to support metaobjects which programmers then use as sources of information about the implementations.

Introspection

Question: What happens if an optimized implementation compiles away the runtime dispatch of generic functions to speed things up? Then it won't be looking at the CPL anymore. I wrote a function **`@methodCallOrder`** which uses the CPL to analyze which methods will be called in order; won't it break?

Answer: No, that function will not break. The MOP specification requires implementations to provide a correct CPL. This does not mean that those implementations need to use the CPL for dispatch. But their observable behavior must be as if they were using the CPL.

Summary

- ☑ **Introspection**—examination of selected system internals.
- ☐ **Explicit Invocation**—enabling programs to explicitly invoke substrate operations, bypassing syntax in some cases.

  Intercession—metalevel specialization that adds to or modifies existing features of the base object system:
  - ☐ Adding State
  - ☐ Adding Behavior
  - ☐ Modifying Behavior
  - ☐ Adjusting Performance

Put yourself in the shoes of a TinyObjects programmer, who needs to write procedures like @isGfApplicable or @findDiamonds. How difficult would it be to write these without the introspective MOP developed in this chapter? Try to implement the procedures @isGfApplicable and @methodGenealogy of page 85 and page 87 without it.

Now ask yourself the question, is it worth extending TinyObjects with this new functionality? Do the potential payoffs for programmers outweigh the potential costs for TinyObjects implementors? On what grounds did you base your argument? Esthetic, economic..?

The following material will show how other kinds of programmer needs can be addressed by providing different kinds of access. For programmers needing nothing beyond the introspective capabilities developed in this chapter, this would be a coherent place to stop. The current MOP is consistent. It meets a given set of programmer needs, and can be implemented robustly and efficiently.

This is an important point: MOP design is not all-or-nothing. MOPs need only have enough power to satisfy actual programmer needs. Subsequent chapters will show how classes of programmer needs beyond introspection can be satisfied through more advanced degrees of opening substrate implementations. This will be accomplished through commensurately more sophisticated MOP design techniques.

Chapter 4 But I Wish I Could Get At…

The programmer can now enjoy "voyeuristic pleasures." He can look into the substrate through well-defined portholes, and he can use the resulting information towards filling his needs.

He cannot yet change how the substrate behaves. While this will be one of the ultimate goals, there is an intermediate step in between the introspective capabilities introduced in the last chapter and the ability to modify substrate behavior. Notice that at this point the programmer cannot even *effect* anything in the substrate—he cannot cause events to happen. This chapter begins to provide that additional power.

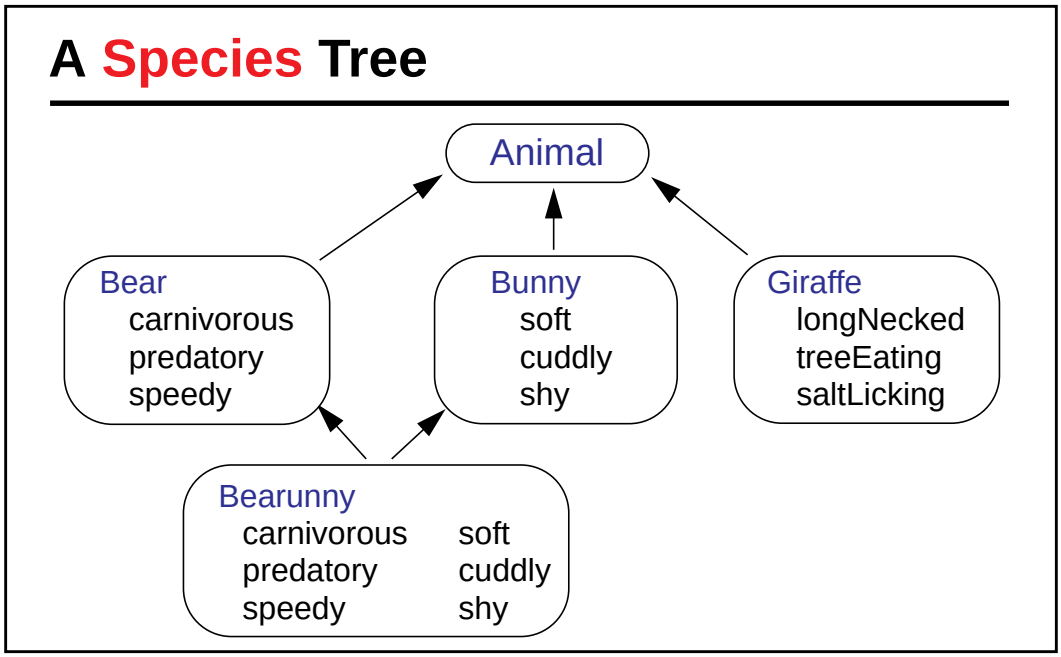
Going to the Zoo…

I am working on a computer game about cross-breeding of animals. The different kinds of animals are displayed on the screen. Children can combine them and add features, building a genealogy of fantasy animals. When a child points to a kind of animal, I want to create one each of all the kinds of animals that have been bred from it.

I want to implement it with TinyObjects.

The next several pages (to page 112) show the programmer explaining how their system works, leading up to a description of the problem they are having on page 113.

Examples:object creation, class known at runtime



Graphically, the program looks like this:

Each species of animal is displayed on the screen with the attributes of that species. The arrows indicate descendance of one species from another.

When a child clicks on a species, the system should create one animal for that species and every species below it.

Because this structure is similar to a multiple-inheritance class graph, and because TinyObjects has now been enhanced with the new funtionality from the previous chapter, the programmer wants to use TinyObjects to implement their system.

Species Are Represented as Classes

```
class Animal () ();

class Bunny (Animal)
    (soft, cuddly, shy);
            ⋮
class Bear (Animal)
    (carnivorous, predatory, speedy);

class Bearunny (Bear, Bunny)
    (...);
```

The programmer has chosen to model species as classes. The attributes of each species are modeled as slots.[1]

By doing this, a program that generates the picture on page 111 is similar to programs from Chapter 3. This is just navigation through the class graph (using **`@classDirectSubs`** and **`@classDirectSupers`**), finding the names of classes (using **`@className`**) and finding out about available slots (**`@classDirectSlots`** and **`@classSlots`**).

Similarly, the programmer's idea is that when a child clicks on a species, their program would first access the metaobject of the selected class, next use it to find all subclasses and then create an object for each of those subclasses.

1. The designer recognizes that this is a distortion of the concept of slots, since they are meant to hold state in each object of a class, not attributes that are valid for the entire class. We will return to address this problem in Chapter 5.

I'd Like To…

```
function makeAnimals (c) {
  var result = list();
  foreach (sub, @allSubs(c))
    extend(result, new(sub));
  return result;}
```

But this won't work because **new** needs the class name to be known at compile time.

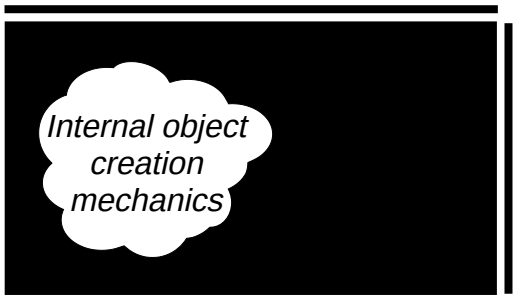
So far so good. But on this page we see the problem the programmer is having.

Here is the central piece of the code the programmer tried to write. It makes use of the **@allSubs** function from page 72. (Recall that this function took a class and returned a list of all its subclasses.) The **makeAnimals** code above just runs through that list, making an object from each subclass.

Or tries to, but the problem is that the primitive **new** requires the name of the class to be fixed at compile time. When TinyObjects was designed, the designer assumed that the name of classes to be instantiated would always be known at the time programs were written. This limitation of the base interface prevents the programmer from writing their code this way. Note, however, that implementations certainly do create objects at runtime. It seems, therefore, that the programmer is not asking for anything impossible or even particularly difficult, since the meta-interface introduced in the previous chapter is limited to introspective facilities, it is not powerful enough to help out either.

Finding Relevant Functionality

```
class Bunny(Animal) ...;
new Bunny()
```



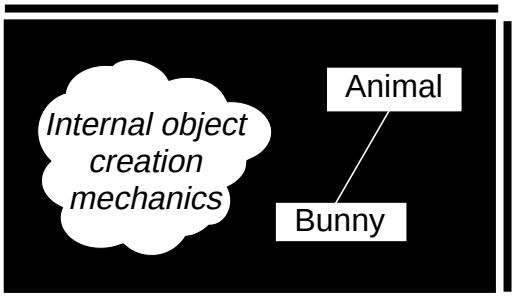
Faced with this problem the designer now returns to the question of what it would mean for TinyObjects to provide such functionality. Again, she begins by wondering what that would be like from the TinyObjects implementor's perspective.

As shown in the figure, the current situation from the implementor's perspective is that the internal object creation mechanism only has one interface through the **new** primitive which only works with class names fixed at compile time.

But the object creation mechanism the programmer wants access to inherently does exist inside the implementation.

New Interface to Object Creation

```
class Bunny(Animal)...;
new Bunny()
```




```
@new(@findClass("Bunny"))
```

The programmer has already given the designer the seed of an idea for the new interface to object creation—through the meta-interface using class metaobjects.

She now considers adding a new function to the meta-interface, called **@new,** that takes a class metaobject and returns an object of the corresponding class.

Note that just as in the previous chapter the designer is not thinking about the details of any one TinyObjects implementation. Instead, she is thinking at a middle-level of abstraction—between the details of specific implementations and the documented interfaces—about how any TinyObjects implementation inherently has to work.

Reification:object creation

The designer has already reified classes to provide limited access to information about classes that is maintained within implementations. But this **`makeAnimals`** programmer now exposes another problem. Sometimes a substrate black box hides not just *state* that would be useful for the programmer to have access to, but also pieces of *functionality* that are inherently present in every implementation, would be useful to invoke from the outside, but are not accessible from the base interface explicitly. In the current example, this functionality is the object creation mechanism underlying the **`new`** primitive. In the figure, this is represented as a cloud.

Allowing the programmer to invoke this functionality explicitly would solve his problem.

Making Objects

Another Protocol Expansion

Given a class metaobject, the programmer may create objects of the corresponding class by calling **@new**. This meta-level function operates directly on class metaobjects to create objects. Like **new**, it accepts a list of initargs. The following calls are equivalent:

```
@new(@findClass("Bunny"), list())
new Bunny()
```

Here is the designer's latest proclamation, signing this capability into law. **@new** is simply a function the programmer may now call with a class metaobject to have an object of the corresponding base level class created. From now on the programmer is allowed to write programs which create instances of classes from class metaobjects. That is, write code that first computes what class it wants to make an instance of and then makes the instance. (As opposed to the class name being hardwired into the code.)

Note that this piece of protocol is qualitatively different from those in the previous chapter, which enabled only introspective capabilities. This piece allows the programmer to cause action in the substrate—in this case the creation of objects.

Reification:object creation; Object:creation

Making Those Animals

```
function makeAnimals (c) {
 var result = list();
 foreach (sub, @allSubs(c))
    extend(result, @new(sub, list()));
 return result;
```

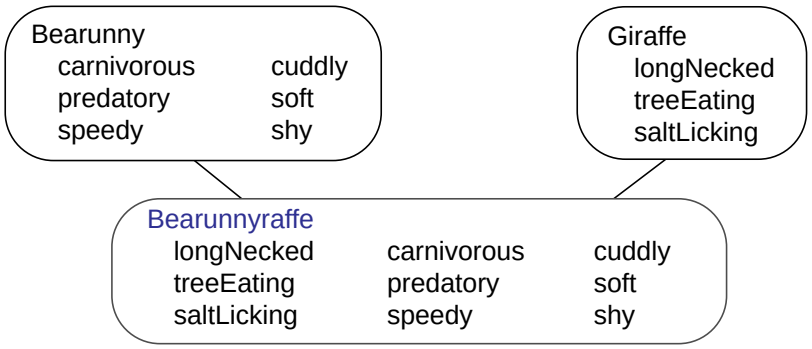
Now the problem with the program of page 113 can be resolved. Just as in the previous chapter, the designer must test the new functionality to see that it meets programmer needs. The code above uses the **@new** interface to object creation functionality to enable runtime object creation.

So, the new functionality does indeed solve this kind of programmer problem.

Examples:Object creation, class known at runtime

How About New Species of Animals?

I want the kids to create new species of animals by clicking on animal classes. How can I implement that?



But this does not solve all the programmer's problems.[1] He would also like to allow children to create new species of animals. When they click on Bearunny and Giraffe, a new species of animal should be created, with all the inheritance done right.

<$startrange>Class:customization; Examples:class creation, specifications known at runtime

1. Keep in mind that these frequent shortcomings of the MOP are due to the fact that for the purpose of teaching the material, TinyObjects was initially designed (and implemented) with no MOP at all, so that we could use its shortcomings to introduce new material. In a real design, at least some of these needs would be anticipated and handled properly by the initial MOP that is delivered with the substrate.

Analysis

The programmer implementing this wants the ability to define new classes on the fly:

```
@newClass("Bearunnyraffe",
            list("Bearunny", "Giraffe"),
            list())
 --> #Bearunnyraffe
```

New class

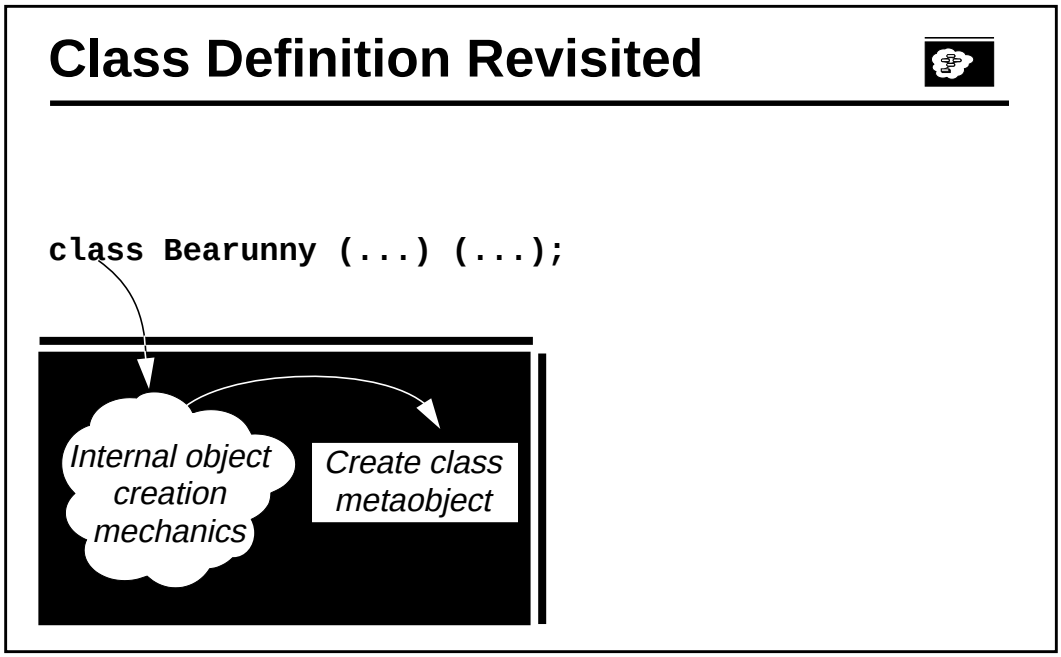
```
@new(@findClass("Bearunnyraffe"), list())
 --> #<Bearunnyraffe 1>
```

Object of new class

But only **`class`** knows how to construct classes.

The designer's first step is to analyze what the programmer needs.

In a sense, the programmer is asking for access to a function with the functionality of **`@newClass`**. The function arguments would be similar to those of **`class`**: a class name, a list of superclasses and a list of slot names.



Class:definition

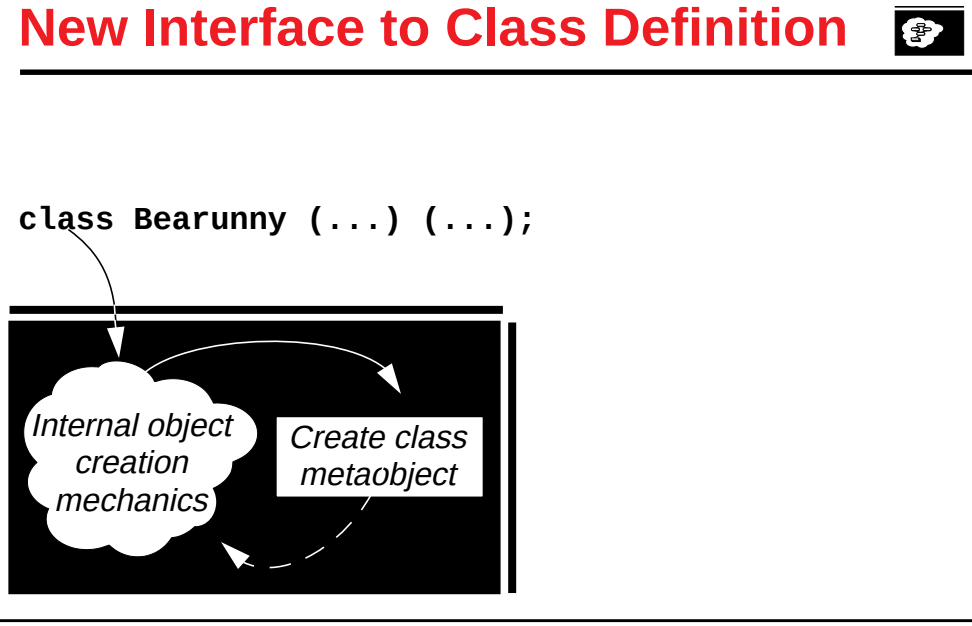
Turning to the implementor's perspective, the designer draws this picture of the current situation. It shows that only the base interface can be used to define classes. The meta-interface can be used to inspect the existing class definitions. It also shows that the details of executing a class definition statement are internal to a particular implementation, which means that they are free to differ. This is represented by the cloud around the class creation mechanics. The solid arrow out of the cloud indicates that all implementations have to create a class metaobject as required by the introspective protocol of Chapter 3.

The fact that the internal mechanics of defining a class also creates corresponding class metaobject gives the designer an idea for a coherent meta-interface extension: what if this also worked in reverse? What if the programmer were allowed to create a class metaobject, and this act of creation caused implementations to create a corresponding base class?

Note, however, that the protocol fits naturally into the model the meta-level programmer has built up in his mind so far. In reality, class metaobjects are merely representations of the parts of an implementation's state that are relevant to base level classes. Implementors are free to make the metaobjects more or less coupled to the internal structures that actually implement classes. But as long as the programmer uses only documented operations, he can think of metaobjects as really being the classes they represent.

If the MOP designer now allows the programmer to create class metaobjects and to have that creation at the representational level actually trigger the creation of base level classes at the implementation level, then the programmer gets to work that much more comfortably within the world the MOP designer is creating for him. The MOP designer is turning the purely introspective interface into an "effective" interface, in the sense that as the programmer manipulates the components of the interface, he is effecting events in the implementation.

New Interface to Class Definition



This slide shows the designer speculating about what it would be like to allow programmers to define classes at runtime by creating class metaobjects. The dashed arrow indicates the “inverse causality” under consideration.

The designer likes the way this is taking shape, but right away she sees that several questions must be answered, including: what is the name of the documented metaclass programmers can instantiate[1] and what initargs does that class take?

1. The slide assumes it will be **`@class`**.

Programmers Creating Class MOs

`@Class` could be the documented metaclass name.
Initargs for making instances of **`@class`** could be:

`Name:` **`“Bearunny”`** ← String

List of class metaobjects ↙

Direct supers: **`list(@findClass(“Bear”), @findClass(“Bunny”))`**

Direct slots: **`list()`** ← List of slot names

With respect to the documented name of the metaclass, the designer thinks the simple **`@class`** would be fine. With respect to the initargs, the problem is only slightly harder. Clearly the initargs must include everything that is essential to the definition of a class (i.e., everything included in a use of the **`class`** primitive).

But, because this is a meta-level operation the designer thinks that the form of the initargs should be consistent with the other operations in the MOP.

Making Custom Classes (1/2)

```
function @newClass(className,
                   directSupers,
                   directSlots) {
 new @Class(className,
            @findClasses(directSupers),
            directSlots);}

function @findClasses(classNames){
 var result = list();
 foreach (name, classNames)
    extend(result, @findclass(name));
 return result;}
```

Testing this proposed new functionality here is the function the programmers needed to implement his animal creation program. Since **new** requires a list of class metaobjects, the helper function **@findClasses** is introduced to produce a list of class metaobjects from a given list of class names.

The new functionality seems to be what is needed so the designer decides to go ahead and make it official.

Documenting Class Making

- Programmers may instantiate **@Class** to create class metaobjects.
- This will cause the definition of a corresponding base level class.
- Required initargs for creating class metaobjects are:

name	*a string*
direct supers	*a list of class metaobjects*
direct slots	*a list of strings*

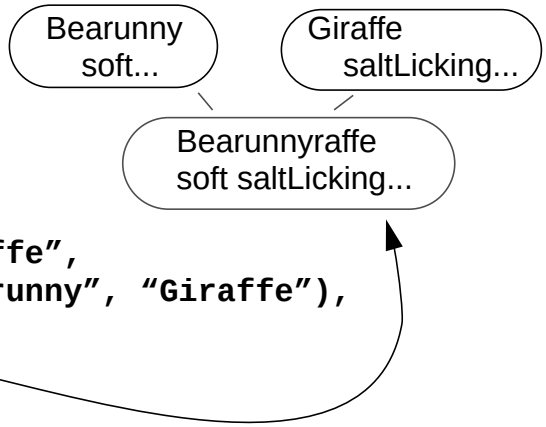
Class:creation through explicit invocation; Class:definition; Metaobject:creation

Here is the documentation of this new meta-level funtionality that makes it official. Because programmers are now able to make (meta-)objects of class **@Class**, and because this action causes base level class definition, they can now define base level classes even when their names, superclasses, and slots are not known at compile time.

Question: Is the designer making guarantees about how the implementation actually works? Is she saying that every class actually is an instance of **@Class** on the inside?

Answer: No, the designer has not required that classes really have to be instances of **@Class**. What she has promised about all implementations is that (i) the class definition expression **class** will cause the proper class metaobject to be created and, inversely, (ii) that meta-programs calling **@new** on **@class** will cause a new base level class to come into existence. Implementations remain free to have any additional datastructures they want as part of classes. They can maintain information about classes that is not required by the protocol, they can use any data structures they want for maintaining classes internally. They must only (i) ensure that the meta-level abstraction always reflect what is happening in the implementation and that (ii) meta-level operations truly are first-class operations that have their intended effect in the implementation.

Making Custom Classes (2/2)




```
@newClass("Bearunnyraffe",
            list("Bearunny", "Giraffe"),
            list())
 --> #Bearunnyraffe
```

Here is a graphical view of how a call to **`@newClass`** will work. Of course, you're glossing over how you manage to come up with a name that makes sense.

Class:customization; Examples:class creation, specifications known at runtime

GFs and Methods Are Analogous

- The programmer can make objects of class **`@Gf`**. The initargs are *name* and *arglist.*
- The programmer can make objects of class **`@Method`**. The initargs are *specializer* and *function.*
- The programmer can call **`@addMethod`** to attach a method to a generic function.

All these will cause definition of the corresponding base level components.

Iterating on her work, the designer decides to make the analogous extensions for generic functions and methods. These extensions allow programmers to create generic function and method metaobjects, promising that through these acts of creation, the actual generic functions and methods will be created by the underlying implementations.

To save space, the work of developing the design is elided. Only the final documentation is shown above.

Generic function:addition of methods; Generic function:creation through explicit invocation; Generic function:customization: Method:addition to generic function; Method:creation through explicit invocation; Metaobject:creation

GF and Method Creation

```
new @Gf("makeNoise", list("animal"))

@addMethod(@findGF("makeNoise"),
           new @Method(@findClass("Bear"),
                       @findFunction("bearGrowl")))


@applyGf(@findGf("makeNoise"), list(smokey))
```

Here is an example of how programmers can write programs that create generic functions and methods for them, and how programs can then invoke those generic functions. The first **new** expression creates a generic function metaobject. The two arguments, **"makeNoise"** and **list("animal")** represent the name and argument lists of the new generic function, respectively.

The **@addMethod** expression retrieves the new generic function metaobject, creates a new method metaobject specialized on the **Bear** class, with the function **bearGrowl** as its body, and then adds the method to the generic function. The **@findFunction** takes a function name and returns a pointer to that function.[1]

Finally, a call to **@applyGf** is used to invoke the new generic function and method with a **Bear** object (**smokey**).

1. Assume the function **bearGrowl** has already been defined.

Explicitly Invoking Operations

Direct access to the metaobjects and the operations that implement the object system features allows the programmer to:

- Bypass the normal user interface.
- Trigger actions not accessible via the base level interface.

A metaobject protocol which provides this capability is called an **explicit invocation meta-protocol**.

Stepping back from the designer's work, we call the new functionality developed in this chapter an *explicit invocation meta-protocol* because it gives the programmer explicit access to internal mechanisms that are otherwise hidden partially or completely by the base interface.

In some ways, explicit invocation meta-protocols can be characterized as allowing the programmer to bypass the syntax of the base interface. (**@new** is a good example of this). But, the ability to explicitly invoke substrate operations can be more powerful. It can also permit invocation of operations which are not available at all with the base interface. For example, the meta-interface for a virtual-memory system might provide explicit access to page-in and page-out operations in order to allow programmers to optimize the virtual memory performance of their programs.

Other Uses of Explicit Invocation

- Variety of browsers, editors and debuggers
- Transforming programs from one language to another
- Computer learning: generating new classes representing combinations of facts

By now, other uses of explicit invocation protocols are probably evident. Apart from the obvious applications of browsing, editing and debugging, some other examples are program transformations or applications in the area of knowledge representation.

Exercise: What other uses can you think of for the TinyObjects MOP extensions developed in this chapter?

Exercise: Can you think of additional explicit invocation meta-protocol that should be added to TinyObjects? What about giving programmers control over the garbage collector?

Exercise: Some virtual memory systems do in fact allow programmers explicit access to mechanisms like page-in and page-out. What other explicit invocation meta-protocols can you find in existing virtual memory systems? Extend your paper design of a MOP for virtual memory form the previous chapter to included appropriate explicit invocation support.

Criteria for MOP Design

Standards New Capabilities Must Meet

- Usefulness

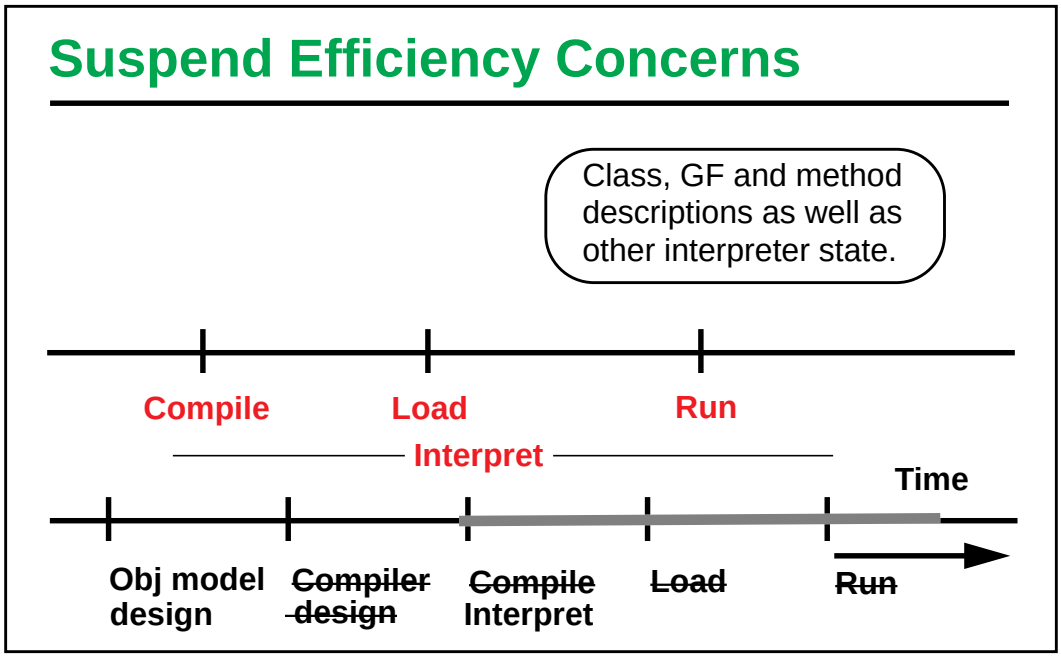
  What is the new capability good for?
  Does it leave programmers with a coherent system?

- Feasibility of implementation

  Can it be implemented robustly?
  Can it be implemented efficiently?

In order to extend the protocol, MOP designers must show how their new extension would be useful, and how the extension cleanly builds on existing notions. In this chapter, for example, the designer builds on the existing metaobject concept by allowing programmers to create those metaobjects. This helps keep the system coherent.

Regarding implementability, we first note that, again, MOP designers are not asking implementors to provide any new internal functionality. All they ask is that programmers provide operations that allow programmers to trigger that already existing funtionality. Let us acknowledge a question that may come up for readers familiar with compiling environments.



When coming from a compiling environment, one is used to pieces of information and certain actions not being available at runtime. Earlier, in Chapter 3, information about classes, generic functions and methods was made available at runtime. Now the ability to make classes is also shifted from compile time to runtime. As mentioned previously, this kind of "time-shifting" is familiar from mechanisms such as compilers leaving symbol tables accessible for debuggers to use at runtime. Moving actions normally performed at compile time to runtime is a bit more unusual. But even that is done to some extent by incremental compilers. We prefer to defer details of this capability until Chapter 7 where we will discuss techniques for managing the shift of information and actions in time.

Meanwhile, it is best to think a bit more in terms of an interpreter model where information and actions are accessible at all times.

Robustness

The only danger would be for programmers to call the new operations in the wrong order.

If the operations are available through the base level anyway, this is not a new complication.

Runtime usage checking might be necessary.

For the examples we presented here, the ordering of explicit operation invocations is no more or less a problem than they already were. Programmers could create methods before defining generic functions for them, or they could try to specialize methods on classes that were not yet defined, even before any metalevel interface came into existence. Any given implementation will be no more or less resilient to such activities now that the MOP presented in this chapter has been introduced.

When operations which had not previously been accessible from the base interface at all become available for explicit invocation, more care may be needed to avoid introducing new failure modes. Note, however, that any additional checking an implementation might perform to avoid such failures can be bypassed when the implementation performs its own invocation of these operations.

This brings us to the question of performance, which, for now, is no more interesting than the question of robustness.

Explicit invocation:robustness in; Robustness:explicit invocation of

Performance

Other than, possibly, time shifting, nothing new is required of the implementation.

Call sequence error checking needs to be done for calls from outside only.

Time shifting techniques covered in Chapter 7 may be required if operations were previously only accessible at compile-time. Other than that, the implementations are asked merely to allow explicit triggering of operations that were being performed anyway. Performance is therefore not a major concern to us right now.

Anything Else to Try?

We have been talking about how allowing programmers to directly manipulate "real" meta-objects can help with a number of problems where the existing user interface to a substrate is deficient.

Did the discussion bring up any similar sorts of problems in your mind?

Summary

- ☑ **Introspection**—examination of selected system internals.
- ☑ **Explicit Invocation**—enabling programs to explicitly invoke substrate operations, bypassing syntax in some cases.

  Intercession—metalevel specialization that adds to or modifies existing features of the base object system:
  - ☐ Adding State
  - ☐ Adding Behavior
  - ☐ Modifying Behavior
  - ☐ Adjusting Performance

In this chapter we watched as the designer added two simple but important features to the TinyObjects MOP. Whereas the introspective MOP in the previous chapter only supports inspecting meta-level states, the explicit invocation MOP of this chapter allows the programmer to cause action in the substrate.

With both introspection and explicit invocation protocols there has been a sense that the new funtionality has been in some sense, “there all along.” All the designer needed to do was come up with a clean abstraction and address certain efficiency issues.

In the next few chapters we will see the designer go much further. Her approach of giving programmers access to that which would previously have been considered internal to TinyObjects will cause more significant changes in how TinyObjects is implemented, and give programmers correspondingly more significant power. In the next chapter the designer will add MOP support that allows the programmer to add certain kinds of new features to TinyObjects.

Chapter 5 But I Wish It Had This Extra Feature…

So far xxx

The cost to implementors has been modest, they have only been asked to expose information and functionality that is inherently part of any TinyObjects implementation. Moreover, because the protocol designer was careful to ensure that the documented MOP is a clean abstraction of implementations, implementors have appropriate leeway within which to craft their specific implementations.

The next two chapters explore a new kind of problem, in which the programmer wants more than enhanced access to what is already there, instead he wants TinyObjects to provide some *additional or different* functionality. Rather than asking the programmer to "code around" the deficiencies in TinyObjects, the MOP design will take the bold step of

designing a protocol that allows the programmer to actually *change the behavior* of TinyObjects.

This is where the decision to use object-oriented programming in the meta-protocol—to have a meta*object* protocol—will come into its own. It will make it possible to ensure that programmer changes to TinyObjects behavior have appropriately localized effect. This scope control is a key element in ensuring that metaobject protocols are an appropriate tool for software engineering, instead of a fragile and dangerous hack.

I Am a Class Library Contractor…

…and I need the ability to record the author of each class and access that in the runtime image.

How can we do this?

Neither the ability to inspect class metaobjects introduced in Chapter 3, nor the explicit invocation capabilities of Chapter 4 will help the programmer cope with this problem. Without improvements to the evolving metaobject protocol, he would have to resort to techniques such as shell scripts and makefiles, in the case of Unix systems, or comparable tools in other environments. While such solutions are workable and even common, they tend to be complex, fragile and non-portable.

Exercise: Using the facilities available in your environment, implement this functionality for yourself. You will want to design a special syntax for annotating a class definition with the author, and then have a preprocessor that goes through a file collecting the author information and storing it in a database file. Your compile driver will need to be sure to maintain coordination between the database file and each set of binaries.

The intuition behind this example is that since TinyObjects is already maintaining so much other information about a class (superclasses, slots etc.), it might be easier to ask it to maintain the author information as well.

<$startrange>Class:adding information to;Class:customization; Examples:variables on classes; <$startrange>Metaobject:adding information to

Analysis

This programmer wants to write something like this:

```
class Phone (CommDev)
     (number)
 author: “Fred”;

@classAuthor(@findClass(“Phone”)) --> “Fred”
```

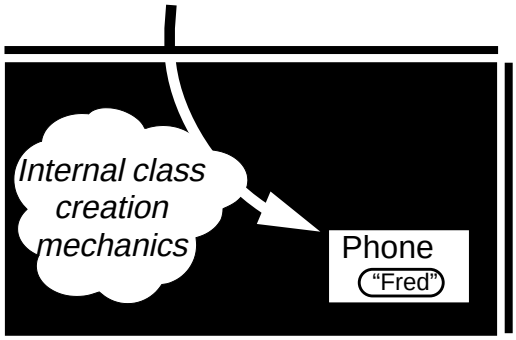
To address this request, the MOP designer sits down with the programmer to collaborate on a solution. In doing so, she employs a useful trick to organize the dialogue: they begin by focusing on what the desired functionality would have looked like if it had been part of TinyObjects in the first place. They extend the syntax and behavior of the object system on paper, until they are satisfied that they could cleanly use the new functionality if the object system had the respective extensions.This approach allows them to get a clear sense of what the programmer really wants, before worrying about what new (or existing) metaobject protocol is required to support it.

In this case, the author information would ideally go right into the class definition, using a syntax that allows the programmer to specify an author only when he wants to. An accessor could then extract it from the class’s metaobject anytime.

Programmer Injecting State

```
class Phone (CommDev)
    (number)
  author: "Fred";
```

Programmer wants to cause more information to be kept in class metaobject



Here is a graphical view of presenting what the programmer is asking for. By adding the author information to the definition of his **Phone** class, he wants that information stored in the corresponding class metaobject.

A First Attempt

Proposal: The MOP designer adds an author slot and appropriate initialization mechanisms to class metaobjects.

Bad because:

- Not general enough: what if more information is needed later?
- No control over scope of modification: all classes would have author information.

One possibility would be for the MOP designers to add the concept of class authorship directly into all class metaobjects and to provide a respective reader and initialization mechanism. But that would be inappropriate, because it would solve much too narrow a problem. The next day, other programmers might want to add a creation date or other information to class metaobjects which would then require renewed intervention of MOP designers and object system implementors.

Another problem with this suggestion is that *all* TinyObjects classes would have this author information, even the ones written by programmers who do not need it. There would be no way to limit the scope of this change. If more such information were added over time, the implementation would get more and more bloated, forcing all programs to pay for all new pieces of information maintained.

Searching for a More Subtle Touch

Resulting conclusions:

Only *some* class metaobjects should carry extra info

⬇

More than one kind of class metaobject is needed

⬇

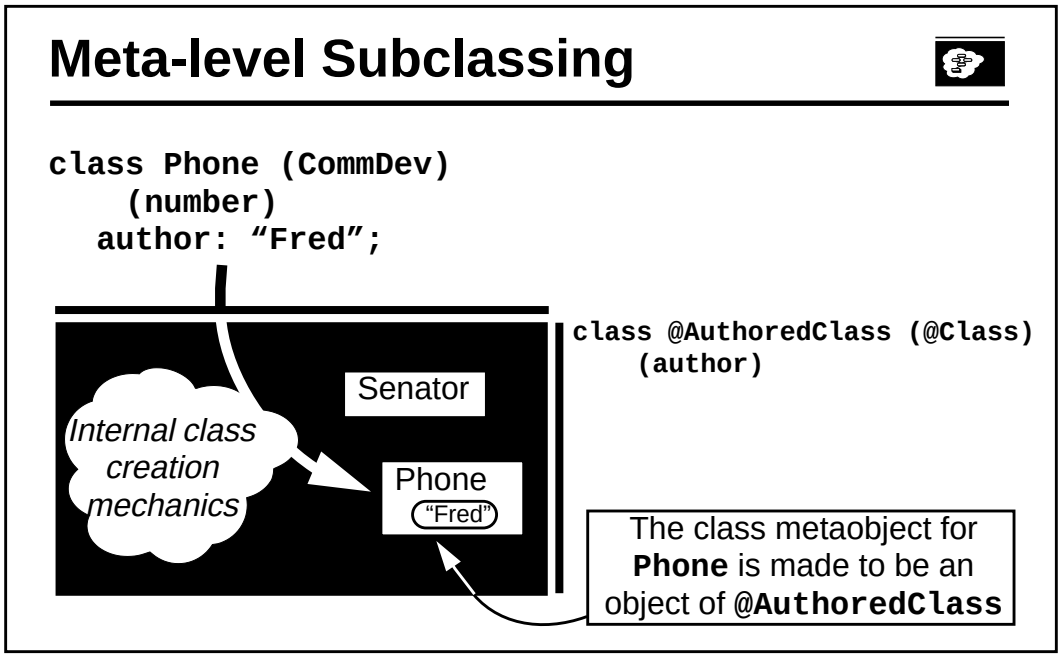
More than one kind of (meta-)class of class is needed

⬇

```
class @AuthoredClass (@Class)
        (author);
```

In essence, what is needed is class-specific storage that the programmer can control. The problem is to find a meta-interface extension for providing such a facility that is powerful enough to do the job, yet fits into the model of the object-oriented meta-interface developed so far.

The programmer already has the notion that he can make class metaobjects by instantiating **`@Class`**. The MOP interface could build on this notion by allowing the programmer to subclass **`@Class`**. He could add slots in such a subclass to hold any additional information he might want to maintain. One example could be an **`author`** slot.



Classes that were to retain author information would have an **@AuthoredClass** metaobject representing them. Other classes without author information would continue to be represented by objects of **@Class**. That approach would provide the desired control over the scope of the modification, and it would blend well with the programmer's understanding of the meta-interface being object-oriented.

The mechanism at the meta-level is clear now. The figure above still has the arrow from the base level class definition to the **Phone** metaobject. How would that arrow work?

Metaclass:subclassing

Defining an Authored Class

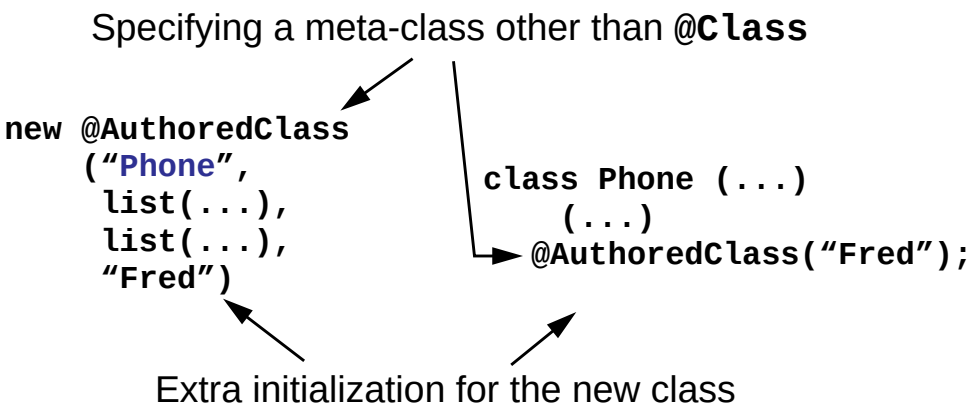
To build a base level class **Phone**, authored by Fred:

```
new @AuthoredClass("Phone",
                   list(@findClass("CommDev")),
                   list(),
                   "Fred")
```

The programmer would define authored classes as shown here. Instead of making his TinyObjects class be described by a regular **@Class** metaobject, he would specify that he wants an **@AuthoredClass** kind of TinyObjects class. In addition to the usual class name, list of superclasses and list of slots, the programmer would provide a fourth argument to the creation of such authored TinyObjects classes: the class creator's name.

But there is a drawback to this solution: the base-level programmer can no longer use the convenient **class** declaration for making classes, because that primitive does not allow for the fourth argument. Being restricted to explicit invocation is unpleasant. MOP designers need to update the user interface syntax appropriately.

How to Use the UI for Class Making



Here is a new syntax for the **`class`** statement which allows an additional argument that determines the meta-class to be used in making the Tiny-Objects class being defined. The figure shows the correspondences between the explicit invocation way of creating a new class and the friendlier **`class`** interface to this operation.

Notice that the programmer does not quite get the syntax first proposed on page 142. The author is not introduced by a keyword. Instead, a fourth argument to the **`class`** primitive takes the non-standard class meta-class to be used for making the new metaobject. The author information is then provided in the initialization argument which is passed along to **`new`**.

The MOP designer is now almost ready to make the new piece of protocol official. But there are still a few odds and ends to take care of.

Initialization of Author Information

```
initializer @AuthoredClass (c,
                            className,
                            directSupers,
                            directSlots,
                            author) {
  c.author = author;}
```

Author information would be seeded by means of the above initializer. The information provided in the initialization argument to the **class** primitive would make its way to the **new** invocation that produces the class metaobject. That instantiation process would run this initializer.

The only piece missing now is a way to access author information in class metaobjects. Here is the code for that.

Access to Author Information

```
generic @classAuthor (class)

method @classAuthor (@AuthoredClass c) {
   return c.author;}
```

And for safety:

```
method @classAuthor (@Class c) {
   return "";}
```

A generic function, **@classAuthor**, would allow author information to be retrieved via class metaobjects. As usual, a corresponding generic function would be written for setting and changing the value.

One matter of safety must be considered in this context. Programmers will get used to asking classes about their author. Sooner or later they will try to obtain author information from class metaobjects that are not of the **@AuthoredClass** lineage. Such a mistake should be handled gracefully, or at least explicitly. The code shown here just returns an empty string. It could instead raise an error condition.

Having ensured that the new meta-level capability of subclassing would work for the programmer's request, it needs to be documented as part of the protocol.

<$endrange>Class:adding information to; <$endrange>Metaobject:adding information to

Supporting Extension

- Programmers may define subclasses of **@Class.**
- The class definition primitive **class** supports an option which selects the meta-class of the new class.
- **class** passes additional options as initargs to **new**.

Here is a summary of what programmers can now do with the above new piece of protocol: they can build new class meta-classes which add state to the TinyObjects classes they produce. They can thus build new kinds of TinyObjects classes. Since the mechanism uses inheritance, it is additive: only the new piece of state is added. The remaining class creation mechanism is unaffected. The use of inheritance further ensures that these extensions affect only the classes of programmers that explicitly request them.

Question: Can programmers have authored kinds of classes inherit from default kinds of classes? Or the other way around: can programmers write un-authored subclasses of authored classes?

Answer: Yes. There is nothing that prevents programmers from doing this. And in this case that is probably fine. It could make sense for an application to have **CommDev** classes be authored, but not all the different specific devices below that class. Such a decision would be up to the extension designer, that is the programmer who knows the application. If they wanted to force inherited classes to be authored if the parent was authored, they would need to wait till later when techniques will be introduced that allow programmers to change what happens when a class is initialized.

<$endrange>Class:customization; <$endrange>Examples:variables on classes; Metaclass:initialization; Metaclass:specialization; Metaclass:subclassing

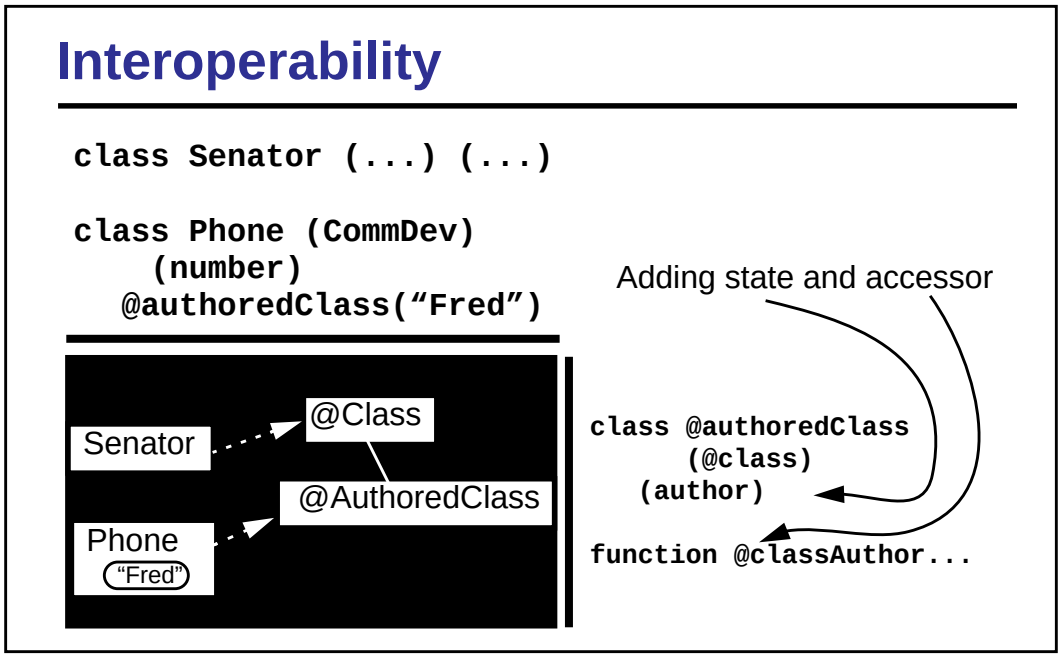
Intercession

The process of programmers adding state or modifying operations of a substrate is called ***intercession***.

Examples so far were limited to programmers **adding** behavior or state.

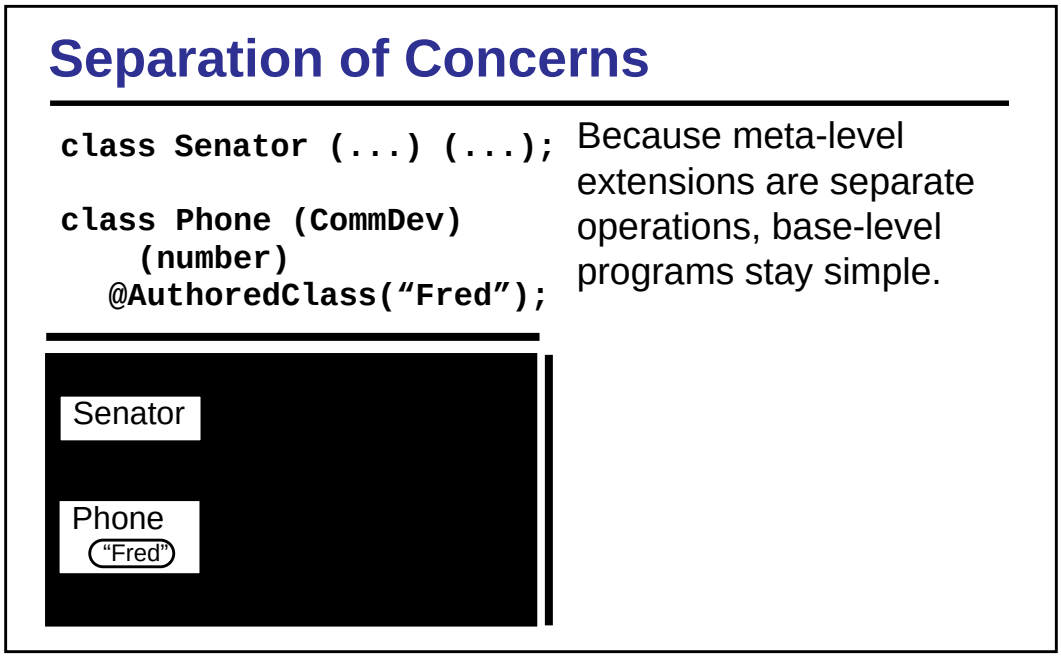
Notice that MOP designers have now taken a step qualitatively different from the earlier steps in their protocol development. The first step provided representation of selected implementation state. The second provided functions within the framework of this representation for programmers to call when they needed to trigger implementation operations that were previously hidden or only indirectly invoked. This third step again works within the framework of the object-oriented meta-level interface. This time the change is that programmers can add state to that interface.

Intercession is the term for allowing programmers to "get into" substrate operations in a controlled manner. Like introspection and explicit invocation, intercessory techniques solve a whole class of problems. The authored class example was a first, 'mild' form of intercession which limits itself to adding state, rather than modifying how the implementation does its job.



The authored class example illustrates two advantages of the approach to flexibility that is being developed in this book. This black box picture shows the first of these advantages.

Note that the two kinds of TinyObjects classes, **Senator** and **Phone** are peacefully living in one address space and can both be used in a single program. They are *interoperable*. This is an important point in that it shows how object-oriented technology is used here to insulate substrate extensions from each other. This insulation addresses two fundamental problems with tailorability: (i) it allows the addition of features to the substrate without requiring all programs to pay the additional cost, even if they do not make use of those extensions. And (ii) it allows programmers to use in one address space sets of components that are alike but have been extended differently. The insulation of substrate extensions therefore keeps use of the substrate manageable. Systems that are only globally tailorable, are often very hard to use by multiple people, because interface customizations diverge. By allowing programmers to explicitly control the scope of extensions, it is possible to customize substrates to suit different tasks, yet still have the default configuration available.



This figure illustrates the second important point about this open implementations technology: once the meta-level extension has been completed, all irrelevant portions of the substrate box can be "blacked out" again. The details of the extension, like the subclass operation producing **`@AuthoredClass`**, are irrelevant to the users of the **`Phone`** class and can be hidden. The meta-level program has been removed in this figure to illustrate that the programmer working at the base level can ignore it, focusing instead on his problem, which is now easier to solve because the extended substrate is better suited to support him.

This *separation of concerns* between the work on meta-level extensions and base level goals is highly compatible with the notion of software libraries. Library providers can write meta-level programs to implement the extensions they wish to provide. The base level programmer can take advantage of these extensions, without having to concern himself with the details of their implementation. The abstraction layer black boxes are designed to provide is therefore preserved, while the problem of their inflexibility is nevertheless being addressed.

Anything Else to Try?

Exercise: In the case of authored classes, the author information is best kept with the class metaobject because it is an attribute of the class. Sometimes, information should be kept with a class because all objects of the class should have access to it, and when it is updated, all objects should 'see' the new value. For that reason, some object systems provide a built-in *class variable* facility that can be used to implement features like class author. Design a class variable extension to Tiny Objects. Use the MOP to implement that in a new meta-class called **`@ClassVarsClass`**. Now reimplement the class author facility using class variables. Did you define a new meta-class or a new base class to do this?

In addition to class variables, class *properties* are often a useful facility to have. They differ from class variables in that they do not contain state. One occasion for their use would have been the example on page 121 and page 122 where animal species were modeled with properties such as furry and carnivorous. In this example, those properties were implemented as slots. That is not a good solution because being properties, they do not involve values requiring storage. Since furthermore they are properties at the species level, their repetition in each (animal) object is unnecessary and consumes space each time.

Write a program that implements animal properties at the species level. Begin by defining a **`@ClassPropsClass`** as a subclass of **`@Class`**. Provide for the necessary storage to hold properties, a way to initialize them, and the accessors for retrieving an animal's properties.

Automatic Subclass Selection

We have different implementations of set functionality. They are organized as multiple subclasses of a **`Set`** class. For a given set, the optimal implementation to choose depends on:

- How often elements will be accessed concurrently,
- How often elements will be added or removed and
- How big the set is likely to get.

We want programmers to obtain optimal impls of sets by specifying these properties, rather than the subclasses.

The problem is that object creation in TinyObjects is a much more rigid operation than what is requested here. In the default system, programmers have to specify exactly which class they want instantiated. What these programmers want is the ability to be a bit more declarative about their intention, and to have TinyObjects be more 'creative' in servicing programmer requests.

The programmers again start by imagining an ideal object system for their purposes. Here is a way this could look.

Examples:automatic subclass selection

What Is Being Asked?

Programmers using the set functionality should be able to write:

```
new Set (list ("high", "low", "large"));
```


```
--> #<ReplicatedHashSet 1>
```

When making a **`Set`** object, programmers would specify a usage profile they expected the new set to be subject to. Somehow the operation behind **`new`** would pick the correct subclass of **`Set`** to instantiate. The advantage to programmers would be that they would not have to learn about the details of **`Set`**'s subclasses, and would instead declaratively provide the implementation with the necessary information.

Implementors of the set library would have an additional advantage. They could add new, cleverly streamlined subclasses to **`Set`**. Client code would start classes right away when appropriate, without the base level programs having to be changed.

Analysis

- The subclass selection must happen after **new** is called, but before any object is created.
- Programmers may call both **new** and **@new**.

Conceptually, the programmer wants to extend the behavior of new and @new

Both **new** and **@new** need to look at the initialization arguments passed to them. From those arguments they must determine which subclass of **Set** should be instantiated.

This is a new class of problem. This time, programmers want to 'get in the middle' of a formerly atomic substrate operation to inject some additional behavior. What they want is commonly called a 'hook'. Hooks have a generally well-deserved bad name as non-modular, difficult-to-maintain programming style. Let us see whether our MOP designers can find a way out of this dilemma.

An Interface to This Functionality

```
class @SubcSelClass (@Class)
     (selectFunc);

class Set ()
     ()
   @SubcSelClass("@selectSetClass");

class ReplicatedHashSet (Set)...);

function @selectSetClass(properties) {
   case first(properties) == "high"
      ...}
```

Here is code that shows the way set functionality providers could go about expressing what they want.

Programmers would have the ability to specify that their **Set** class was to be a self-optimizing kind of a class. This would be done by specifying the **@SubcSelClass** option in **Set**'s class definition. This is the same syntactic form that was used for authored classes on page 146. The **@SubcSelClass** would be a meta-class subclassed from **@Class**. It would have a slot to hold a function pointer to a function that was responsible for computing an appropriate subclass of **Set** when given a list of usage profile properties. The **@selectSetClass** above is the stub of such a function.

This is at a syntactic level how programmers should be able to operate. Next, the MOP designers need to examine how this functionality would fit into the framework of the existing protocol and the model programmers have in their minds of how the meta- level works.

Fitting It into the Meta-level Model ?

What if **new** worked by calling **@new** and **@new** was a generic function?

```
method @new (@Class c, initargs) {
    allocate the object
    and call initializer
    }
```

If, whenever an application used the **new** primitive to create an object, the protocol promised that **@new** were called, then the protocol would ensure the existence of a "choke point" through which all object creation had to pass. If, furthermore, **@new** were a generic function, then programmers could add behavior to the object creation process by adding methods to **@new**.

Playing this out, it is clear what the **@new** method for the standard **@Class** would do. It would allocate space for the object about to be created, and it would call the initializers.

It is important to keep clear in one's mind that methods on this proposed **@new** would be for class metaobjects. They would not be base level methods. For example, programmers would not write an **@new** method on **Giraffe** but on **@findClass("Giraffe")**.

Notice that this is more than what the protocol on page 117 provided, where programmers were first allowed to call **@new** to trigger object creation. What goes beyond that specification is (i) that even the implementation is guaranteed to call **@new** when creating objects, and (ii) that programmers can write methods on this generic choke point function.

Selecting the Subclasses

Now the MOP programmer could do this:

```
method @new (@SubcSelClass c, initargs) {
   var subc = applyFn(c.selectFunc, initargs);
   return @new(subc, list());}
```

The programmer would add a method to extend what happens when objects of optimizing classes are created. The method shown here first retrieves the subclass-finding function from the class metaobject and runs it, passing as an argument the properties specified in the **new** statement. The code then calls **@new** recursively on the subclass computed by the subclass-finding function. That subclass would be a regular, non-optimizing class.

Question: Couldn't the programmers have written a **createSet** function that did the subclass selection and then called **@new**?

Answer: Yes. This is a good example for judgment calls as they sometimes come up. Programmers using object creation routines as suggested in the question would dilute the base interface. Their clients would have to remember that making set objects required a different construct than regular object creation. Any other, similar extension would require yet more special object creation routines whose names would have to be remembered.

By putting effort into a meta-level adjustment once, the providers of **Set**s allow cleaner programs in the context of the creation of **Set** objects, which is much more frequent than that initial definition of **Set**.

Documenting This Protocol

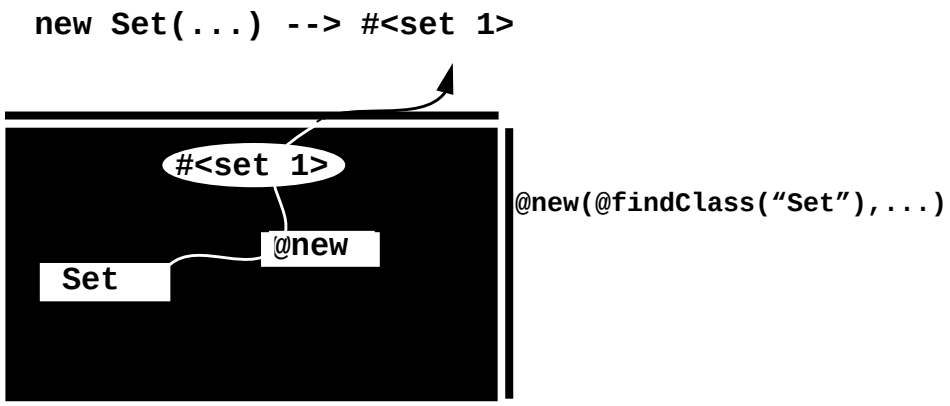
@new is the generic function that is called to create new objects of a class. It is called by **new** and any other code that creates objects. It accepts a class metaobject and initargs and returns an object. Programmers can add methods to **@new** that are specialized to subclasses of **@Class**.

Having satisfied themselves that this piece of protocol would indeed help programmers with their problem, the MOP designers can now go ahead and extend the protocol.

The above protocol extension is the solution to automated subclass selection.

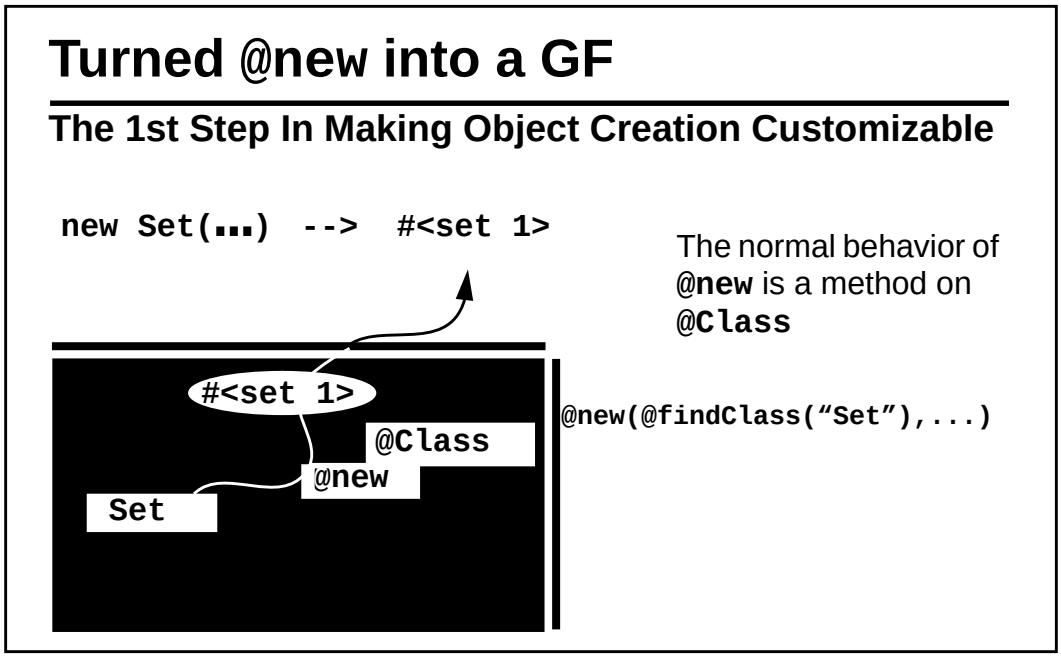
<$endrange>Examples:automatic subclass selection

Started with Reified Object Creation



Let us get our bearings again by looking at what this piece of protocol means in terms of black-box abstraction.

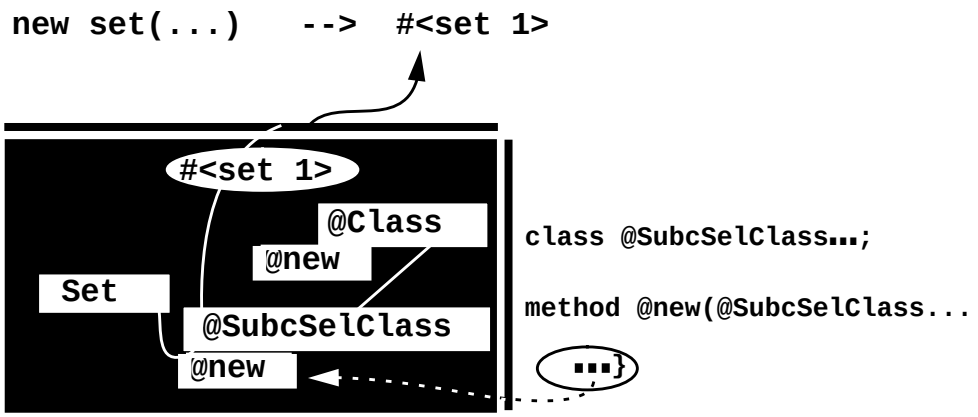
In order to allow the creation of objects from programs, the **@new** function was introduced in Chapter 4. It worked with classes, such as **Set** to create objects with the correct number of slots.



The next step towards allowing the implementation of automated subclass selection was to turn the **@new** function into a generic function. This in itself changed nothing. All programs on top of the box still ran; so did any meta programs written so far.

Localized Extension

New Behavior Applies Only to Optimizing Classes



Since the protocol already allowed subclassing of meta-classes, no other extensions to the MOP were required, other than explicitly allowing meta-interface users to add methods to **@new**.

Note that the ability to add an **@new** method now enables programmers using the meta-interface to inject behavior into the black box. Before, they were only able to inject state. Note as well that the use of object-oriented technology again provided the means for controlling scope: only TinyObjects classes explicitly declared to be implemented by **@SubcSel-Class** will have their object creation mechanism modified by the injected code. Such scope control is even more important for injected behavior than for injected state, because behavior extensions have the potential for more far-reaching consequences if their scope cannot be contained.

Once the correct protocol was in place, the implementation of optimizing classes and a whole family of similar extensions was straightforward.

Review of This Extension (1/2)

```
class @SubcSelClass (@Class)
     (selectFunc);

method @new (@SubcSelClass c, initargs) {
   var subc = applyFn(c.selectFunc, initargs);
   return @new(subc, list());}

function @selectSetClass(properties)
   case first(properties) == “high”
      ...}
```

Here is all the code programmers wrote on the meta-interface to implement the optimizing subclass selection extension.

Review of This Extension (2/2)

```
class Set ()
    (items)
  @SubcSelClass();

new Set (list ("high", "low", "large"));
```


```
--> #<ReplicatedHashSet 1>
```

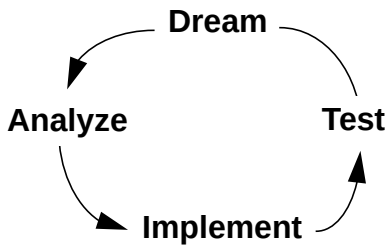
And here is what base-object system programmers can now do.

Exercise: This is not bad. But notice that whenever a new subclass of **`Set`** is created to provide an additional specialized implementation, the subclass-finding function **`@selectSetClass`** must be modified to check whether the given properties warrant this new class to be selected as the subclass of choice.

Rework this example so that every subclass of **`Set`** holds its own function for evaluating a usage profile. Each of these functions determines how appropriate its subclass would be as the implementation of choice under the given usage profile. The functions return a bid indicating how well their subclass could do. The **`@new`** method finds each of **`Set`**'s subclasses, obtains its evaluation function and calls it. Finally, **`@new`** instantiates the subclass who's evaluation function returned the winning bid.

Programmer's Work Cycle

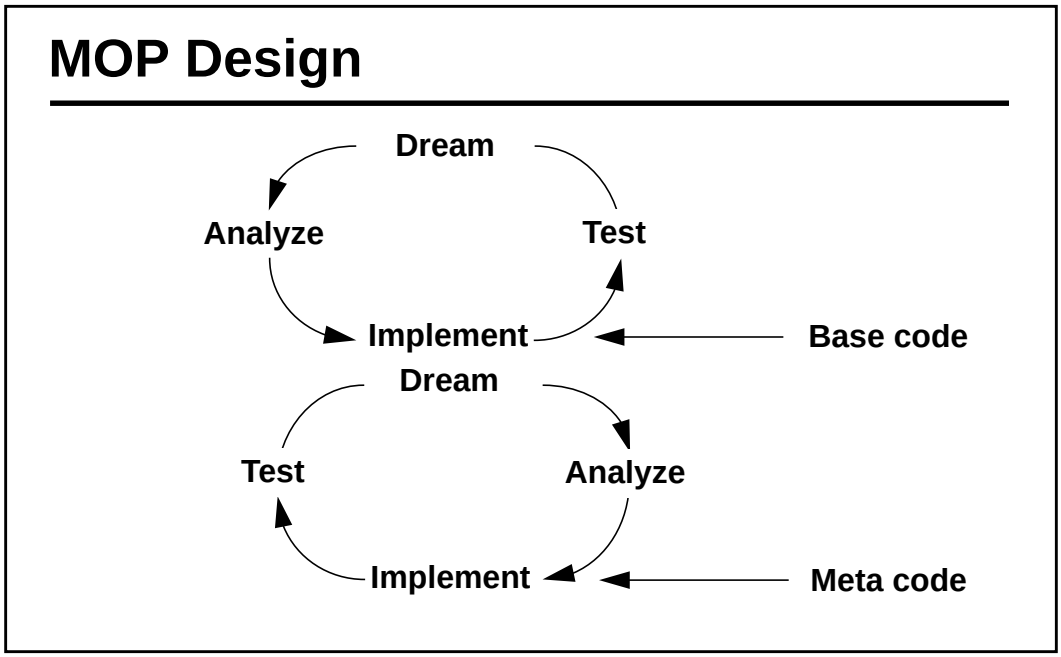
1. Dream: Write code in "ideal" base object system.
2. How is that object system different?
3. Implement new base object system with the MOP.
4. Test.



Now that we have gone through the base / meta design model a few times, we can visualize it. When base programmers run into a problem they cannot cleanly solve within the base object system, they work out the object system constructs that would best fill their needs, writing the code as if these constructs were already available. They then analyze what would be required of the substrate implementation to provide the constructs. To implement their extension, they turn to the MOP that comes with the substrate. Writing meta programs, they create the best object system for their purposes, test the result and continue their work.

Hopefully, the MOP is powerful enough to enable the necessary substrate adjustments. But if it is not, programmers need to turn to the MOP designer for help.

<$startrange>Design cycle

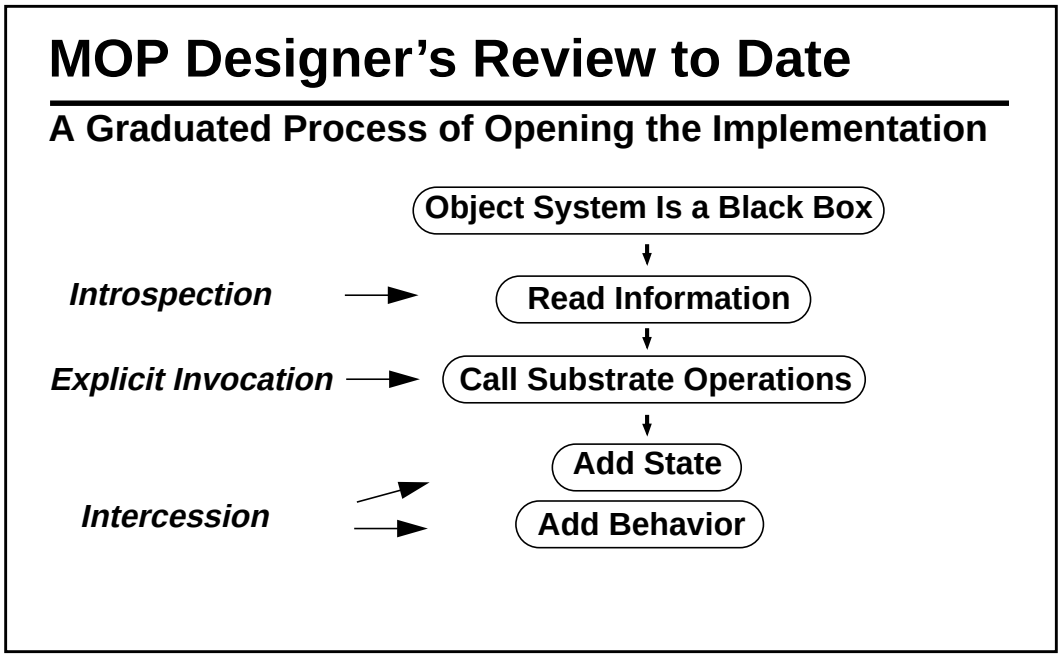


Here is the whole picture. Programmers select or construct substrate configurations that best suit their needs. If they cannot implement a new configuration they need by using the existing MOP, their problem becomes the subject of the MOP designers' concern. The application programmers' meta-interface programming needs thus drive MOP development.

MOP designers follow a cycle similar to that of programmers. When programmers come to them with shortcomings of the existing MOP, they begin by imagining the protocol that would most easily allow the programmers to write the meta programs they need. They then consider implementations of the substrate and analyze which structures or behaviors need to be reified and documented. The protocol 'implementation' consists of a thorough documentation, and the proof that efficient substrate implementations are possible. Testing of the protocol can involve the meta program needed by the programmer whose request initiated the MOP extension.

Eventually, the MOP will stabilize and programmers will be able to perform most of the substrate adjustments that are reasonable.

<$endrange>Design cycle

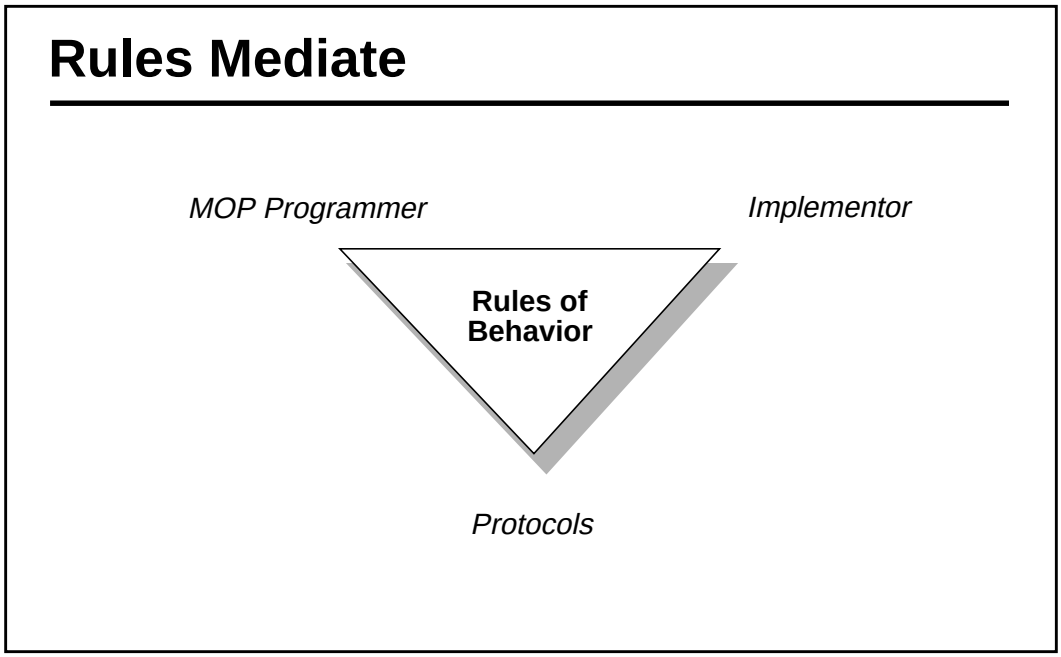


Here is another graphical view of our progression so far. Note that we have slowly moved along a spectrum of involvement with the object system. Initially, the substrate was a black box. Introspection added selected, and properly prepared transparent spots to the black box, corresponding to abstracted implementation state.

Explicit invocation allowed programmers to invoke substrate operations directly. Intercession enabled them to inject additional state and behavior into the box.

Adding author information to the metaobjects describing classes was an example of injected state. Causing **@new** to select a subclass before proceeding in the usual way was an example of behavior injected into the substrate to augment what the substrate was already doing.

<$startrange>Examples:changing inheritance model



What we have implied as well throughout is that when programmers work on the meta-interface, their behavior must be moderated by "rules of behavior," as is customary for software engineering in general.

Rules for Introspection

These rules ensure integrity of data structures and implementor freedom:

- No assumptions about ordering in sets (for example, **`@classDirectSubs`**)
- Do not modify lists or sets
- Successive reader results may or may not be identical
- Lists and sets may change if program changes (loading/reloading)

In particular, we have seen in an earlier copy of this figure that even simple introspective protocols come with such rules. They ensure that implementors retain the freedom they need to produce efficient code. Now we can begin to see that these rules are part of a pattern. These were the ones for introspection. Let us look at what kind of restraints the additional power of intercession requires.

Rules for Intercession (1/2)

Examples of what such rules must guard against:

- Slots that are private to the implementation being read/set by programmers
- Implementation methods being altered or prevented from running

Designing these rules is again just software engineering. The thought process that goes into them is the same as for making other systems robust. The capabilities available to programmers through intercession potentially harbor the dangers of private implementation information being accessed and system behavior being altered inappropriately.

Rules for Intercession (2/2)

- Specialization must not interfere with existing slots
- System methods must not be overridden

To guard against these, MOP designers need to document these rules for the use of intercessory capabilities: programmers must not define slots in subclasses of system classes that already exist, and they must ensure that system methods will run, unless the MOP explicitly specifies otherwise.

Note that many of these rules are checkable. There are other, more subtle ones that apply to substrate implementors. They are covered in [Kiczales and Lamping 92].

Changing the Inheritance Model

Dusty Decks

I have all this Flavors code and want to switch to TinyObjects. A crucial incompatibility between the respective object models lies in the inheritance rules. Can I fix TinyObjects to execute Flavors programs?

<$startrange>Class precedence list

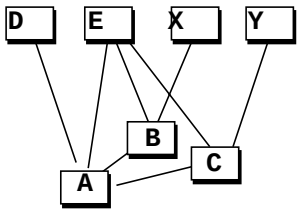
This example was actually an important part of the CLOS design. Flavors was a language that preceded CLOS, which is a language that preceded Dylan. Both Flavors and CLOS have multiple inheritance object models. Loops and Commonloops were in the family of multiple inheritance models as well. While some people love multiple inheritance and some people hate it, everyone agrees that we do not quite understand it. In particular, there is a question on how to resolve conflicts that can arise when inheriting from more than one superclass. These conflicts, for instance, require decisions on which slot should “win” when multiple slots with the same name but different properties are inherited, or on the order in which methods will be executed when several are applicable.

Object model designers cannot quite agree on whether and how to map multiple superclasses into linear CPLs as TinyObjects does. Flavors did it one way, CommonLoops another. Then several smart object model theoreticians sent megabytes of electronic mail back and forth with carefully reasoned arguments as to why there was a better way. CLOS ended up with yet a third solution, different from both Flavors and Commonloops. C++ expects programmers to manually resolve the conflicts, which is yet another approach.

Differences in Inheritance Models

It's all in the ordering:

```
class X () ();
class Y () ();
class D () ();
class E () ();
class B (E X) ();
class C (E Y) ();
class A (B C D E) ();
```



Different CPLs:

TinyObjects: **(A B C D E Y X ...)**

Flavors: **(A B C D E X Y ...)**

TinyObjects has multiple inheritance as well—different from Flavors. To exemplify this, consider the little example class graph above. **A** inherits from **B** and **C**, etc. TinyObjects and Flavors linearize this inheritance lattice slightly differently, as shown. The details do not matter. The point is, the linearization is different.

Impact of Inheritance Model

Conceptually, the CPL Encodes Inheritance Behavior

```
generic test (foo);

method test (X foo) do the X thing;
method test (Y foo) do the Y thing;
var obj = new A()

test(obj)
```

TinyObjects does the Y thing

Flavors does the X thing

Dispatch behavior is based on the CPL

Here is a program that specializes on the classes **X** and **Y**. If the **`test`** method is run with an **A** object in both Flavors and TinyObjects, TinyObjects runs the **Y** method, while Flavors runs the **X** method.

If the programmer needs to use a large legacy system written in Flavors and wants to add a few classes written in TinyObjects, he has a backward compatibility problem. What can the programmer do to make his programs work with the Flavors legacy code?

The first point to note is that the class precedence list is the key to inheritance. This was made clear on (page deleted) and through the subsequent examples there: when method dispatch is computed to execute a call to a generic function, the CPL of the class of the first argument is the determining factor for deciding which methods to run and in which order.

How could this insight help understand what this programmer needs?

Inheritance Behavior Control

Declaring Inheritance Behavior on a Per Class Basis

```
class SomeOldClass () () @FlavorsClass();

class AnotherOldClass (SomeOldClass)
   (...)
  @FlavorsClass();
```

Inheritance of these classes should work the Flavors way

Conceptually: a differently constructed CPL for some classes—by programmer choice

The programmer would like to say "most of my classes have TinyObjects inheritance, but some of them are Flavors classes, and they should have this other—Flavors—inheritance behavior."

Since inheritance hinges on CPLs, per class control over inheritance behavior translates to per class control over CPLs.

What the programmer is asking for here is an ability to introduce very deep deviations in object system behavior, yet to retain modularity within his programs. The facilities that will be shown through the following pages are actual CLOS features. Thanks to them, CLOS did not need a special option in the language to switch between Flavors and CLOS inheritance semantics. Instead, programmers are empowered to shape the object model in a controlled manner.

For now, however, the TinyObjects programmer does not yet have such lofty facilities. Here is the MOP designer going to work again.

<$startrange>Class:initialization

Where is the CPL?

Class A

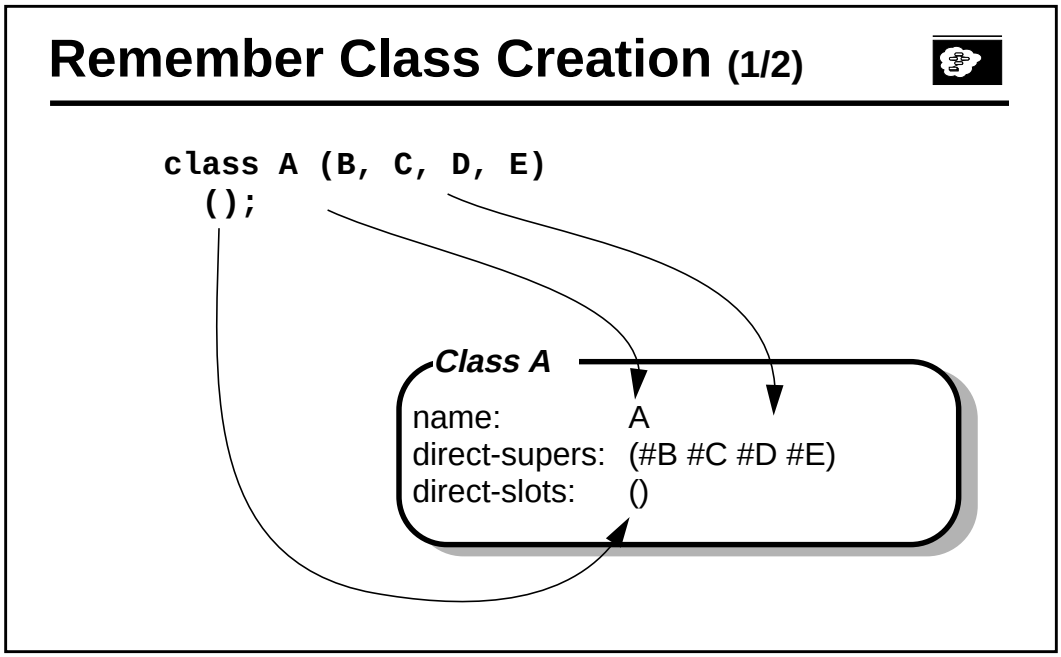
name:	A
direct-supers:	(…))
direct-slots:	(…)
cpl:	(#A #B #C #D #E #Y #X)
slots:	(…)
direct-subs:	(…)
direct-methods:	(…)
initializer:	#code

- The CPL is part of the information about a class
- What if the programmer could control the CPL?

As usual, the MOP designer needs to think first about the prototypical implementation. Conceptually, CPLs are part of the information all implementations of TinyObjects somehow need to maintain about each class. The introspective protocol of Chapter 3 guaranteed that **`@classCpl`** would return a given class' CPL. But it gave the programmer no control of CPL construction. If the MOP designer could find a way to extend the protocol so that the programmer could participate in the initial computation of each CPL, she would be close to a solution.

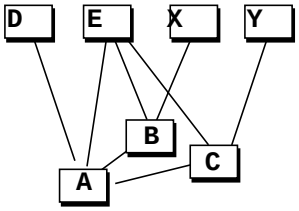
The question is, where does the CPL come from?

<$endrange>Class:initialization



Clearly, the CPL information must be constructed during class initialization. Remember that at the outset of this process, the only information available is what comes from the **`class`** definition directly. Somehow, the entire class definition is constructed from that information.

Remember Class Creation (2/2)



Class A

name:	A
direct-supers:	(#B #C #D #E)
direct-slots:	(state)
cpl:	(#A #B #C #D #E #Y #X)
slots:	()

Class precedence list (CPL) and complete set of slots are computed and stored

In particular, the class precedence list and the slots are computed from this initial information. Somewhere in the class initialization machinery, therefore, is a place where the CPL is computed. To find out where this happens, the MOP designer looks at class creation in more detail.

Looking for CPL Construction Site 

```
class A (B, C, D, E) ();

    new @Class(...)

      @new(@findClass("@Class"),...)

         run the initializer for @Class

                  ???
```

These are the inherent dynamics of class creation. It is already clear that there is a call to **new** to create a class metaobject (page 126). And it is known that **new** calls **@new** (page 162). Somewhere inside, the system will have to run the initializer of the class. And this is where the CPL will be constructed because that is where the class description metaobject is built.

Here is a look at what inherently happens when a class is initialized.

Details of Class Initialization

```
initializer @Class(c,
                   name,
                   directSupers
                   directSlots){

  c.name = name;
  c.directSupers = directSupers;
  c.directSlots = directSlots;

  updateDirectSupers(directSupers, c);

  c.cpl = @computeCpl(c);
  c.slots = @computeSlots(c);}
```

direct properties

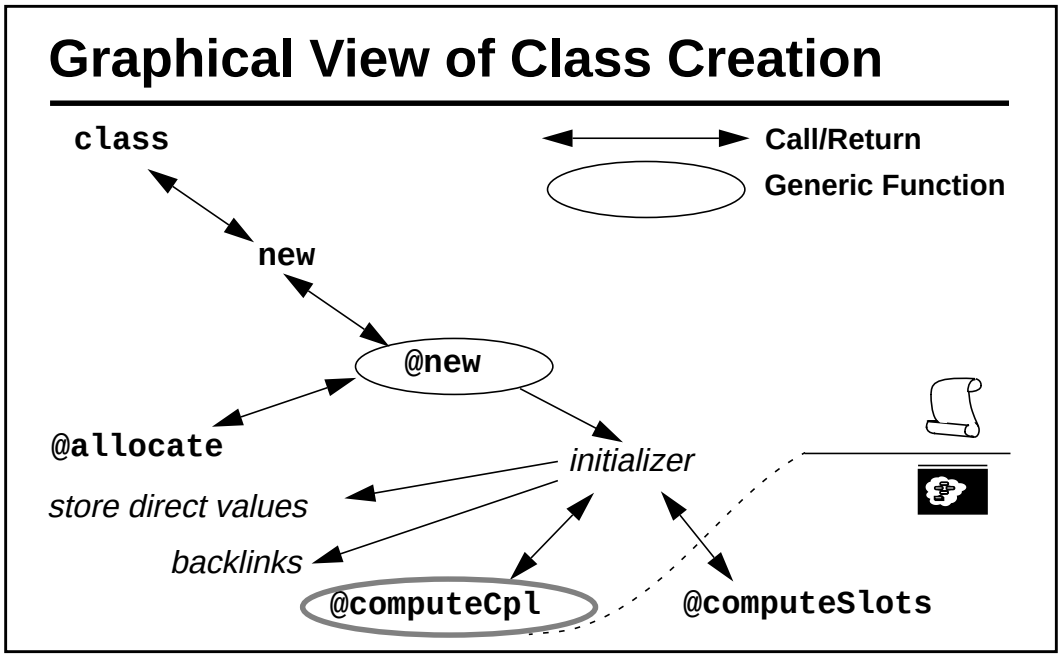
backlinks

derived properties

The three main pieces of work done during class initialization are (i) the storage of properties that are specified by the programmer in his class declaration, (ii) the maintenance of backlinks, and (iii) the computations of class properties that are derived from the ones that are directly specified.

The latter point is were the MOP designer strikes gold: one of the derived properties is the class' CPL. Finally, she has found the choke point a protocol extension might use to allow the programmer into the CPL construction process.

<$endrange>Examples:changing inheritance model



Graphically, the whole initialization process looks like this. Everything above the solid horizontal line on the right of the figure above has already been specified in the MOP developed so far. It is known that the **`class`** declaration invokes **`new`**, which calls **`@new`**. Now, the MOP designer will pull the line that divides the documented from the undocumented part of the class initialization process down along the dotted path.

This means that implementations will now have to compute the CPL by calling **`@computeCPL`** after they have stored the direct superclasses, so that **`@computeCpl`** has access to them.

Programmer Control over the CPL

If **@computeCpl** were a generic function, programmers could do this:

```
method @computeCpl (@FlavorsClass c){
 Sort supers the Flavors way}
```

If, furthermore, **@computeCpl** were a generic function, the programer could add methods that performed the CPL computation appropriate for his purposes. In particular, he could add a method that sorted superclasses the Flavors way.

This thought experiment shows that a MOP extension along these lines would solve the programmer's problem and, most likely, a whole class of other inheritance-related issues as well. Here is the documentation to make the new piece of protocol official.

Access to CPL Computation (1/2)

- **`@computeCpl`** is the generic function that is called by all TinyObjects implementations to compute class precedence lists. It takes a class metaobject and is called during class initialization, after the direct superclasses have been stored.
- The CPL returned can be retrieved with **`@classCpl.`**
- Implementations will use this CPL as the basis for all inheritance-related computations, such as method dispatch.

The only way any implementation is allowed to compute the CPL from now on is by calling **`@computeCpl`**. This means that every class can compute the CPL its own way. The protocol further specifies that the CPL will then be available through calls to **`@classCpl`**.

<$startrange>Class precedence list:participation in construction of

Access to CPL Computation (2/2)

The programmer can define methods on **`@computeCpl`** that override the system method, but the CPL returned by those methods must:

- Include each and every ancestor once and only once
- Start with the class and end with **`Object`**
- Respect direct precedence relations
- Be a function of the class graph only

The protocol also explicitly allows the programmer to add methods to **`@computeCpl`**. But it would be wrong to allow him to return just any list of superclasses as his CPL. This is because there are many decisions in the design of TinyObjects that depend on certain assumptions of how the inheritance will work. For example, systems should be able to rely on every CPL being arranged such that subclasses come before superclasses and that **`@computeCpl`** is a function of the class graph and the precedence relations only. This restriction is necessary to ensure that inheritance behavior is not arbitrary and can be predicted from those properties of programs that are relevant to inheritance.[1]

<$endrange>Class precedence list:participation in construction of

1. It will be shown later that this restriction has the additional purpose of ensuring that **`@computeCpl`** can be called at compile time to ensure that TinyObjects operations such as method dispatch can be optimized.

How It All Works

The implementors do:

```
generic @computeCpl (c);

method @computeCpl (@Class c) {
    Sort supers the TinyObjects way}
```

And ensure that **@computeCpl** is called.

The MOP programmer can then write:

```
method @computeCpl (@FlavorsClass c){
    Sort supers the Flavors way}
```

This new protocol is quite simple to implement and easy to use. The implementors simply need to make **@computeCpl** generic, providing a method specialized to **@Class** which performs the default TinyObjects CPL operations.

The lower half of the figure summarizes again how the programmer in this example could make use of this new piece of protocol. It is an example that initially appears too deep to tackle successfully, but proves to be quite reasonable and useful.

<$endrange>Class precedence list

Central Questions to Address

Standards New Capabilities Must Meet

- Usefulness
  - What is the new capability good for?
  - Does it leave programmers with a coherent system?
- Feasibility of implementation
  - Can implementation be made robust?
  - Can it be efficient?

While the programmer and MOP designer should challenge implementors who object to requests merely on the basis of tradition, the MOP designer in particular must provide good answers to valid implementor concerns: will the programmer build crazy concoctions and then complain? Will the implementation be too fragile? Will the implementation be slow?

In that spirit, it is time to return to the checklist above. The Flavors legacy system is one example of what control over CPL construction is good for. Another application might use **`@computeCpl`** to add required superclasses into the CPL. This would be in a situation where every class in some application framework was required to inherit from those superclasses, but the framework provider wanted to save the programmer from having to specify those superclasses whenever he used the framework.

But what about coherence? Does programmer control over CPL computation jeopardize system usability?

Coherence

Must ensure that the programmer cannot hurt himself too badly and that he cannot hurt others:

- CPL invariants to maintain reasonable inheritance rules.
- Programmer "confined" to the **`@computeCpl`** GF; cannot randomly change the CPL anytime.

In order to evaluate threats to coherence when CPL computations are made accessible, the MOP designer essentially said: class precedence lists must be sorted lists of ancestors, and they have to respect direct precedence relations. TinyObjects simply is the kind of object model for which this is true. The protocol provides the programmer with some leeway within these boundaries, but the basic nature of the model must be safeguarded.

Note that adherence to this requirement is checkable. The initialization process will call **`@computeCpl`**, and it can inspect its answer to ensure that the function respected these rules.

Since the CPL computation is a function of the class graph, the CPL cannot change over time. This is a further level of confinement that other programs and the object system implementations can rely on.

Coherence:intercession of; Intercession:coherence in

Robustness

Must ensure that system data structures will not change unpredictably or otherwise be damaged:

- The programmer cannot modify the CPL once it is computed
- The computed CPL must exhibit specific properties which are checkable after the call to **`@computeCpl`**

The reason robustness can be maintained in this example is, again, that the CPL is checkable right after it is computed. This ensures that implementations will not be tripped up later on. The rule about the CPL being required to remain static afterwards ensures that implementations can continue to make correctness assumptions about the CPLs in the system from then on.

Intercession:robustness in; Robustness:intercession of

Performance

Must ensure that implementation can employ caching and other optimization techniques, in spite of programmer involvement:

CPL computations are O.K. because:

- New CPL comes in at a well-defined point before any objects are created. System calls **`@computeCpl`** and installs the result.
- Programmer cannot otherwise modify a class's CPL

This will continue to be important later.

The performance question involves an examination of whether the default object system implementations and their extensions can be fast. This will come up in Chapter 7 and will therefore not be discussed in great detail here.

An important point, however, is that since the CPL must be a function of the class graph only and cannot change over time, implementations are free to cache the CPL and all computations dependent on it after its initial construction. The protocol has therefore carefully preserved exactly the freedom implementations need to optimize its post class definition run-time.

Just Good Software Engineering

These rules and protocol design esthetics are nothing fancy. They come from adopting our basic stance and then just doing elegant modular design.

Note that all these points about coherence, robustness and performance are generally applicable issues around good design. They are as important to consider in application programs or traditional systems programs as they are in the context of metaobject protocols.

The Stance

Be open-minded towards meeting programmer needs by giving him principled effective access to the object system implementation, with the thought that if this is done, he will be able to solve many of his problems himself.

The motivation behind the extra effort required for the design of MOPs is the stance seen earlier: if provided with principled substrate access that does not jeopardize coherence, robustness and performance, the programmer can solve many problems easily and cleanly, which he would otherwise have to "hack together."

Here's a Punch in the Nose

PROPOSAL: Document a **`@setCpl`** function which programs can call at any time with any list of classes.

BUT:

- What if the CPL is changed in the middle of method execution and **`callNextMethod`** breaks as a result?
- What if the CPL is changed to not include **`Object`**? Anyone relying on methods on **`Object`** being applicable to all objects will be in trouble.
- Performance!

Here is an example of what happens when a protocol is not carefully designed. This proposal is superficially similar to the protocol developed earlier. After all, it provides the programmer with control over the CPL which in turn drives inheritance behavior. But it causes ill-defined behavior in many cases, can break important object model properties and presents tremendous implementation problems.

This bad example illustrates that coherence, robustness and implementation feasibility require careful protocol design which provides levers, not crowbars.

<$startrange>Coherence:violation, example

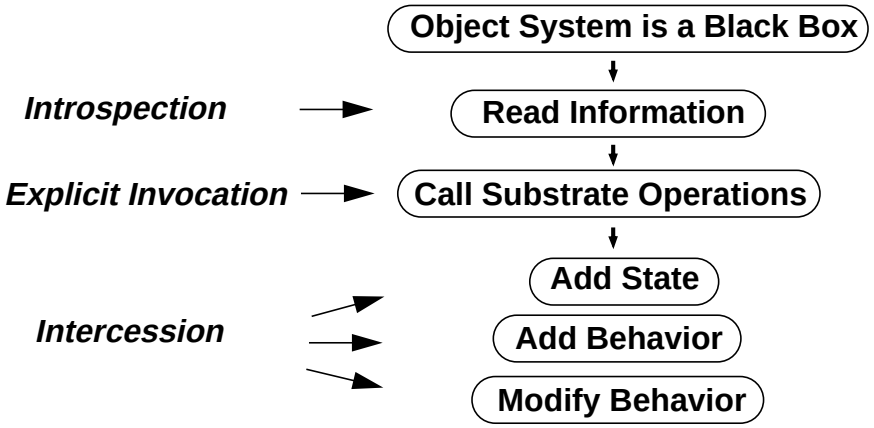
Here's a Knockout Punch

PROPOSAL: Allow the program to hold on to the CPL returned by their method on **`@computeCpl`**, and to randomly mutate it anytime.

This MOP design would be even worse: it would allow the programmer to modify the CPL at random times without even calling **`@setCpl`** to let the system know. Unpredictable programs would undoubtedly result sooner or later.

<$endrange>Coherence:violation, example

Review: The Terrain Covered So Far



At the starting point of this tutorial, TinyObjects was a black box substrate. It was useful, but often inadequate for a particular programmer. The terms *introspection*, *explicit invocation* and *intercession* were introduced over the previous chapters to describe three kinds of capabilities the programmer often misses when dealing with black box substrates.

Introspection allows the programmer to look into parts of the substrate, through an appropriate abstraction layer. Explicit invocation makes available selected functionality that was previously hidden. Intercession, finally, lets the programmer add to the substrate, and make modifications. All within boundaries defined by protocols.

Those capabilities are presented to the programmer through an object-oriented meta interface constructed from metaobjects. Each successive capability is achieved through a progressive enrichment of that interface.

Review: How and Why

Protocol Exposes	Capabilities	Concept
Metaobjects and readers	Specialized browsers, analysis	Introspection
Meta-level operators that implement obj system ops	Custom interfaces to base obj system	Explicit invocation
MO classes subclassable: to: add slots add methods	Add information to program rep. Add behavior to the object system	Intercession

Here is a more detailed view of the meta-level capabilities so far. When reading this table from top to bottom, the protocol becomes more and more rich, a deeper and deeper cut into the object system substrate, and the MOP programmer can achieve more and more powerful work with the object system. To enable introspection, as needed for the construction of specialized browsers or analysis tools, the metaobject protocol exposed metaobjects and readers for them. This was a reification of information maintained by all implementations.

The implementation of custom interfaces to the base object system, such as alternative ways of creating objects and classes, required the ability to effect corresponding actions in the substrate, without using standard syntax which is sometimes too restrictive. This requirement was met by reifying behavior, that is by making some operations on metaobjects available to the programmer.

Finally, intercession worked by allowing the programmer to subclass metaobject classes and to write methods on specific “choke point” generic functions.

The Design Process

It's Not All or Nothing

- The implementation is not completely opened up
- Access may be provided to varying depths, providing correspondingly increasing power
- Depth of access is matched to programmer needs
- Access is organized through well-documented protocols, not random user hacking of internals

It is important to keep in mind throughout that MOPs do not expose the entire implementation, but very specific parts of it. The substrate designer only needs to go as far as her clients require. If only introspection is needed, only an introspective protocol needs to be provided. The MOP designer could stop right there.

Access to the substrate implementations can be graduated, matching programmer needs. This is very different from letting the programmer randomly change the implementation, as happens when large numbers of more or less random "hooks" directly into the implementation are provided.

<$startrange>Design cycle

Anything Else to Try?

Well? Can you think of times when you wished in retrospect that you had been more assertive with your substrate provider and would now like to try your hand at introducing a protocol that afforded tailorability you needed?

<$endrange>Design cycle

Summary

- ☑ **Introspection**—examination of selected system internals.
- ☑ **Explicit Invocation**—enabling programs to explicitly invoke substrate operations, bypassing syntax in some cases.

Intercession—metalevel specialization that adds to or modifies existing features of the base object system:

- ☑ Adding State
- ☑ Adding Behavior
- ☑ Modifying Behavior
- ☐ Adjusting Performance

Chapter 6 But I Could Make It Run Better for My Application…

Intercession in Slot Access

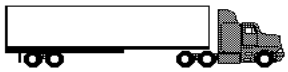
The objects in my program have images associated with them. My problem is that the images take up a lot of room in virtual memory. I would rather not read them in until I know I actually need them.

So far, all examples have used metaobject protocols to add features or adjust behavior. There are other reasons why the programmer might want to use MOPs. Often particular applications have characteristics that would allow significant optimizations in the substrate they build on, if only there was a way to adjust the substrate to take advantage of those peculiarities while serving that one application.

Here is an example involving savings in disk access time and memory space. This programmer would like “lazy slot caching” for his objects. He begins by extending the base object system syntax on paper to work out what he wants.

The Programmer Would Like to Write

```
class Truck ()
    (weight)
  @LazyInitClass("front", "side");

trl = new Truck(60000);

trl.side -->
```



This slot access should load the image, and save it away in the slot.

The **Truck** class defined here has three slots—**front**, **side**, and **weight**. The declaration following the list of slots would identify the class as one that is able to defer slot initialization, that is, as a lazy class. Recall that this class definition syntax came up before on page 148. The **@LazyInitClass** field after the slot list is a specification of which class meta-class **Truck** is to be made from. The list following it are initialization arguments used in the class initialization process. In this case, that is a list of lazy slots.

In analyzing what he needs to do to obtain this facility, the programmer comes to the following conclusion...

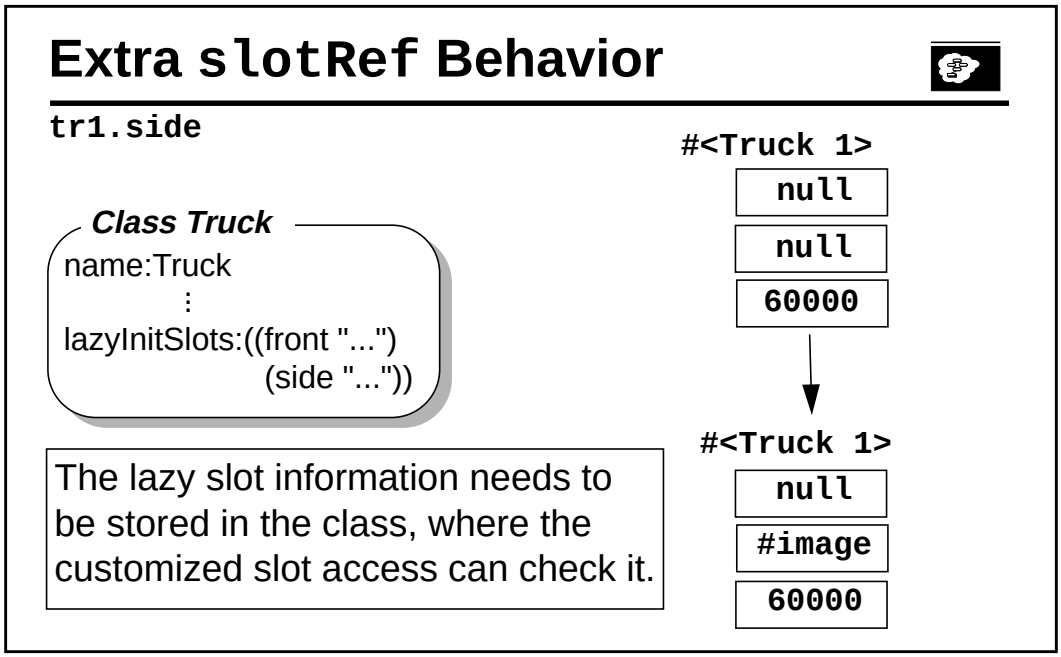
How to Make This Work?

The programmer needs to do two things:

1. Classes must have information on which slots have lazy initializations—this is simple intercession.
2. The behavior of slot access must be augmented—more MOP will be needed to do this.

The first of these is straightforward. He needs to keep a piece of information with a class metaobject. Intercession allows him to do this. So the current MOP is powerful enough to meet the first requirement. But for the second, the programer needs to modify what happens when slots are accessed. The MOP developed so far is insufficient for this.

To clarify these two goals further, he thinks a bit more in terms of what—qualitatively—he needs to accomplish in the object system machinery.



On the left of this figure is a visualization of the information maintained about the **`Truck`** class. In addition to the class name and the other class-related information, there is the new information about which slots are lazy, and where their contents are stored. On the right are two snapshots of one **`Truck`** object: one taken before execution of the **`tr1.side`** operation, one after. In the before picture, the second slot (**`side`**) is not filled in. But after the operation, in the lower snapshot, the image has been loaded. This loading operation is what the second problem in the previous figure is about.

Extra Information in the Class

```
class @LazyInitClass (@Class)
    (lazyInitSlots);

initializer @LazyInitClass (c,
                            className,
                            directSupers,
                            directSlots,
                            lazyInitSlots) {
  c.lazyInitSlots = lazyInitSlots;}
```

Here is the easy part. This code adds the lazy slot information to class metaobjects. It is by now idiomatic for the programmer versed in the meta interface.

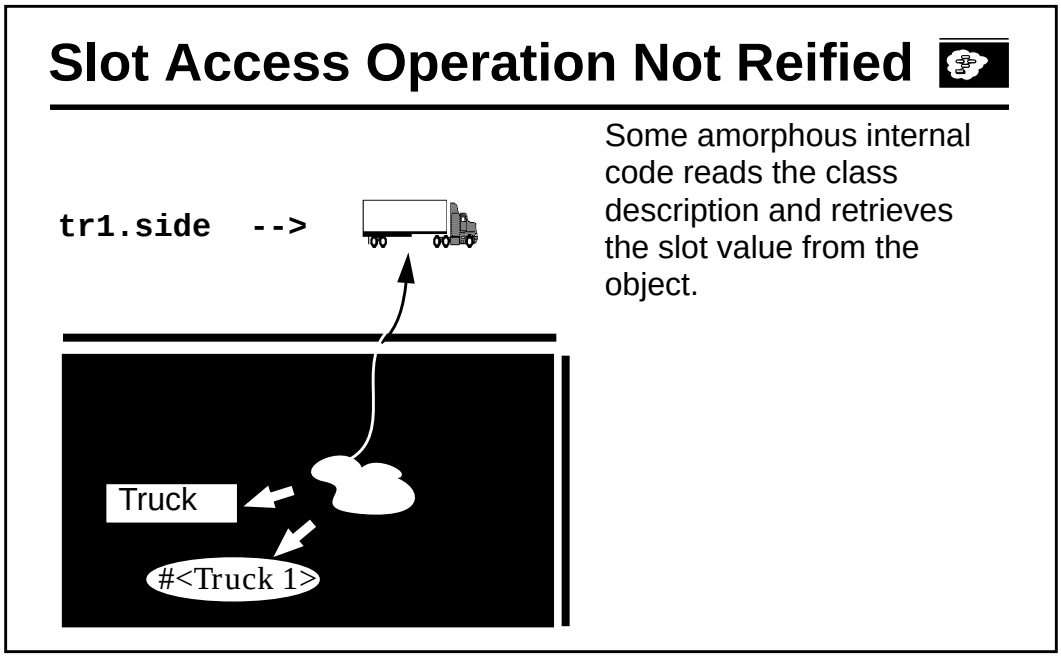
Changing Slot Access

The programmer wants to augment slot access behavior, but the mechanism behind expressions like **`obj.slotName`** is not reified.

More protocol is needed, in particular, some way of controlling slot access.

The obvious place to intercede with slot access is the function which is responsible for handling operations like **`tr1.side`**. The problem is that this operation is not reified. So the programmer cannot cleanly provide any facilities that add or change functionality around slot access.

The MOP designer did not anticipate this class of programmer needs, and since it is likely that there are other examples like the lazy slots which require principled ways for working with slot access, it makes sense for her to extend the metaobject protocol.

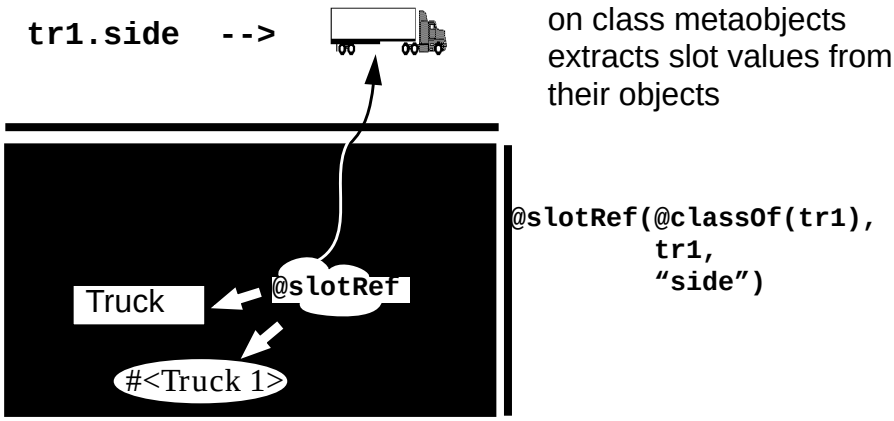


To do this, the MOP designer again looks at the basic events around slot access that are true for all implementations alike.

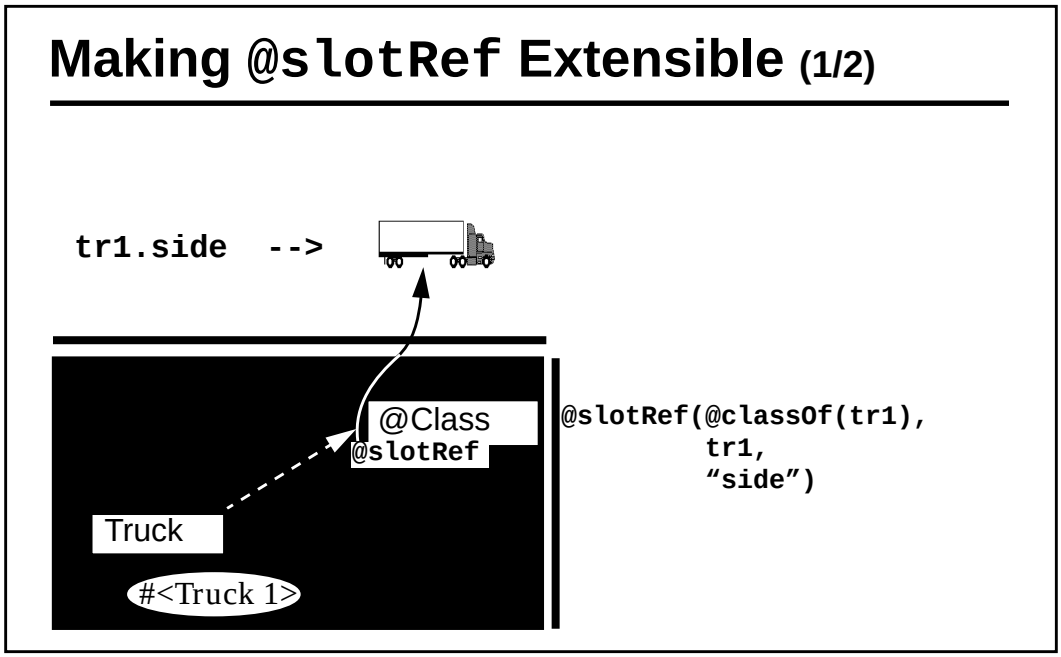
Somehow, a piece of code or collection of functions in every implementation uses information from the class metaobject to extract the desired information from the object. This code is indicated by the white cloud in the black box.

Recall that the MOP designer was confronted with a similar situation on page 204, when the programmer requested ways to create objects from classes known only at runtime. The same technique for MOP development that was used then will work here...

Reifying Slot Access

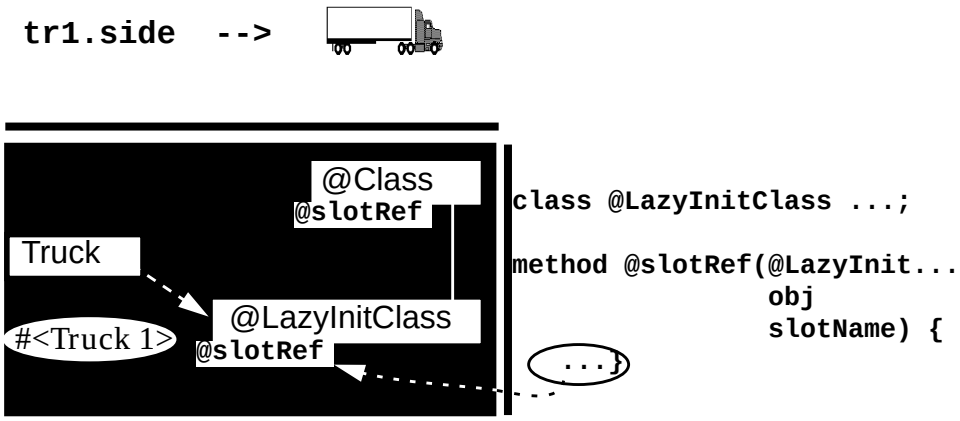


The MOP designer gives a name to the slot access operation, reifying it, so that it can be made accessible from the outside. One possibility would then be for the resulting entity to be made a function for the programmer to call. As was true with **@new**, this would not be acceptable, because it would not afford control over the scope of programmers' extensions: if the programmer changed **@slotRef**, then all TinyObjects programs would be forced to use that new version.



And so the final solution is to make **@slotRef** generic. The code which provides slot access for programs using the default TinyObjects is simply the **@slotRef** method on **@Class**...

Making @slotRef Extensible (2/2)



...to which the programmer adds his own methods, specialized to sub-classes of **@Class**.

This will work for the programmer. But before plunging ahead, some more checking is needed.

Implementing This New Protocol

```
obj.slotName
   compiles to: @slotRef(@classOf(obj),
                         obj,
                         "slotName")


method @slotRef (@Class c, obj, slotName) {
    whatever the original definition of slot access did}
```

More about this shortly

Before really documenting the new proposed piece of protocol, the MOP designer needs to examine whether it can be implemented in such a way that old code will not break, and whether code taking advantage of the new protocol is *interoperable* with code using the default protocol: can they exist in the same address space, and can they interact with each other?

Here is the result of this analysis. At least conceptually, under the new protocol implementations could compile any **`obj.slotName`** operation into a call to **`@slotRef`**. As mentioned, an **`@slotRef`** method on **`@Class`** could contain the current (default) slot access code.

Furthermore, before adopting this piece of protocol, the MOP designer needs to perform a deeper analysis yet to ensure that efficient implementations are possible and that the protocol does not prevent use of implementation optimizations. Chapter 7 will introduce this type of analysis.

Slot Access Intercession

- **`@slotRef`** is the generic function that is called to implement slot access. It receives as arguments the class of object, the object and the slot name.
- Methods specialized to subclasses of **`@Class`** may be added to **`@slotRef`**. These may override the system method.

Having satisfied herself that the new piece of protocol will (i) solve an important class of problems for this and other programmers, and that (ii) implementors will not be unduly constrained, the MOP designer can finally publish the extension to the MOP.

Question: Couldn't there instead just be a generic function that takes an object and a slot name? It wouldn't be meta any more, but it would have the same functionality.

Answer: Yes, that would be similar, but not the same. There are two reasons to do it the way presented here. The first is that conceptually, slot access is more a behavior belonging to classes than to objects. It can therefore be argued that methods specialized to class meta-classes are more appropriate for modifying slot access behavior. A second reason for using the meta-level is that this enables individual classes to have special slot access behavior which is not passed down to its subclasses. The solution proposed in the question would need to work through the subclassing mechanism, making the scope of the slot access behavior modification less easily limited: once the behavior was inherited, it would be propagated to all subclasses.

Extending @slotRef (1/2)

```
method @slotRef (@LazyInitClass c,
                 obj,
                 slotName) {
   var value = callNextMethod();
   if defined(value)
      return value;
   else
      value = lazySlotInitValue(obj, slotName)
   return value;}}}
```

With this piece of protocol, the programmer of this example can solve the second of his problems: modifying slot access behavior to support lazy slots. This takes the form of an **@slotRef** method on **@LazyInitClass** which was already introduced on page 208.

The **callNextMethod** returns a value if either the slot is not lazy, or if it is lazy but has been accessed before[1]. If the slot is lazy, **@lazy-SlotInitValue** finds the proper initial value and caches it into the slot, so that it will be found on any subsequent **@slotRef** calls for this object and slot.

1. Note that for simplicity uninitialized, non-lazy slots are ignored here.

Extending `@slotRef` (2/2)

```
generic lazySlotInitValue(obj, slotName);
method lazySlotInitValue (Object x, slotName) {
   error("No such lazy slot.");};
method lazySlotInitValue (Truck obj, slotName) {
   var value;
   if slotName == "side"
      value = fetchFromFile("/truck.side");
   if slotName == "front"
      value = fetchFromFile("/truck.front");
   else value = callNextMethod();
   @slotSet(@classOf(obj), slotName, value);
   return value;}
```

The **`lazySlotInitValue`** is a generic function to which methods may be added for base level classes. It is responsible for initializing values of lazy slots when they are first accessed. Methods might initialize all slots of all objects of a class the same way, they may perform different initializations for each object, or they might treat different slots differently, but treat all objects of a class the same.

The code above does the latter. If the slot passed into the method on **`Truck`** is a lazy truck slot, the correct value for the respective slot is retrieved from a file. Otherwise the method defers to methods that might be defined on the superclasses of **`Truck`**. The value retrieved is placed in the proper slot to cache it.

The method on **`Object`**, the root of the base level inheritance graph, catches cases of undefined lazy slots.

This concludes the lazy slots example.

I Need More Than Slots…

In my banking application I keep information in slots. But I want the ability to store information about the information. I tried to keep extra slots for additional material, but that was hard to maintain and confusing. I would like to have something like sub-slots or slot attributes, that are inherited by subclasses.

This new request tests the generality of the new **`@slotRef`** protocol.

This programmer has a banking application, and his objects hold information such as account balances, or credit line information. But bankers always like to look over each other's shoulders, and so they want also to store information about the slot values themselves. They may, for example, want to see the credit level and then want to find out who most recently adjusted that credit level. The programmer originally tried this by adding additional slots with this information. But what he really wants is something like subslots.

This Programmer Wants to Write

```
class Rating ()
    (name level)
 @AttributesClass(list(list("level","dateSet"),
                       list("level","whoSet")));

cr = new Rating()

cr.level = 3000
attributeSet(cr, "level", "dateSet", "9/27")

cr.level --> 3000
attributeRef(cr, "level", "dateSet")--> "9/27"
```

Here is a syntax which would fit the syntax pattern that has evolved throughout the previous examples: the use of a class option to introduce a new notion. In this example usage, two attributes, **dateSet** and **whoSet** are provided for the slot **level**. They could be used as shown. The programmer can use **attributeSet** and **attributeRef** to set and retrieve attribute values for slots. Once the **attributeSet** mechanism was available, one could imagine using the slot access intercession techniques of the previous example to set the date and operator attributes automatically.

What Is to Be Done?

This is again a two-part problem:

1. Class must have information on which slot has which attributes (analogous to **`@classSlots`**)
2. Must keep attribute values in each object (analogous to slot storage in objects)

Will require knowing where in the object to look for each attribute—more protocol will be needed.

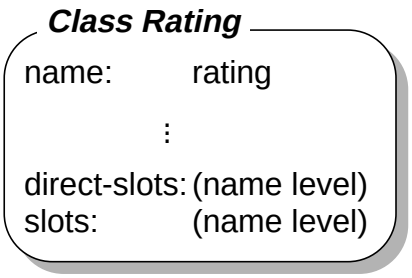
To implement this object system extension, the programmer needs to accomplish two tasks: he must maintain information with each class about which slots have which attributes, and he must find a way to store the actual attribute values in each object.

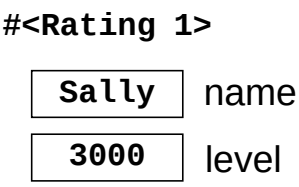
The main job for the programmer now is to find out whether the current MOP is powerful enough to let him accomplish these subgoals.

The first is simple. It is almost like class authors or lazy slots and can be handled by intercessory techniques at the meta interface.

But the MOP developed thus far says nothing about how to add information to objects for which space has not already been provided by slot definitions in the class. So the MOP designer needs to work to cover this terrain. Here is a visualization of the situation.

Class Is a Map to Objects (1/2)

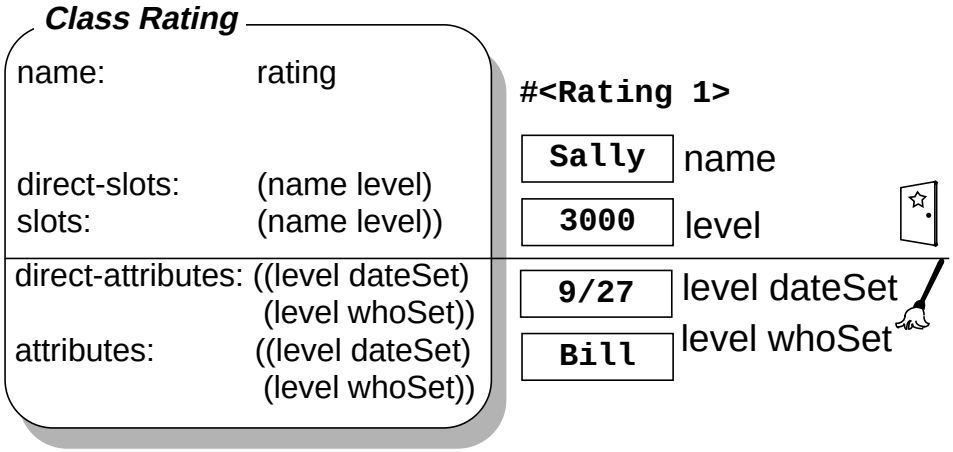




One way to think about the role of classes which helps in this situation, is to think of them as maps to the layout of the objects made from them. Objects are blobs of storage. In order to get/set slot values in them, there needs to be a mapping from the slots names to their locations in objects. Classes perform this mapping function. For example, this figure shows a visualization of the metaobject which represents the **`Rating`** class. On the right is the representation of an example object created from this class. The metaobject provides object information—such as the names and number of slots—which is needed to update or retrieve slots in the object.

Now, how could attributes fit into this?

Class Is a Map to Objects (2/2)



One way could be to just generalize from how slots are handled and to give the programmer appropriate abstractions into the mapping functionality.

This would imply an expansion of the mapping facilities of classes to cover access to attributes as well as slots. This figure illustrates what needs to happen. In analogy to slots, space must be provided in class metaobjects to hold information about attributes defined on the class directly, and about the ones inherited from superclasses.[1]

If a new piece of MOP were to be worked out carefully enough, the programmer could cleanly use the meta interface to extend the default object layout strategy shown above the line in the figure.

In the following, the problem of the programmer adding information about attributes to classes will be solved. Then the MOP designer will find a way to provide an abstraction for adding space to base-level objects.

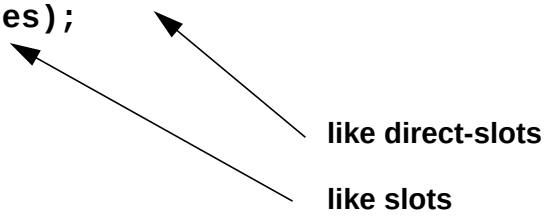
---

1. The map could be made more space efficient by not repeating the slot name each time. Doing that introduces some uninteresting complexity, so it will be done this way here.

Keeping Information about Attributes

Intercession Again

```
class @AttributesClass (Class)
    (directAttributes,
     attributes);
```



Adding space for the information in the class is simple. It works just like authored classes.

But there is one issue which has not been addressed in previous examples: the programmer needs to decide on and implement an appropriate inheritance behavior for attributes. Examples: what should happen with attributes defined in superclasses? Should they be inherited or ignored? What to do with the same attribute handed down from several superclasses?

The programmer can again take the corresponding handling of slots as a template for this.

Remember Class Initialization

```
initializer @Class(c,
                   name,
                   directSupers,
                   directSlots) {

   c.name = name;
   c.directSupers = directSupers;          | direct
   c.directSlots = directSlots;            | properties

   updateDirectSupers(directSupers, c);    | backlinks

   c.cpl = @computeCpl(c);                 | derived
   c.slots = @computeSlots(c);}            | properties
```

Remember how slot information is inherently prepared during class initialization. First, the slots defined directly on the new class are stored in the metaobject describing the class. Then **`@computeSlots`** derives information about slots that are being inherited from superclasses. This information is stored in the class metaobject as well.

Something like this is clearly needed for attributes.

Imitation…

…Is the Sincerest Form of Flattery

```
initializer @AttributesClass(c,
                             className,
                             directSupers,
                             directSlots,
                             directAttributes) {
  c.directAttributes = directAttributes;
  c.attributes = @computeAttribs(c);}
```

The programmer can use the same pattern. In the initializer for his **`@AttributesClass`** he can install the attributes which are directly defined for the new class in the class metaobject and then call a new generic function, **`@computeAttribs`** to compute the inherited ones. The code above provides the initialization for the direct attributes and attribute entries shown below the implementation / meta interface line on page 222.

Inheritance of Attributes

```
method @computeAttribs(@AttributesClass c) {
   var result = list();
      foreach (ancestor, @classCpl(c))
         union(result,
            @classDirectAttributes(ancestor));
   return result;}
```

Must be careful!

This programmer will settle on very simple inheritance behavior for attributes: All attributes in superclasses are inherited. If several with identical names are handed down, one attribute of that name is retained. With this decision, the actual code for the attributes computation is simple. The method above runs through the CPL, asking each ancestor class about the attributes defined on it. The union of the answers is the set of inherited attributes.

But simple as it seems, this code conceals a problem similar to, but worse than one that came up earlier in the context of authored classes: what if not all the new class's superclasses are **`@AttributesClass`** kinds of classes? Then the class metaobjects describing those classes would not know how to respond to the **`@classDirectAttribtes`** message! This code would be in trouble. So the programmer will have to work out what to do in such cases. One reasonable solution semantically is to say that all such superclasses are simply classes which have no attributes defined on them. How can these semantics be implemented?

Non-Attributes Class Inheritance

CPL will contain classes that are not objects of **@AttributesClass**. They need to be handled gracefully.

```
generic @classDirectAttributes (c);

method @classDirectAttributes (@Class c) {
   return list();}

method @classDirectAttributes(@AttributesClass c){
   return c.directAttributes;}
```

As was done in the case of authored classes, the way to do this is to make **@classDirectAttributes** be a generic function, rather than just a regular function. This allows the programmer to ensure that even the default **@Class** knows about **@classDirectAttributes**. He does this by providing a method on **@classDirectAttributes** for **@Class** which returns an empty list. When inheritance is used to extend any object-oriented system, the inheritance hierarchy sometimes needs to be "stitched together" like this.

Note that none of the rules have been violated because, even though the programmer is specializing on a default (built-in) meta-level class, he is just *adding* a generic function, not overriding one.

What Next?

The two-part problem:

- ☑ Class must have information on which slot has which attributes (analogous to **`@classSlots`**)
- ☐ Must keep attribute values in each object (analogous to slot storage in objects)

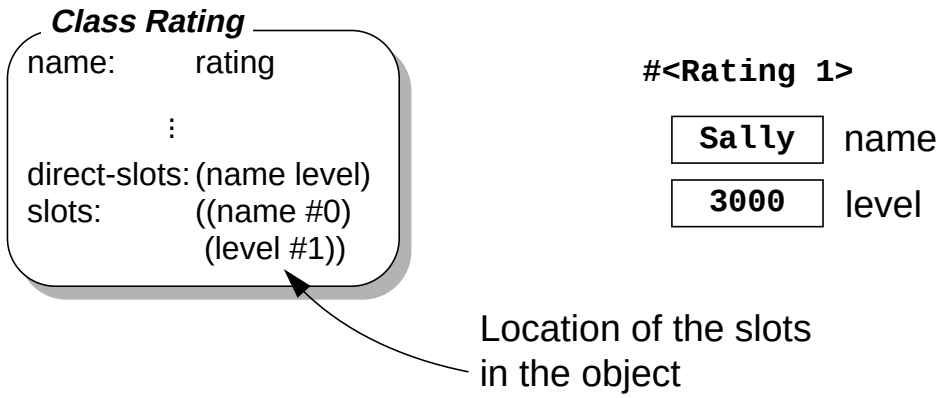
Now more detail will be examined to see how storage in objects is managed and accessed.

The first part of the agenda has now been covered. No MOP extension was necessary for it. The programmer was able to handle this part on his own.

But he has not yet obtained any storage in objects to store attribute values. As noted before, this is not possible with the current MOP, and the MOP designer needs to develop a new piece of protocol.

The Maps in More Detail

In reality, classes have a detailed map of where in their objects each slot can be found.



The visualizations shown so far of the information stored in class metaobjects were actually simplified. The information is in fact more detailed than was disclosed. In particular, each class metaobject needs to hold not just the names of slots but also their position within objects of that class. Otherwise the slot access mechanism would be unable to locate any particular slot within an object. So, in reality, the mapping from slots names to storage in objects is conceptually done more like this: in their list of slots, implementations probably keep detailed layout information for each slot. This would be true even more in typed models where values can be less than one word long, making precise locatability of slot values in memory more tricky. When slots are computed during class initialization, not only the full set of inherited and direct slots is therefore obtained, but also their locations in future objects of the class.

The notation used here uses **`#0`** to indicate a slot in the zero'th position within each object of the class; **`#1`** means the first position, and so on.

Slot Maps—Official Documentation

- **`@computeSlots`** produces slot-name and location pairs
- **`@classSlots`** returns just the names
- **`@classSlotLocations`** returns names and locations
- Methods on **`@slotRef`** and **`@slotSet`** use very low level object access by location

Here is the draft of a protocol that will eventually give the programmer the ability to solve his problem. In this new protocol, the designer changes **`@computeSlots`** to return pairs of slot names and their locations within objects. Notice that, in contrast to the preliminary visualization of page 229 which showed conceptually what happens in implementations and therefore used integers to exemplify slot location, the MOP designer is more abstract in the published protocol, talking instead simply of "locations" that are returned by **`@computeSlots`**. These locations can take any form decided on by the respective implementation.

The protocol above retains the specification that **`@classSlots`** simply returns a list of names, rather than names and locations, because earlier examples showed that this is often the most convenient behavior. The new **`@classSlotLocations`** can be used to get a hold of slot locations.

`@slotRef` and **`@slotSet`** are built on top of more primitive functions which take a location specification and access the respective slot in a given object.

Each of these will be examined now.

Slot and Location Accessors

```
function @classSlots (c) {
  var result = list();
  foreach (slotDescription, c.slots)
    extend(result, first(slotDescription));
  return result;}

function @classSlotLocations (c) {
  return c.slots;}
```

Here is the implementation of **@classSlots** and **@classSlotLocations**, the first returning just slot names, the second the slot/location pairs.

Reserving Locations for Slots

```
method @computeSlots (@Class c) {
   var result = list();
   foreach (slotName, @allSlotNames(c)) {
      var loc = @allocateLocation(c);
      extend(result, list(slotName, loc));}
   return result;}

function @allSlotNames (c) {
   var result = list();
   foreach (ancestor, @classCpl(c))
      union(result,
         @classDirectSlots(ancestor));
   return result;}
```

How does the new **@computeSlots** work? It will go through each of the slots (**foreach**...), directly defined on the class, or inherited, and use the primitive **@allocateLocation** to obtain a position in future objects for each of these slots. At runtime, when objects are created, they will have enough memory allocated for them to hold all slots.

@allocateLocation returns some quantity (location) that can be passed to access functions to get and set slot values at runtime. Again, these are just cookies, nobody outside of the object system implementation can use them for anything other than for passing them to other functions that expect them.

New Slot Access Method

```
method @slotRef (@Class c, obj, slotName) {
   var loc = lookup(slotName, c.slots);
   if (defined(loc))
      return @objectRef(obj, loc);
   else
      error("Can't find slot.");}
```

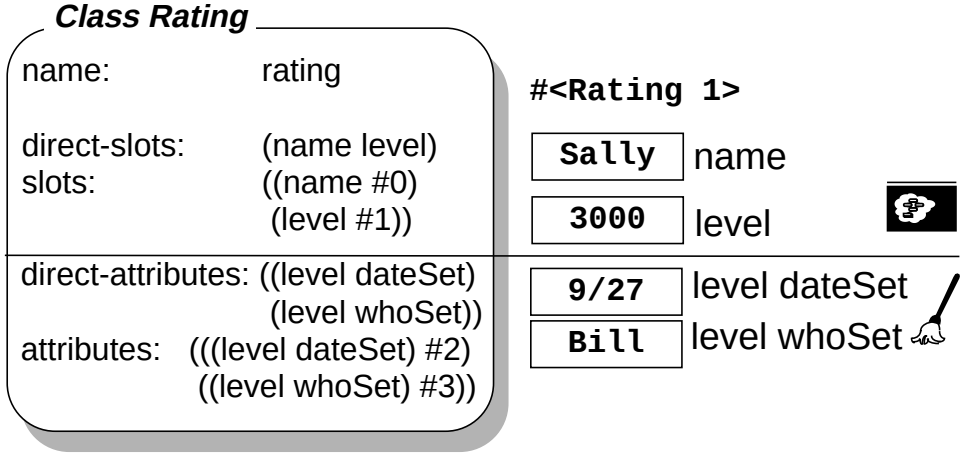
In particular, **@slotRef** uses the fast, low-level function **@objectRef** which takes an object and a location previously produced by **@allocate-Location** and returns the value stored there.

Protocol for Object Locations

- **`@allocateLocation`** can be called during class initialization only, to request that a location be allocated in each future object of the class. Each time it is called for a class, it returns a different location.
- **`@objectRef`** and **`@objectSet`** take objects and locations and reference/set that location in the object. The object must be an object of the exact class for which **`@allocateLocation`** returned the location.

Object maps, locations and the increased exposure of detail in slot access represent a layer of protocol below the **`@slotRef/@slotSet`** protocol introduced so far. Since the programmer will need to rely on these notions for his attribute solution, the MOP designer needs to write them into the official protocol. They also form the basis for alternative slot access behavior the meta programmer might want to implement.

Maps for Attributes



And with this piece of protocol in place, it becomes clear how the programmer will obtain space for his attributes in objects: he simply mimics the basic slot access mechanism just shown. The programmer extension code, in addition to computing the names of all attributes during class initialization, also requests space for them in the future objects. Reading and writing of the attributes will call the low-level accessors.

Obtaining Locations for the Attributes

```
method @computeAttribs(@AttributesClass c) {
   var result = list();
   foreach (attribute, @computeAttribs1(c))
     extend(result, list(attribute,
                         @allocateLocation(c)));
   return result;}

method @computeAttribs1(@AttributesClass c) {
   var result = list();
   foreach (ancestor, @classCpl(c))
     union(result,
        @classDirectAttributes(ancestor));
   return result;}
```

Here is that extension code. **`@computeAttribs1`** collects the names of all attributes in the class being defined when the method is called. The calls to **`@allocateLocation`** within the **`@computeAttribs`** method then ensure that extra space will be allocated in future objects to hold the attributes.

Accessing Attributes

```
function attributeRef (obj, slot, attribute) {
  return @attributeRef(@classOf(obj),
                       obj,
                       slot,
                       attribute);}
method @attributeRef (@AttributesClass c,
                      obj, slot, attribute) {
  var loc = lookup(list(slot,attribute),
                   c.attributes);
  if defined(loc) return @objectRef(obj, loc);
  else
    error("No such attribute");}
```

Accessing attributes, finally, works analogously to slots. The **lookup** call tries to find the proper entry in the attributes / attribute-location list kept in the class metaobject, extracting the attribute's location if an entry is found. In that case it uses the low-level **@objectRef** to retrieve the attribute's value from the given object.

Anything Else to Try?

The protocol has now moved well beyond the purely introspective facilities of Chapter 3 and the explicit invocation protocols of Chapter 4. Intercessory techniques first allowed the programmer to add state and behavior. Then they enabled the deeper modification of inheritance behavior. Next, the lazy slots example demonstrated how the setting and referencing of slots can be controlled with an appropriate protocol in place. Finally, the protocol was extended to provide control over how much space was allocated in objects. Note that while the programmer was inched closer and closer to the implementation in this chapter, the boundaries between programmer and implementations were never allowed to vanish altogether. The protocol for slot allocation of page 234, for instance, does not expose how object storage is allocated and where. This remains for each implementation to decide.

Using MOPs to Optimize Applications

So far, users have mostly been allowed into the implementation to extend or modify *behavior*.

What about letting them extend or modify the *implementation strategy* to improve the performance of their program?

Lazy slots were the first example in which the programmer used open implementations technology not to add or change features, but to move the substrate towards more efficient treatment of a particular application or a class of applications.

The application programmer frequently can and wants to provide valuable information to substrates about the resource usage profile of his programs—if only a channel for such communication is provided. One role of metaobject protocols is to allow for the construction of such channels, and for the ability of substrates to take advantage of the information passed in through them.

It's Such a Waste

In tracking diseases around the world, I use two classes: **`Place`** and **`Person`**. The **`Place`** class just has slots for country, city, and hospital. The **`Person`** class has many different slots describing a person's personal profile. But in any given case, only a few of these slots are needed because the other information is not available. My runtime gets unnecessarily large because of all the **`Person`** objects with hundreds of empty slots.

Here is a problem that is reasonably common. What exactly is this person is talking about?

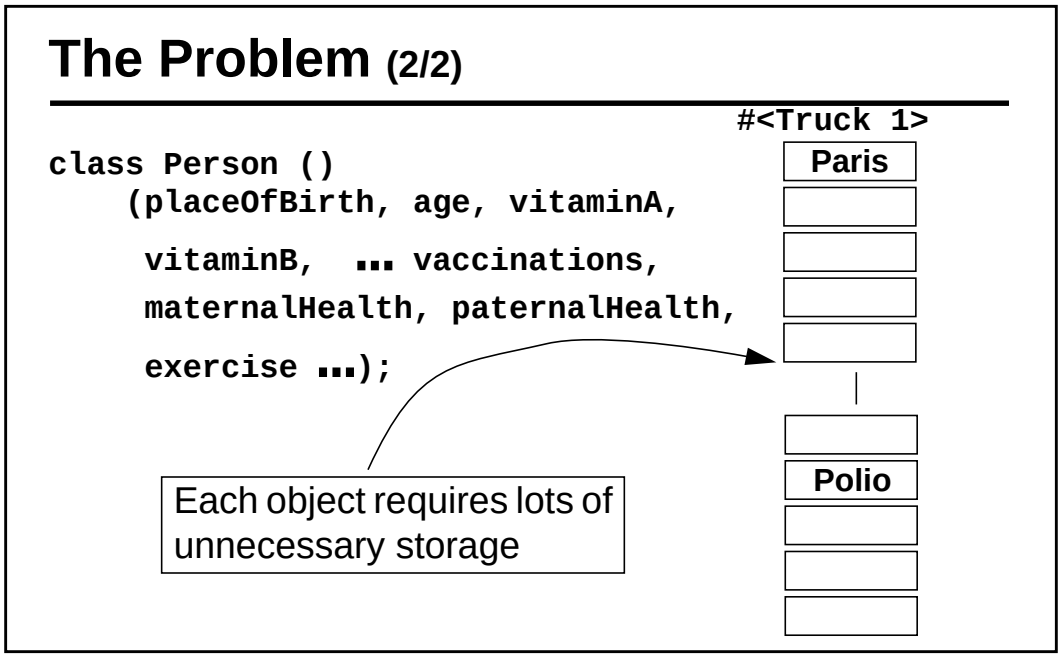
The Problem (1/2)

```
class Place ()
    (country,
     city,
     hospital);
```

For each slot, the default implementation allocates one location in the object.

This is fine for the **`Place`** class. But…

He has this unremarkable **`Place`** class, with a few slots that are filled in with relevant information.



The problem lies in the classes with many slots, only few of which are used in any given object. This might be true because the information is not available, or because the objects represent sparse matrices, or for any number of other reasons. The point is that each object is very large, because the entire set of—mostly empty—slots is allocated each time. This is wasteful.

Allowing Space Optimization

- Normally, such implementation decisions are irreversible.
- The programmer would have to implement the code some other way. But TinyObjects has so much of what is needed.
- Can an open implementation allow the programmer to adjust the default implementation to tailor for this program's performance profile?

Unless some ways can be provided for this application to run more efficiently, this programmer will be forced into developing his own data structures, losing the benefits of object-oriented programming, and maybe introducing other deficiencies as well.

In most object systems without a MOP, implementation decisions about storage strategy are irreversible. If open implementation technology lives up to expectation, this programmer should be able to do better.

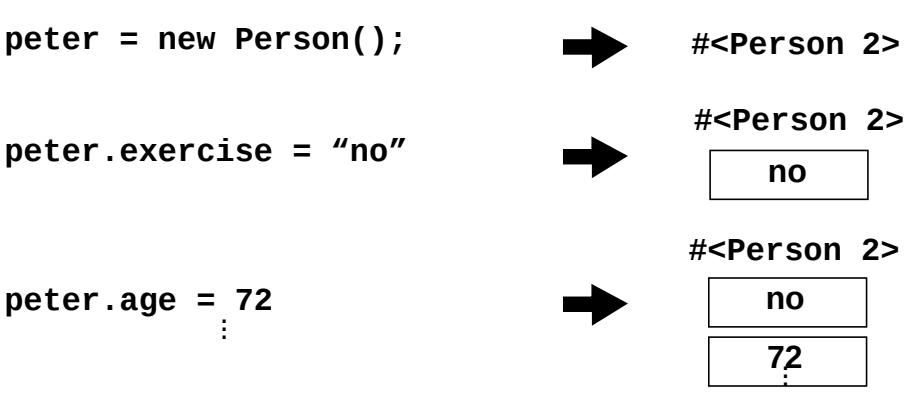
Which Implementation Is Needed?

- This programmer wants **`Person`** objects that are initially empty, requiring minimal storage, even though all the slots appear to be present. When any of those slots are set, storage is allocated for that one slot.
- Such dynamically allocated slots are called ***dynamic***

Ideally, objects would be created without any storage. Room for slot values would be allocated on demand. From the outside, this behavior would be transparent: the objects would behave the same way as regular objects.

Using Dynamic Slots

Conceptually, Objects Grow at Runtime



Here is a graphical way of showing this: The **peter** object created at the top results in an object "stub" with no associated storage (top right). As the **exercise** slot is set, one slot's worth of storage is allocated in the **peter** object. Setting the **age** slot further enlarges the object dynamically at runtime.

What Are the Issues?

- How to determine which slots should be dynamic
- How to prevent the system from allocating storage for dynamic slots
- Dynamically allocating and accessing storage

What are the subgoals the programmer needs to accomplish towards realizing this behavior? (1) A decision has to be made about the inheritance behavior of dynamic slots. This is really a piece of object model design that the programmer needs to work out. He knows best how his application would benefit most. (2) The system must be prevented from allocating the initial storage for the slots when the class is first defined and (3) the dynamic allocation itself must be made to occur when slots are updated.

Extension Design…

…Is Object Model Design

Must decide which slots in a class will be dynamic. All? None? An explicit option?

Decide that all slots defined directly in a class marked as dynamic will be dynamic slots. One could, of course, do something more fancy.

On the first point, the programmer decides for the simplest solution: All the slots defined on a dynamic class will automatically be dynamic. More fine-grained solutions are, of course, imaginable. One could, for example, introduce options at the slot level, which would allow individual slots to be marked as dynamic.

Here is how dynamic slots could look to the programmer.

Classes with Dynamic Slots

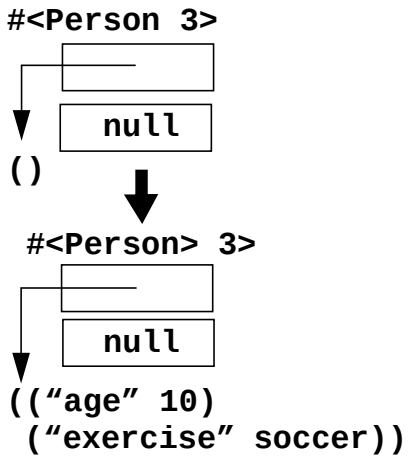
```
class Creature ()
    (soul);

class Person (Creature)
    (placeOfBirth, age, vitaminA,
     vitaminB,... vaccinations,
     maternalHealth, paternalHealth,
     exercise...)
  @DynamicClass;
```

All but the **soul** slot will be dynamic.

In this example all the **Person** slots will be dynamic. But dynamic classes may inherit from non-dynamic ones, so the **soul** slot will be treated normally, having storage allocated for it right away whenever a **Person** object is created.

Storage Layout for `Person` Objects

```
peter.age = 10
peter.exercise = "soccer"
```



Here is a conceptual depiction of an appropriate implementation for this behavior. Initially, the **`Person`** object above has only two fields of storage: one to hold a data structure that will contain all the values of dynamic slots as they are set during runtime, the other to hold the **`soul`** value.

When the **`peter`** object's age slot is set to 10, the dynamic values data structure starts to get filled with a slot-name/value pair: **`("age" 10)`**. Alternative data structures could, of course, be used to satisfy different performance requirements.

Whenever a slot is set that was never set before, a new pair, such as **`("exercise" soccer)`** will be added to the dynamic values data structure. If a slot has been set before and is updated, the value will be modified in place.

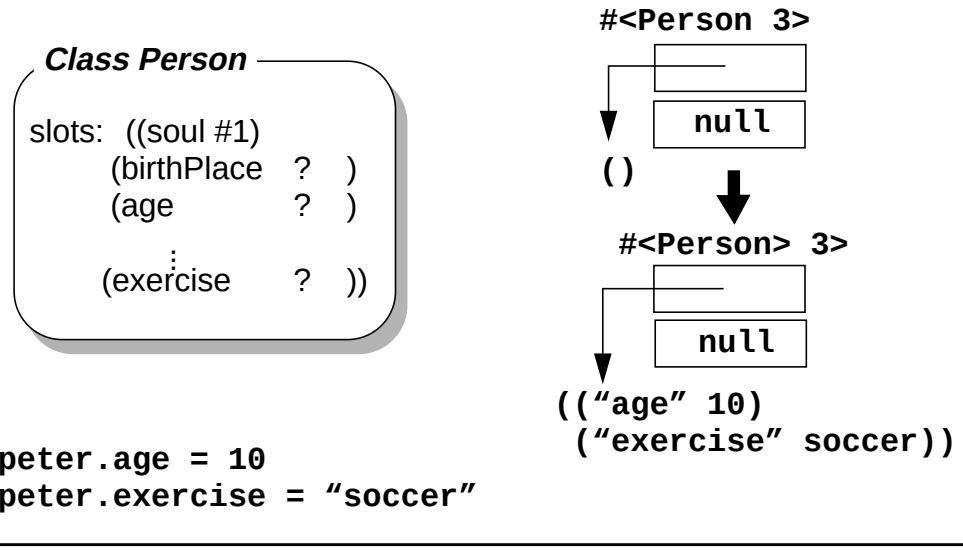
How to Prevent Storage Allocation

There is a need to…

- Request one location in each future object to hold a datastructure for dynamic slot values
- Override **`@computeSlots`** to not allocate individual locations for dynamic slots
- Specialize **`@slotRef`** to deal with both dynamic and regular slots

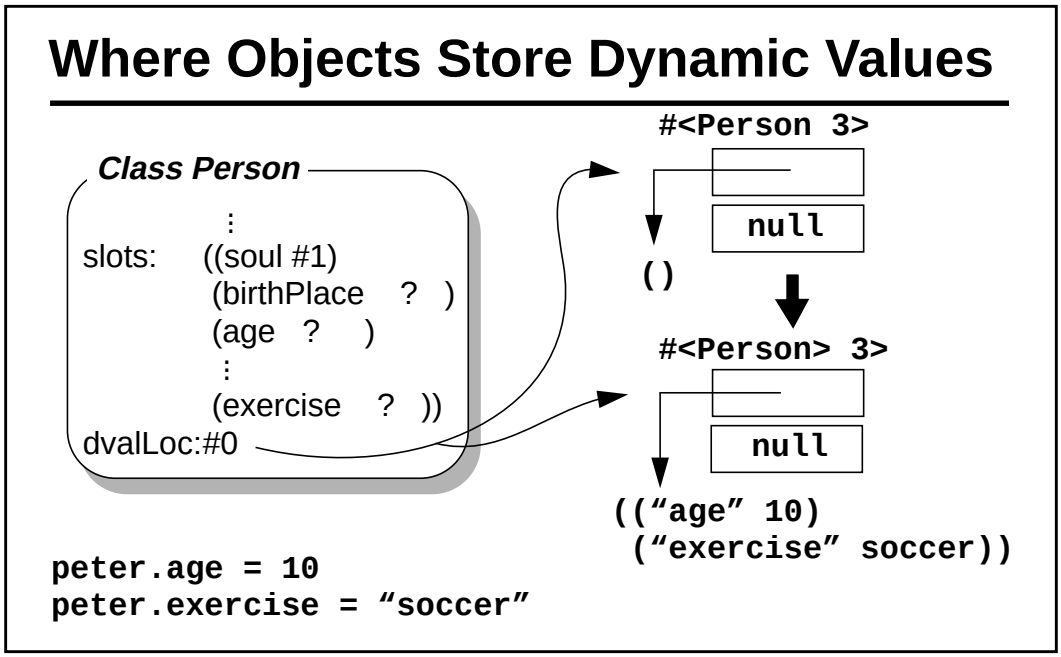
Regarding slot allocation, there are three subissues to consider: one additional storage location must be requested to be allocated in each future object to hold the values of dynamic slots. This is like slipping an invisible slot into those objects. Then, a method on **`@computeSlots`** must be written which will prevent future objects from having space allocated for the dynamic slots. Finally, a new **`@slotRef`** method must extract slot values from the right places in the object.

Storage Map for Dynamic Classes



Towards finding solutions to slot allocation management, let us get back to viewing classes as storage maps. On the left of the figure above is the first draft of a map view for a class with dynamic slots. The **soul** slot is represented normally, as a name/location pair. The dynamic slots do need to be represented in the class, because browsers and other tools need to see them, even if they have never been set before in a given object and therefore do not have an allocation there. Yet they will not have an offset location associated with them in the class. Something will need to replace the location part of the slot information for dynamic slots in the map, which is why the location spot for the dynamic slots are left blank in the figure.

A place in the class to remember the location of the dynamic values in objects is also needed: in this example, it happens to be first, but programmers cannot count on that.



The **`dvalLoc`** in this updated view will be used for this last purpose. It allows meta programs to find the dynamic slot-name/value pairs for any object of the **`Person`** class.

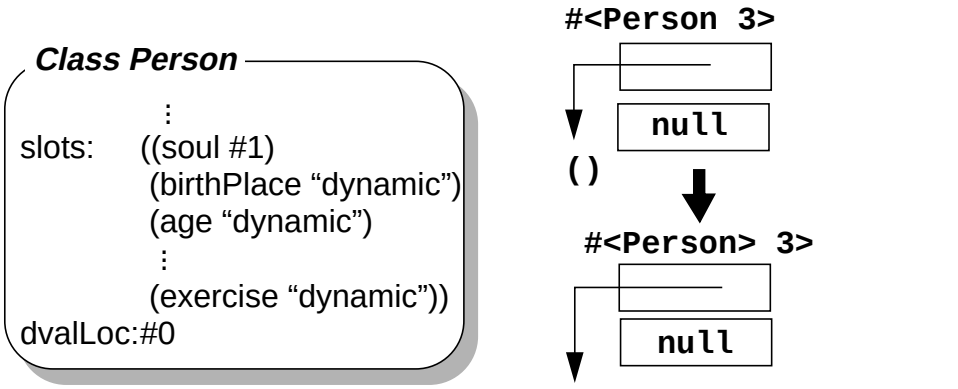
Building the Storage Map

```
class @DynamicClass (@Class)
    (dvalLoc);

initializer @DynamicClass (c,
                           className,
                           directSupers,
                           directSlots) {
  c.dvalLoc = @allocateLocation(c);}
```

Adding this space in the class metaobject is easily done with intercession, so we get that task out of the way right here.

During the initialization of dynamic classes, **@allocateLocation** requests a location, which is then saved in the class metaobject (**c.dvalLoc**). Remember that **@allocateLocation** does not allocate any space right away. It simply makes the promise that one additional space will be allocated in every future objects of the class **c**, and it returns a location which, when handed to **@objectRef** or **@objectSet** with an object, will access that location in that object.

Marking the Dynamic Slots




```
peter.age = 10
peter.exercise = “soccer”
```

Since dynamic slots have no normal locations, they need to have a non-standard appearance in the storage map so that they can be recognized when slots are accessed at runtime. Instead of a storage location, the value part of entries for dynamic slots in the class will therefore contain the string **“dynamic.”**

Completing the Storage Map (1/2)

```
method @computeSlots (@DynamicClass c) {
   var result = list();
   foreach (slotName, @allSlotNames(c)) {
      var tst = slot should be dynamic?;
      var loc;
      if defined(tst)
         loc = "dynamic"
      else
         loc = @allocateLocation(c);
      extend(result, list(slotName, loc));}}
```

Here is the **@computeSlots** method for dynamic classes which puts all this information together when a new class is being initialized. Given a dynamic class metaobject **c**, it builds the list of slot-name/location-information, collecting all the slots defined in **c** and all the ones that come down from **c**'s superclasses. It checks each slot to see whether it is dynamic. Depending on the decision, it fills in the slot information in the class metaobject: a regular location, or the string **"dynamic."**

Completing the Storage Map (2/2)

For this extension, the *should be dynamic?* test would go through the CPL to look for the first class in which the slot is directly defined. If that class is dynamic, so is the slot.

A different design is possible. This predicate could even be a generic function itself.

Dynamic Slot Testing

```
function @isDynoSlot (c, slotName) {
   var slotLocs = @classSlotLocations(c);
   var loc = lookup(slotName, slotLocs);
   return (loc == "dynamic");}
```

Here is a predicate which, given a class metaobject and a slot name, will be true or false, depending on whether that slot is dynamic.

Note that **`@computeSlots`** now incorrectly began to return a datastructure which is different from what has been required since the MOP specification on page 230: the location information for slots may now sometimes be the string **`"dynamic."`** The protocol will therefore have to be made slightly more liberal. But not before an analysis is made as to which parts of the protocol are affected by this new form of slot information. It turns out that **`@slotRef`** and **`@slotSet`** are the generic functions whose methods make use of the slot information. If the form of that information is changed, they must be updated too.

Accessing Dynamic Slots

```
method @slotRef (@DynamicClass c,
                 obj,
                 slotName) {
   if (@isDynoSlot(c,slotName)) {
      var dvalLoc = c.dvalLoc
      var dvalues = @objectRef(obj, dvalLoc);
      return lookup(slotName, dvalues);}
   else
      return callNextMethod();}
```

Here, then, is an **@slotRef** method for dynamic classes. It checks whether the slot to fetch from is dynamic. If yes, it retrieves from **c** the location where dynamic slot values are kept in **c**'s objects. Using this location, it reaches into the object through **@objectRef** to retrieve the slot/value pairs. The call to **lookup** returns the value for the slot.

If the slot to be fetched is not dynamic, this method defers to the default way of doing slot access business (**callNextMethod**).

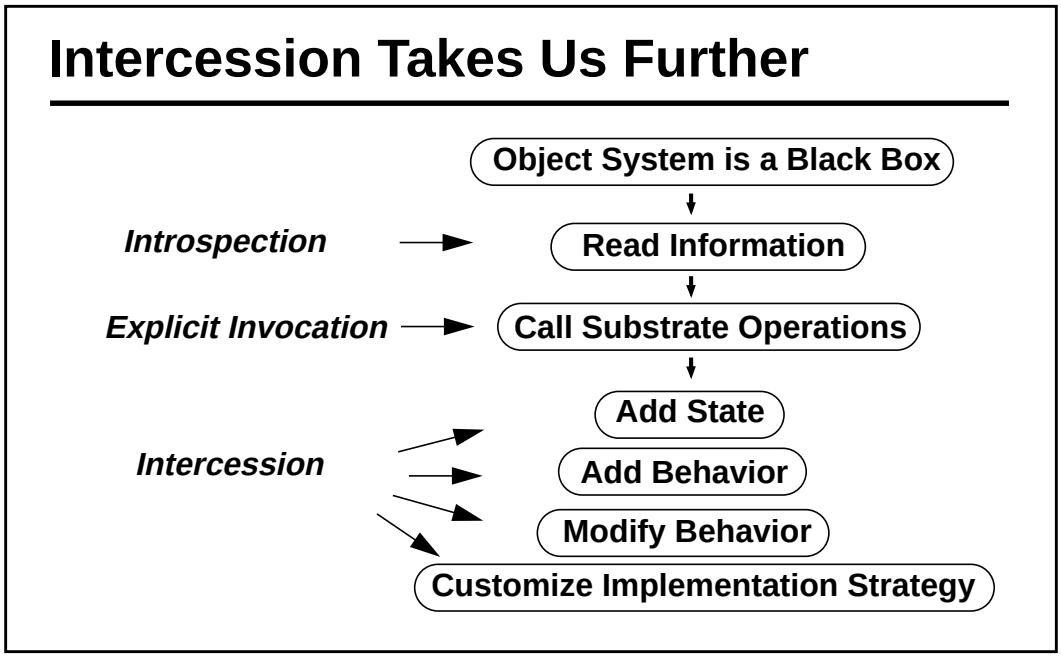
It is thus easy to write an **@slotRef** that handles the alternate slot information produced by the new **@computeSlots**. But the necessity to write such a method must be documented.

Allowing User-Defined Slot Maps

The **@computeSlots** GF must return a list of pairs (name/allocation). If allocation is not a location returned by **@allocateLocation**, then a method on **@slotRef** and **@slotSet** must be defined to handle that slot.

Note that the new **@slotRef** of page 258 does its job correctly, because it calls **callNextMethod** only after ensuring that it is accessing a standard slot.

This protocol is an example for a phenomenon often encountered in the development of metaobject protocols. As with the **@computeSlots**, **@slotRef**, **@slotSet** triplet above, it is often necessary to require that MOP users keep their methods on two or more generic functions consistent with each other. Such requirements must be an explicit part of the protocol.



Here is a map of where we are. We introduced introspection for satisfying applications needing information about a substrate. Explicit invocation enabled programs to cause operations to be performed in the substrate, without necessarily going through the regular syntax.

Intercession then allowed application programmers to add pieces of state and behavior to the substrate. More radical forms of intercession, finally, enabled the modification of substrate behavior and implementation strategy choices.

Before continuing, let us quickly check whether any more rules of behavior need to be added to the MOP, now that we added a bit more depth to what programmers are able to do.

What about Our Rules?

For selected intercessory points, the MOP allows system method overriding with relevant semantic constraints. Examples: **`@computeSlots`** and **`@slotRef`**.

Note: It is still illegal to change methods of built-in system classes.

Recall that up to intercession, programmers were not allowed to override system methods. All they could do was to extend the substrate by subclassing metaclasses and writing methods specialized to those subclasses.

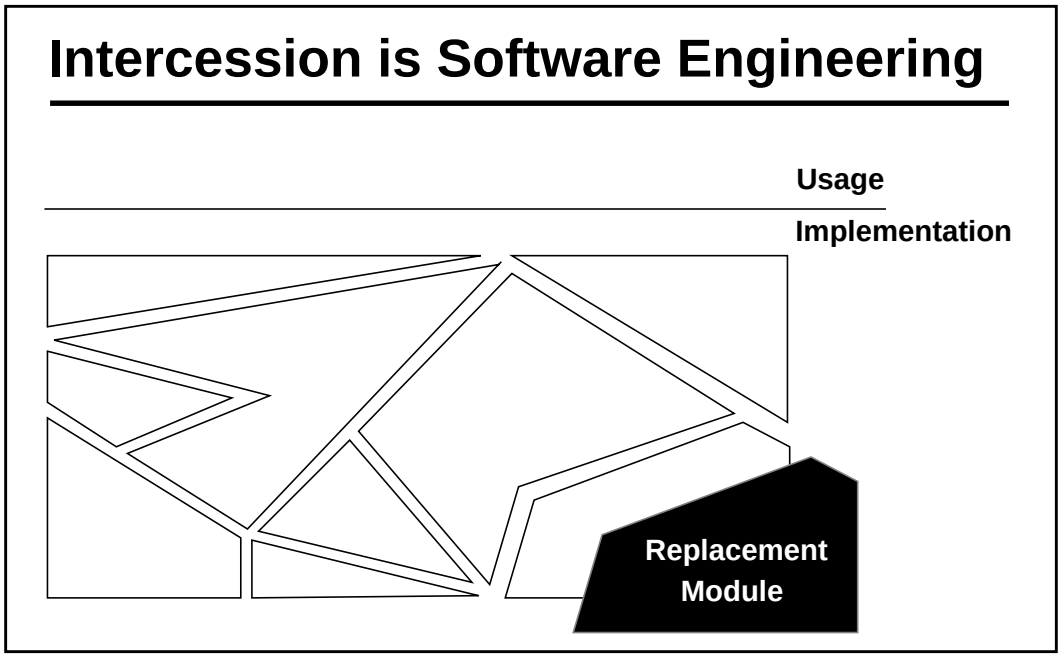
In the context of intercession, the protocol sometimes allows the overriding of system methods, that is the modification of existing substrate behavior, provided this is explicitly stated in the protocol, and relevant semantic constraints are satisfied. We first saw this with **`@computeCpl`** whose system method was allowed to be replaced for **`@FlavorsClass`**. The same is now true for the **`@slotRef`** methods on dynamic classes. This relaxed condition must, however, be specified in the protocol, which was done on page 230.

Rules for Intercession

For the purpose of intercession the “no method override” rule is relaxed for specific system GF’s:

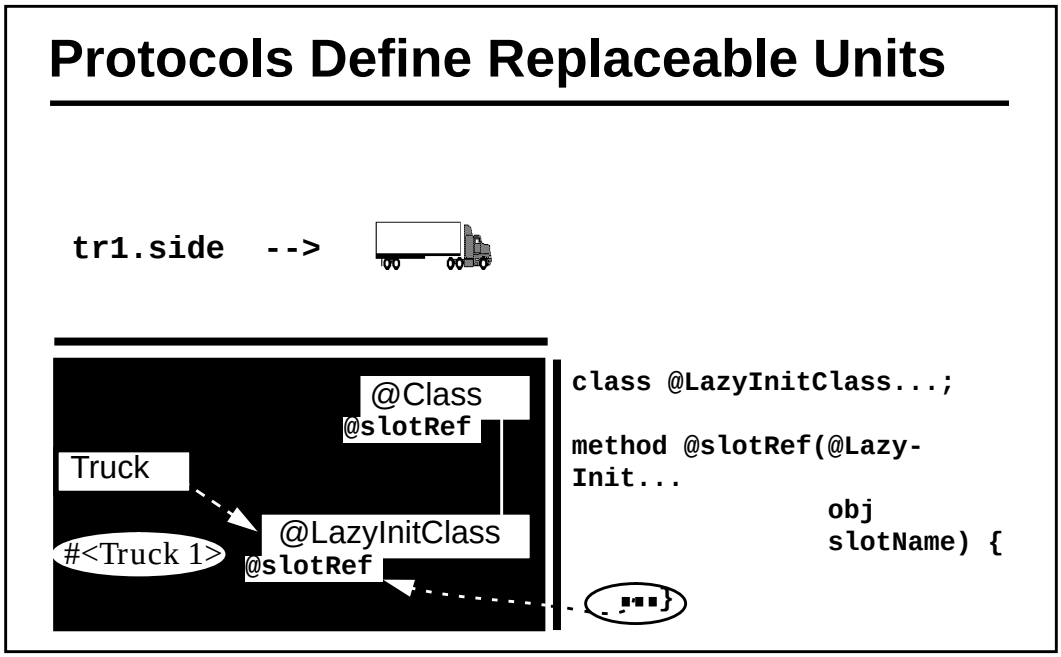
- **`@computeCpl`** may be overridden, but the CPL returned must include all supers and **`Object`**. (Note that this is checkable).
- **`@computeSlots`** may be overridden but must obey rules, and **`@slotRef`** must stay consistent.
- **`@slotRef`** may be overridden.

Here is a summary of the overrides that were made legitimate in the MOP extensions for intercession.



At first sight, intercession looks like classic software engineering: the substrate is partitioned into appropriate modules, isolated from each other through interface specifications. Ada, for instance, is a language founded on this principle. If the partitioning and interface designs are done well, modules may be replaced at will, without disturbing the overall operation of the substrate and the applications based upon it.

But intercession goes one step further: it adds scope control.



As we saw in the **`@slotRef`** example reproduced above, MOP technology allows programmers to replace a piece of substrate code, just like the modularity techniques of traditional software engineering. But while the replacement in the previous figure replaces the code for everyone, this **`@slotRef`** adjustment is visible only to users of the **`@LazyInitClass`**.

This is a key difference. The module replacements in a traditional software engineering environment are just as much required to present a compromise among multiple application demands as the modules they replace, unless the substrate is being specialized to serve a narrow application community exclusively. The MOP techniques presented here allow multiple applications or parts of applications to be served by one substrate, but under the illusion that the substrate has been customized to serve each set of particular needs.

Intercession Rules

Reliance on basic SE principles

- Rules ensure that modules fit together
- They guarantee independence from implementation details

The rules associated with intercession ensure that the substrate modules will still fit together, after some of them have been modified or replaced.

Summary

- **Introspection**—examination of selected system internals.
- **Explicit Invocation**—enabling programs to explicitly invoke substrate operations, bypassing syntax in some cases.

  Intercession—metalevel specialization that adds to or modifies existing features of the base object system:
  - Adding State
  - Adding Behavior
  - Modifying Behavior
  - Adjusting Performance

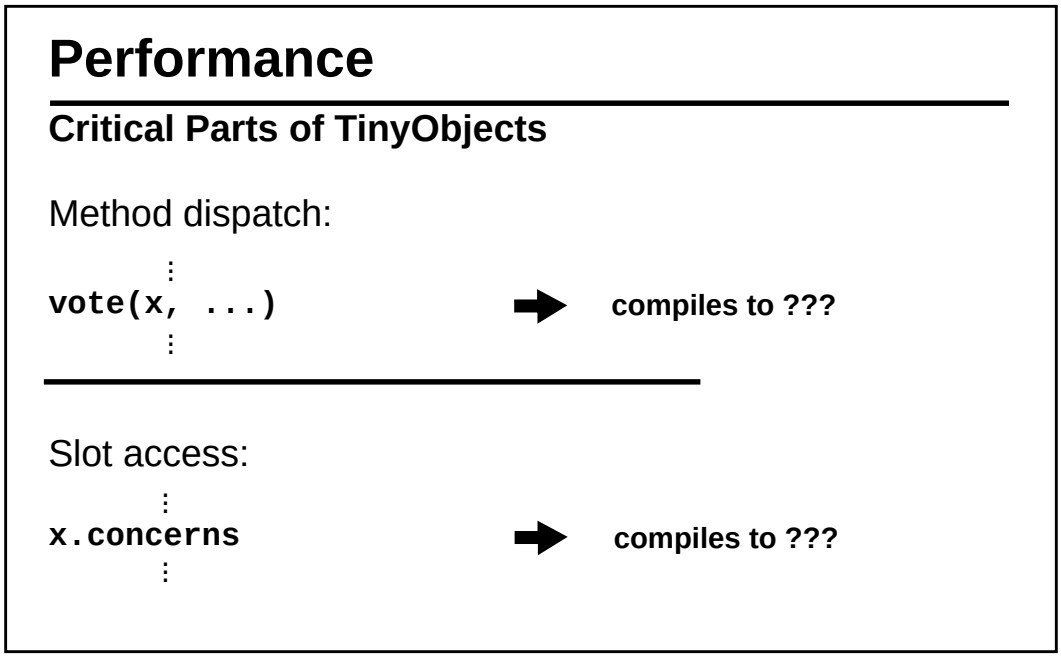
All the elegance, programming and maintenance convenience provided by the open implementation techniques presented would be useless if they could not be realized efficiently. Implementors would simply refuse to provide implementations that support the protocols.

The following chapter will therefore take a good look at this central aspect: how can MOPs be designed to allow efficient implementations?

Chapter 7 And I Want It All to Be Fast!

The question of performance has become more and more pressing as MOP designers pushed more and more information and, especially, operations from compile time to runtime. As things stand now, programmers have a very flexible substrate, but their applications will be too slow.

In this chapter we will recover performance by a synthesis of techniques known as *incremental specialization*. We will introduce some additional techniques for designing protocols and show how standard compiler techniques can be applied to MOP design and implementation.



Ignore the MOP for a moment and consider what is important with regard to performance when implementing just regular TinyObjects. There are two main operations that object systems frequently perform at runtime and that consequently need to be fast: method dispatch and slot access. Instance creation is important as well, but techniques for the former two will also cover instance creation.

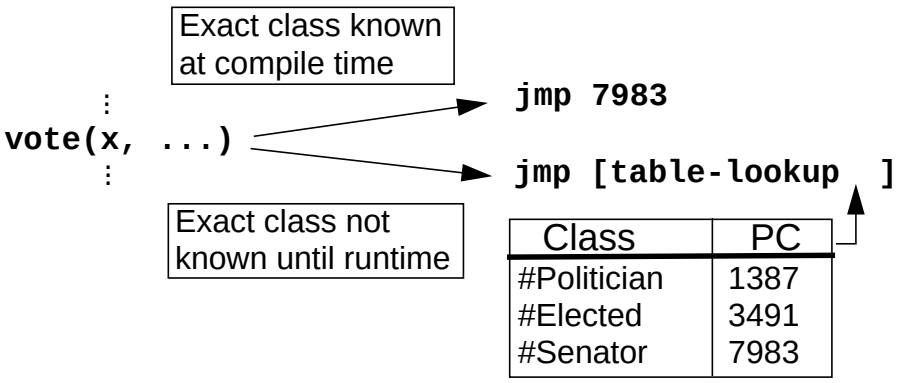
Guiding our exploration in this chapter will thus be the two questions above: what should statements like **`vote(...)`** and **`x.concerns`** compile to in the presence of a MOP?

Let us look first at efficient implementations of these two operations, in the absence of a MOP. This will give us a performance marker against which we will be able to measure the MOP implementation solutions we come up with.

Method Dispatch

Traditional Optimization

Precompute as Much of Lookup as Possible.



Looking at dispatch first, how can implementations be made efficient? Traditionally this is done by following the standard technique of pre-computing as much as possible. The best case for the **`vote`** call above is when the compiler can tell exactly which class **`x`** will be an object of. In that case, an inline jump instruction is all that is left of the dispatch operation.

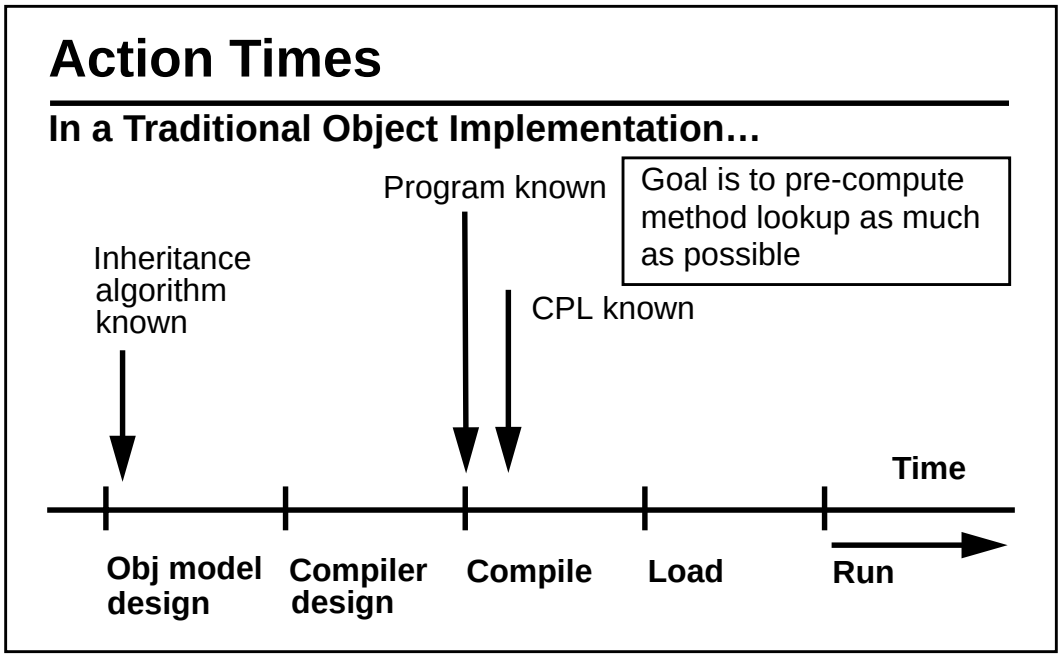
Sometimes it is not clear at compile time which exact class **`x`** will be of, but the compiler can ensure that it will be one of a set of classes. In this case, a table can be constructed which maps classes to jump destinations. At runtime, the code inlined by the compiler accesses the table, looks up the class **`x`** turns out to be an object of, and jumps to the correct location.

In practice, details such as the choice of information in the table and techniques for table lookup matter and need to be considered carefully. But in principle, the technique shown here is what makes method dispatch fast in standard object-oriented object systems.

Why Does This Approach Work?

…because object model semantics are fixed, and the program is known. The implementation can therefore determine when it is safe to memoize a shortcut.

Why are compiler writers allowed to take this shortcut around method dispatch? They may do it because the combination of generic function name and the class of the argument uniquely determine which method to run. In other words, method dispatch does not depend on the time of day or the phase of the moon, but just on these two quantities.



Another way of looking at this is by using the timeline we saw earlier. In traditional object systems the inheritance algorithm is fixed at design time. When a particular program is read into the compiler, an analysis of the class structure leads to the CPL of each class. Once a class' CPL is known, dispatch can be resolved just based on the class of the argument to a generic function call. The time at which this resolution takes place is up to the implementation. The point is that it can be done at compile time or later, just as long as the CPL is known.

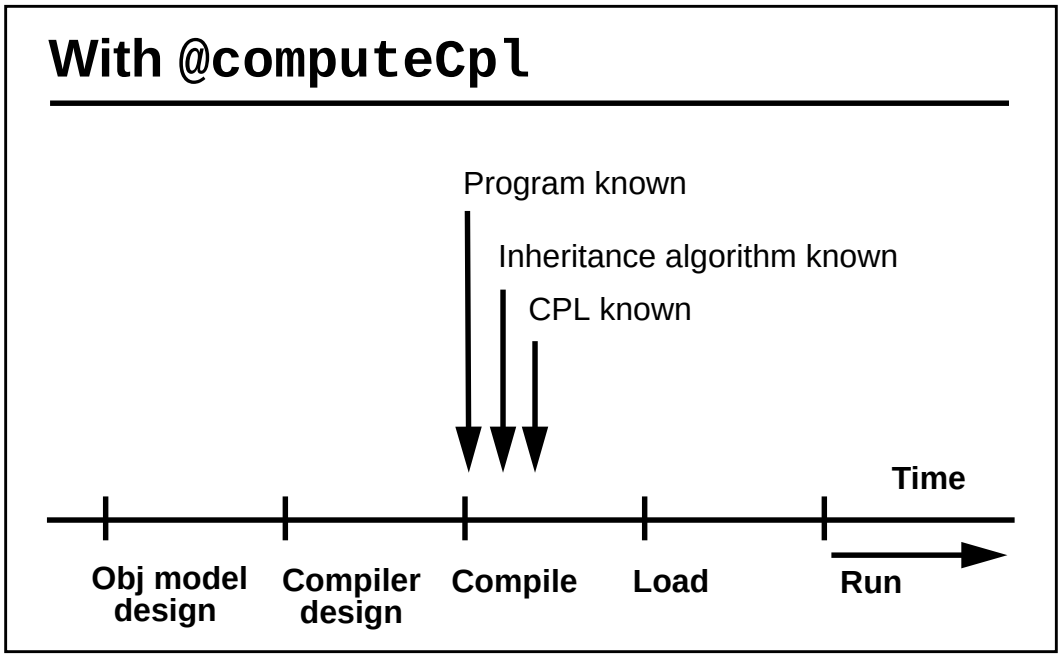
But What about `@computeCpl`?

Has the flexibility introduced with the programmer's ability to provide methods on **`@computeCpl`** destroyed the ability to optimize method dispatch?

No, because the protocol requires the CPL to be computed early enough and to remain unchanged.

The question is whether the introduction of **`@computeCpl`** as a generic function to which users may add methods has invalidated these basic facts that enable compilers to take advantage of the shortcuts on page 269.

The answer is "no" because the timeline that takes **`@computeCpl`** into account is almost exactly the same as before...



The inheritance algorithm is no longer known at object model design time, because user methods on **`@computeCpl`** may change it. Only certain constraints on it are known at that early time, such as **`Object`** being at the root of the inheritance tree (see the rules on page 187).

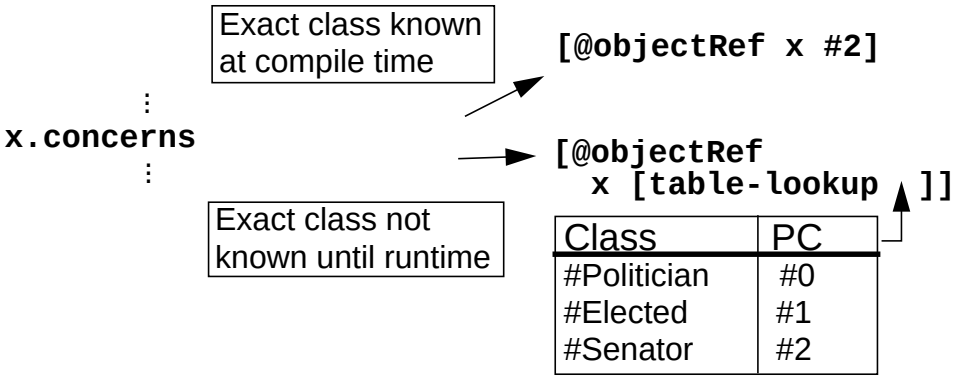
Instead, after the program is read at compile time, the inheritance mechanism becomes available once the class metaobjects are created and the—possibly non-standard—**`@computeCpl`** methods have done their job. At that point, the CPLs are available, encoding within them all the implementation needs to know about inheritance.

Note that the CPL is known at about the same time as before. The MOP has therefore not invalidated the assumptions underlying the dispatch optimizations of page 269. Note as well the less obvious point that this scheme requires the ability to run **`@computeCpl`** at compile time. The reason this is not a problem goes back to the CPL rules on page 187 where MOP designers required CPLs to be functions of the class graph only, which is available at compile time. In this case of opening the inheritance implementation, careful protocol design therefore preserved the necessary optimization opportunities.

Traditional Optimized Slot Access

Same Approach As Method Dispatch

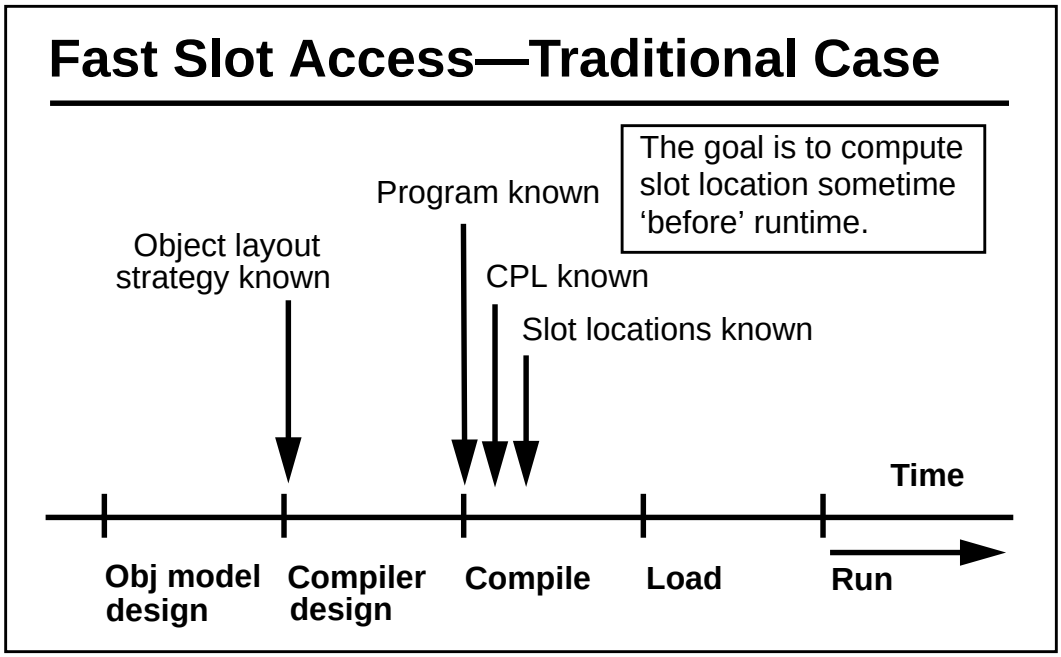
Precompute as much as possible:



Looking at slot access, we will see that matters do not work out quite as well as with the inheritance intercession we just examined.

Traditional slot access optimization techniques work analogously to method dispatch: if the respective object's class is known at runtime, access to a known offset within the object's storage can be inlined. Otherwise, a class-based table lookup for the correct offset needs to happen at runtime.

Whether the class is known early or late, a fast implementation can therefore be provided.



This works because in traditional object implementations the strategy for managing storage for objects is known at compiler design time. Once the program and the CPLs are known, slot locations will therefore be known as well, and a class uniquely determines where in one of its objects a given slot can be found.

We Committed to Huge Overheads!

The specter of `@slotRef`

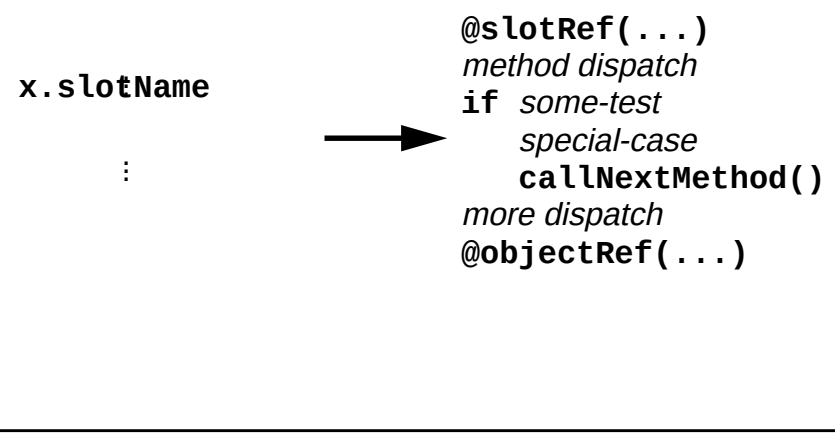
Recall this spec:

> **`@slotRef`** is the generic function that is called to implement slot access. It receives as arguments the class of object, the object and the slot name.

A generic function call for each slot access!

The MOP developed so far thoroughly broke this sequence. Its specification required that every TinyObjects implementation call **`@slotRef`**. Let us look at what this means.

HUGE Overheads



The call to **`@slotRef`** requires a method dispatch at the metalevel, because a method may have been provided on the respective class metaclass. Often, user-defined methods will check whether the slot being accessed is one of the special ones they know how to manage. If not, they will generally defer to the system default, requiring yet more dispatch activity, until, finally, the actual object access is performed.

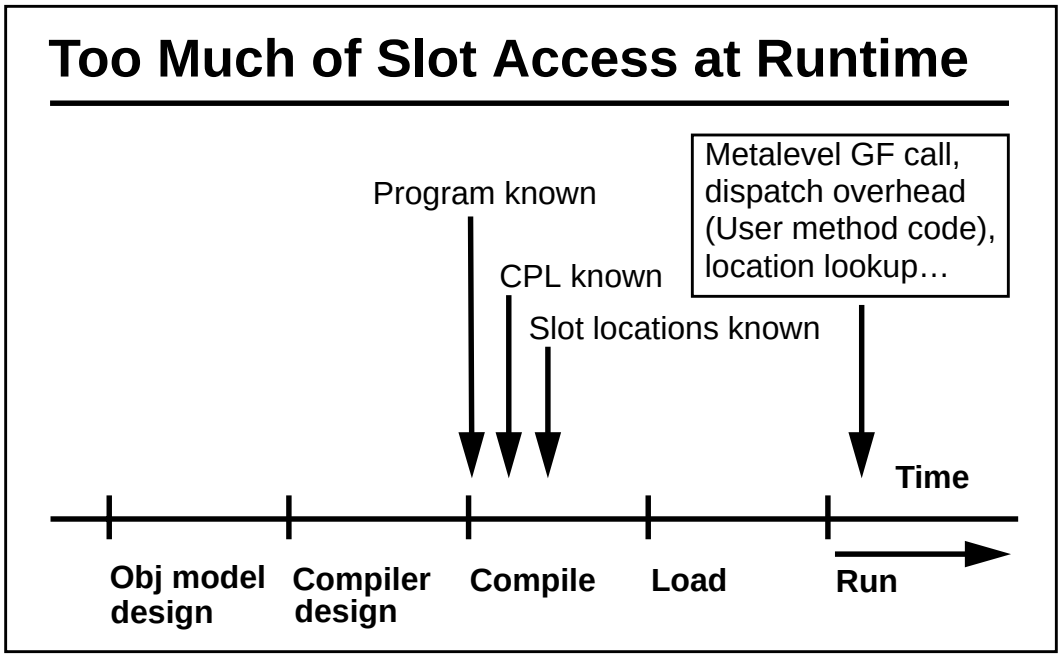
Case in Point:

Dynamic Slots...

```
method @slotRef (@DynamicClass c,
                 obj,
                 slotName) {
  if (@isDynoSlot(c,slotName)) {
    var dvalLoc = c.dvalLoc
    var dvalues = @objectRef(obj, dvalLoc);
    return lookup(@slotName, dvalues);}
  else
    return callNextMethod();}
```

This worked, but…

The dynamic slots of Chapter 6 are a good example. Once the dispatch to the **`@DynamicClass`**' **`@slotRef`** is complete, the—potentially lengthy—**`@isDynoSlot`** runs. If this test comes out negative, the **`callNextMethod`** produces additional dispatching costs.



Here is this bad news seen on the time line. The problem is that the object layout strategy is known only at runtime.

Even if the code within **`@slotRef`** was fast, just the dispatch would be much too expensive. Everything is happening too late and on every single slot access.

Let us start by focusing on the meta-level dispatch overhead. We will do this by considering only the system method for a moment.

A Closer Look at Slot Access

The System's `@slotRef` Method

```
obj.state
    ⬇
@slotRef(@classOf(obj), obj, "state")

method @slotRef (@class c, obj slotName) {
  var slots = @classSlotLocations(c);
  var loc = lookup(slotName, slots);
  return @objectRef(obj, loc);}
```

In the case of default slot access, an expression like **`obj.state`** compiles to **`@slotRef`** as shown. That default method finds the list of slot/location pairs in the class metaobject, pulls out the appropriate location and fetches the respective value from the object.

The goal is to reduce this process to what is shown on page 275.

What Can Be Precomputed?

What if the class and slot name are known at compile time? (Same assumption taken by traditional object system implementations.)

What can be precomputed?

This reduction of runtime activity hinges on the implementation's ability to precompute at least some of what is currently happening at runtime. A closer look reveals that neither the system nor the **`@DynamicClass`** **`@slotRef`** code are taking advantage of what is known or computable ahead of time.

Standard optimization techniques are all predicated on the assumption that they can learn early about the class and the slot name involved in a slot access. Therefore, if we where also to require those two quantities to be known, we would be no more restrictive than traditional systems. These two assumptions are a kind of upper bound on restrictions we allow ourselves to make in the following.

Manually Pre-Computing Slot Access

obj.state

If class is known to be **Senator**
so class of class is **@Class**

```
@slotRef(@classOf(obj), obj, "state")

method @slotRef (@class c, obj slotName) {
  var slots = @classSlotLocations(c);
  var loc = lookup(slotName, slots);
  return @objectRef(obj, loc);}
```

Residual: **@objectRef(obj, #2)**

Here is a copy of the **@slotRef** system method with all the pre-computable or pre-known information underlined for the case when **obj**'s class is known at compile time. In that case, the **@slotRef** method to run can be determined at compile time, because the relevant class metaobject can be obtained from the class.

Proceeding into the system **@slotRef** method, compile time possession of the class metaobject allows implementations to retrieve the list of slot-name/locations early on as well. This in turn results in advance knowledge of where in the object the respective slot will be located.

If one were to pull out all the underlined advance information, only the **@objectRef** call would be left to do at runtime. In an analysis like this, the work left over for runtime is called a *residual*.

Making no more restrictive a set of assumptions than compiler builders for traditional object implementations thus took us to the same runtime complexity they end up with. We still have to figure out how to take advantage of these insights in the context of our MOP. But first let us see how the analysis works out for user-defined **@slotRef** methods.

Optimizing User-Defined Slot Access

Precomputing As Much As Possible

If class is known to be **Person** so class of class is **@DynamicClass**.

```
obj.name
@slotRef(@classOf(obj), obj, "name")
method @slotRef (@DynamicClass c,obj,slotName) {
  if (@isDynoSlot(c,slotName)) {
    var dvalLoc = c.dvalLoc
    var dvalues = @objectRef(obj, dvalLoc);
    return lookup(@slotName, dvalues);}
  else
    return callNextMethod();}
```

Residual: **lookup("name", @objectRef(obj, #0))**

or: **@objectRef(obj, #n)**

The story in this case is very similar (refer to page 252 if you need a refresher on this code). Compile time knowledge of **obj**'s class allows early determination of the correct **@slotRef** method to be run. Within that method, the **@isDynoSlot** can be executed at compile time as well, because it also depends merely on a knowledge of class and slot name. Accessing the class metaobject's information on where in its class' objects the dynamic slot values are stored can be done early as well (**c.dvalLoc**).

What is left to do at runtime depends on the slot: if it is dynamic, the object's dynamic values data structure must be retrieved (**@objectRef(obj,#0)**) and the value must be found within it (**lookup**).

If the slot is not dynamic, the implementation ends up with the residual that would result from the **callNextMethod**. This is simply the system's default slot access which we have previously analyzed to have the residual **@objectRef(obj,#n)**.

In other words, if implementors just make the same assumptions everyone else makes—compile time knowledge of class and slot name—and perform the underlining process appropriately, implementors of TinyObjects can be as efficient with the MOP added as without it.

Two Strategies for Pre-Computation

1. Leave protocol unchanged and have implementation do pre-computation automatically—constant-folding, inlining, partial-evaluation techniques
2. Support optimizations directly by "currying" the protocol to fit into standard optimizers

The question is how these optimization opportunities can be realized, and how they relate to the MOP.

The two possibilities are to accomplish all the pre-computations automatically, or to modify the protocol a bit to provide for pre-computation explicitly. The first possibility would, of course, be preferable. But it could in general involve unmanageable complexities, as code analysis of user-defined **`@slotRef`** methods would be required. An implementation would, for instance, need to analyze successfully whether predicates such as **`@isDynoSlot`** may run at compile time or not. Components which attempt this analysis are called *partial evaluators*, and they are subject of research in many places (see bibliography).

In order to avoid difficulties with very general user-defined methods where automatic analysis would be very difficult, we choose the second alternative.

A Change of Protocol

Letting Programmers Do the Underlining

`@slotRef(c, obj, slotName)`
is the GF that is called to implement slot access and returns a base level result.

`@slotRefResidual(c, slotname)`
is a GF that is called at class initialization time. It returns the residual of slot access to that slot in that class. This residual will be invoked at slot access time.

The MOP designers will change the protocol a little bit, to make the equivalent of underlining explicit. Instead of providing **`@slotRef`** which runs every time a slot is accessed, the new protocol would present the generic function **`@slotRefResidual`** instead. Methods on this generic function would be called once for each slot at compile time. Methods on **`@slotRefResidual`** would have to return the kind of fragment we saw in the underlining analysis. This would allow programmers to perform the analysis themselves, and to provide the result to the implementation at the right time.

Here is what the TinyObjects system method would look like under this new protocol.

Example

The Standard System Method

```
method @slotRefResidual (@Class c, slotName) {
  var slots = @classSlots(c);
  var loc = lookup(slotName, slots);
  return [ @objectRef(obj, loc); ];}
```

Remember, this method does not do the actual slot access. It simply returns a fragment which will be invoked at runtime to accomplish the slot access then.

The residual to be returned is shown in square brackets. One way to return a residual is to package it as a function. We will talk about other possibilities a bit later on.

The code above, running sometime during class initialization, first retrieves the list of **`c`**'s slots. From it the slot's location is obtained. When the method returns the residual, the **`loc`** variable within it will have been replaced with the actual location.

So, let us have our MOP designers document this new protocol.

A Curried Slot Access Protocol

During class initialization the GF:

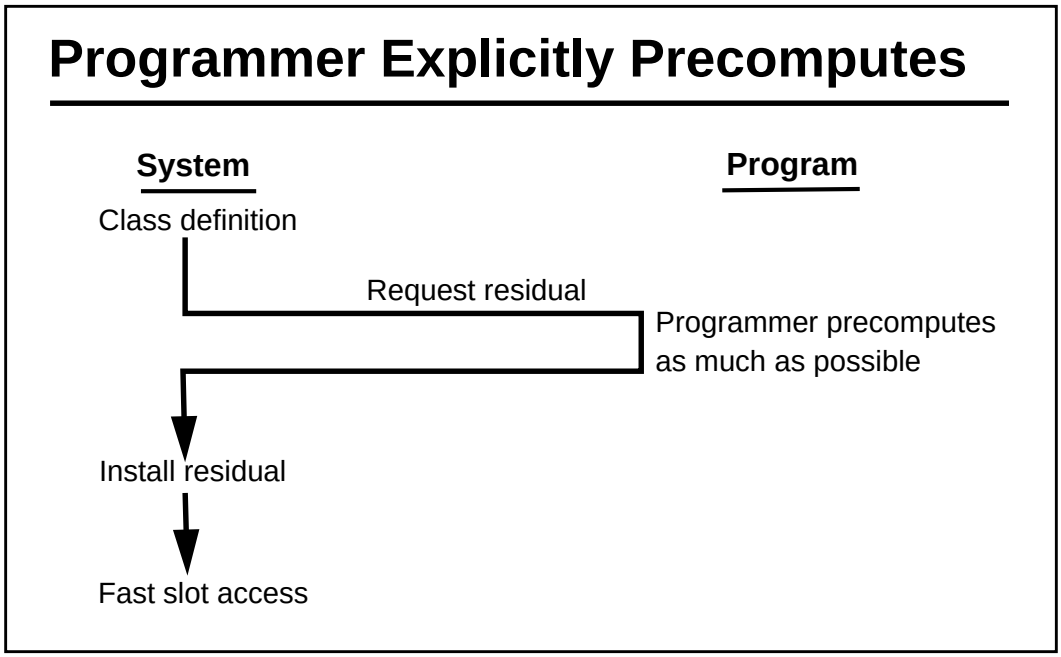
```
@slotRefResidual(c, slotName)
```

will be called once for each of the class's slots, after the CPL and slots have been computed and stored. It must return a residual which accepts objects of the class and produces the value of the slot **`slotName`**.

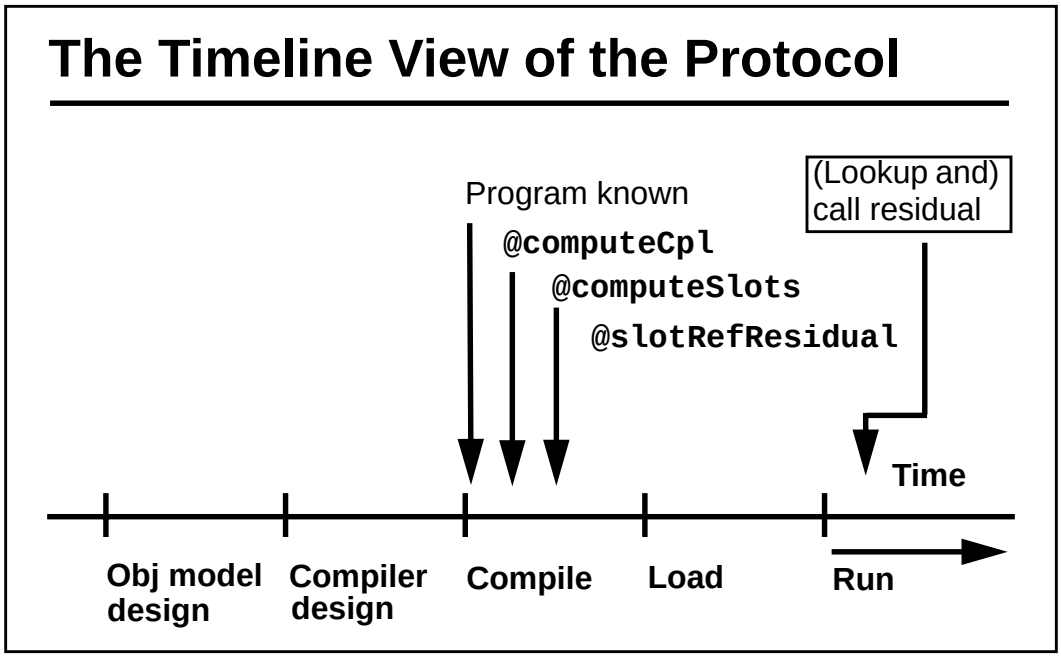
A curried protocol partitions the information needed to perform some frequent operation into two groups: information that will be the same for successive invocations, and information that will typically change each time. In our example, the static information for slot access operations to objects of a given class consists of the slot name and the class metaobject. The information that varies between invocations is the object being accessed.

Based on this partitioning of information, the actions required to perform the operation are partitioned into two groups as well, one that depends only on the static information and the other that may depend on all the information. The first group is performed only each time the static information changes, and the results are saved, so that only the second group needs to be performed each time the operation is invoked. In our example the first group of actions is the test whether the respective slot is dynamic, and the retrieval of the location where dynamic slot values are stored in objects. The second group is the actual slot access.

When a protocol is curried, it is re-organized to take advantage of these time savings. In our case, this meant introducing **`@slotRefResidual`** as an opportunity for programmers to manually perform the partitioning.



In this sense, protocol currying is a kind of manual partial-evaluation, allowing the user into the loop. It helps get around the fact that it is hard to analyze code automatically. In terms of our timeline visualization, the new protocol's effect looks like this...



Here is what the new protocol has done. It has pulled all slot access activity forward in time to when class initialization happens, except for those pieces that must truly occur at runtime. All that must happen there is to look up the residual and to call it.

Dynamic Slots Need to Be Reworked

- Dynamic slots were implemented with **`@slotRef`**. We need to rework that extension using the new protocol
- Recall: Dynamic slots had storage allocation only when needed

There are two tasks left to do. First, the MOP designers must show that the new protocol has the same power as the old one, i.e., that programmers can still accomplish the tasks that caused MOP designers to introduce slot access protocols in the first place. Second, they need to show whether the new protocol really allows slot access to be as fast as slot access without a MOP.

Dynamic Slots Under the Old Protocol

The old **@slotRef** protocol had programmers do this:

```
method @slotRef (@DynamicClass c,
                 obj,
                 slotName) {
  if (@isDynoSlot(c,slotName)) {
    var dvalLoc = c.dvalLoc
    var dvalues = @objectRef(obj, dvalLoc);
    return lookup(@slotName, dvalues);
    }
  else
    return callNextMethod();}
```

As a reminder, here is the slot access method for dynamic slots under the old protocol. It checked whether the slot to access was dynamic. This involved walking up the class hierarchy to see whether the slot was first defined in a dynamic, or in a regular class. If the slot was dynamic, the old **@slotRef** above found the location within the object where the dynamic values list was stored. The value was then retrieved from there. Non-dynamic slot access was deferred to the default method.

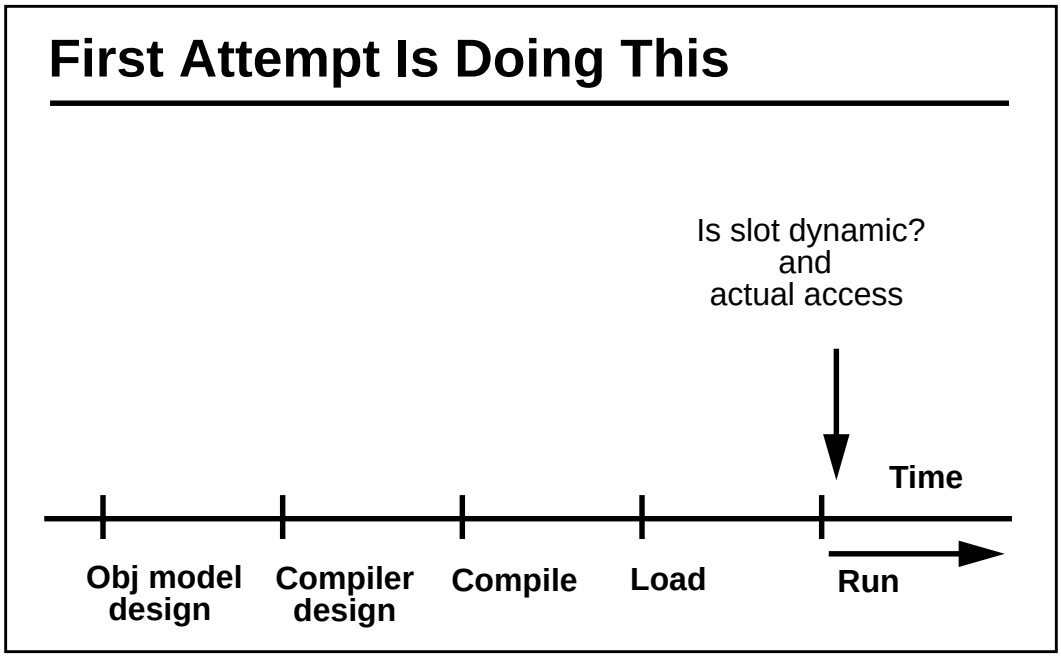
Dynamic Slots For the New Protocol

A First Attempt...

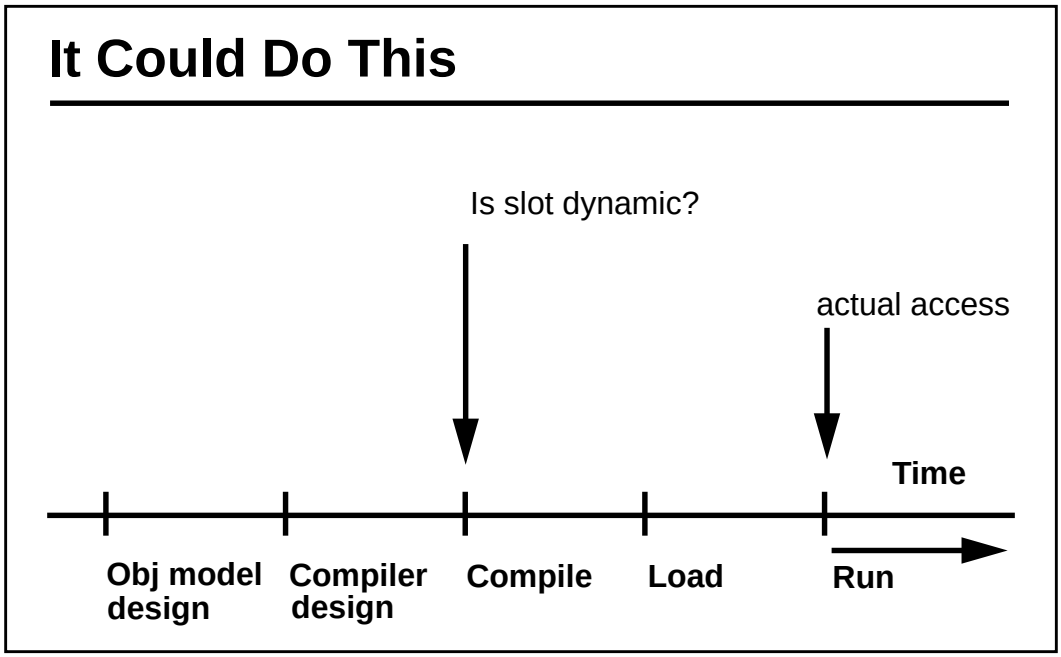
```
method @slotRefResidual (@DynamicClass c,
                         slotName) {
  return
   [ if (@isDynoSlot(c,slotName)) {
       var dvalLoc = c.dvalLoc
       var dvalues = @objectRef(obj, dvalLoc);
       return lookup(@slotName, dvalues);}
     else
       return callNextMethod();];}
```

…only precomputes initial method dispatch.

Here is a programmer taking a first cut at converting this old method to the new protocol. The programmer simply returns the body of the old method as the residual to be executed at runtime. This does make a little progress compared to the old way of doing it, in that it saves the runtime method dispatch to **`@slotRef`**. But if programmers went with this approach, they would miss the point. Most of the optimization opportunities would be lost. Here is the timeline visualization of what would be going on.



Most of the work would still happen at runtime. They would not be taking advantage of the new protocol which provides the opportunity for user-supplied methods to be just as smart as system methods. The timeline picture above should look more like the following.



The check for whether the slot to access is dynamic should happen early.

Dynamic Slots for the New Protocol

Analyzing More Carefully

```
method @slotRefResidual (@DynamicClass c,
                         slotName) {
 return
  [ if (@isDynoSlot(c,slotName)) {
      var dvalLoc = c.dvalLoc
      var dvalues = @objectRef(obj, dvalLoc);
      return lookup(slotName, dvalues);}
    else
      return callNextMethod();];}
```

Residual should be:

```
    lookup("name", @objectRef(obj, #0))
or: @objectRef(obj, #n)
```

A curried protocol provides programmers with the opportunity to think about their code and make it as efficient as it can be. The underlining technique used earlier comes in handy again in finding a better **`@slotRefResidual`**.

We find that the check for whether the slot to access is dynamic can clearly be done earlier than runtime and less often than during every slot access, because both the slot name and the class hierarchy above **`c`** are available at compile- or at least load time (**`@isDynoSlot`**).

The location of the dynamic values in objects of **`c`** can be retrieved from **`c`** before runtime as well (**`c.dvalLoc`**).

After extracting the remaining residuals from the method above, we see that they can be much slimmer than the simple-minded solution. What is left in the case of dynamic slots is the lookup in the slot-name/value pair list within the object or, in the case of standard slots, the normal default residual.

So **`@slotRefResidual`** should be written differently.

Dynamic Slots for the New Protocol

Called at compile time

```
method @slotRefResidual (@DynamicClass c,
                         slotName) {
  if (@isDynoSlot(c, slotName)) {
    var dvalLoc = c.dvalLoc;
    return
     [ lookup(slotName,
              @objectRef(obj, dvalLoc)); ];}
  else
    return callNextMethod();}
```

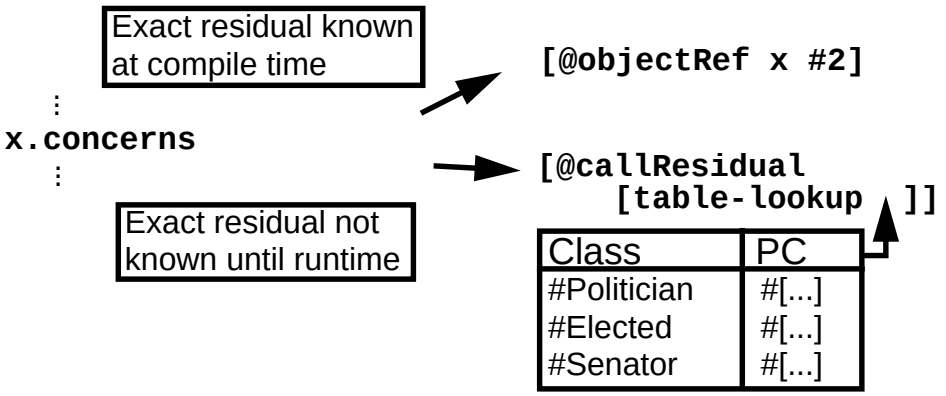
Return regular slot residual

Dynamic Slot Residual:

```
lookup("name",
       @objectRef(obj, #0))
```

Note the square brackets which, in our convention, contain the residual. The dynamic slot test is done within this method which is called at compile time for each slot of each dynamic class.

We Are Almost There



By changing the protocol, our MOP designers have now almost reached their goal of matching the fast slot access in object systems without a MOP (see page 275).

Implementors can now just use the standard table lookup techniques. Only this time what is being looked up are residuals. If the exact residual can be determined at compile time, it is inlined into the code. Otherwise, implementors build a table of possible residuals and inline code that looks up the proper one at runtime.

Remaining Issue: Calling Residuals

Standard Techniques Apply

- Calling a single residual—inlining
- Calling a residual out of a table—some sort of call/return

Or, blend the two using incremental specialization

The only issue left now is how to call residuals quickly.

Packaging and Calling Residuals

Form of Residual	Overhead at Runtime
Procedure	Full function call
Code vector	Fast function call
Symbolic code vector	Inlinable at link time (or later, using Self-like techniques)

The process of calling residuals depends on how implementations represent them. One, cheap, way is to package them as functions. This way the calling mechanism is obvious. But each slot access then costs a full function call. Since the implementation can know and check various aspects of residuals, the full machinery of function calls, with its assorted error checking, is not needed.

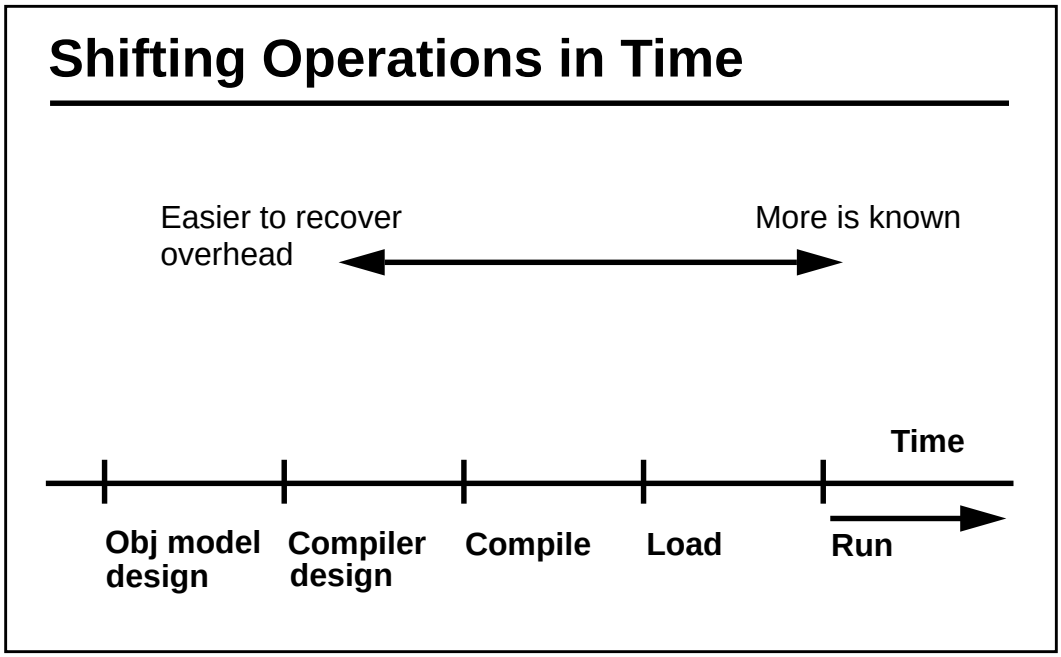
One alternative is therefore the use of light-weight function calls. Or residuals can be passed around not as binary code but in symbolic form, with the implementation performing inlining and incremental compilations.

Review: What Was All of This about?

- Simple protocol places too much overhead in critical runtime paths
- Look for what can be pre-computed—possibly adjusting protocol
- Protocol currying or automatic partial evaluation techniques

In summary, MOP designers had made a mistake with the slot access protocol. Instead of requiring implementations to call **`@slotRef`**, they should have done something a bit more sophisticated. So they changed the protocol to allow programmers to help implementations be efficient, while still maintaining the flexibility programmers need.

The concept of currying a protocol was introduced. It is a technique MOP designers can use whenever they need to enable programmers to analyze their operations, so that those actions that need to be performed only once can be separated from the ones that need to run often. This can allow MOP designers to provide flexibility for programmers, while not making the implementors' job impossible.



For implementors it is most convenient when as much as possible is known early. The earlier a decision is made, the easier it is to implement it efficiently. This is the force that has pushed traditional, static programming languages. On the other hand, the dynamic and lazy slots examples showed that sometimes, better decisions are possible if they are postponed. In both of these cases the slot management strategy was not fixed at compile time as it would in more static languages. For example, if TinyObjects compiler writers had made an irreversible decision that each slot would always occupy a fixed position in objects, overall efficiency would have suffered in the example of **`Person`** objects. Those were much better served by dynamic slots, a fact known much later than the design of the compiler.

Open implementations are about enabling late decisions when they are appropriate, without compromising efficiency normally associated with making those decisions early.

Chapter 8 Summary and Directions

The Problem We Addressed

There are many cases where a system is “almost right” to meet a user’s needs.

- Missing features
- Locally different behavior
- Inconvenient for some frequent action
- Too slow in particular cases

We observed that there are many cases in which programmers want to use a particular substrate and find that it is almost right. There might be some missing features, or it might be inconvenient or inefficient to use for a particular case that happens to occur frequently in the particular application. But basically, the substrate is right.

In those situations programmers often have to compromise, leading to higher application expenses and inefficiencies.

The problem therefore is to avoid forcing programmers into compromising around their use of substrates any more than absolutely necessary.

Our Basic Premise

Address this dilemma by making systems flexible and tailorable, which allows the programmers to adapt the system to their needs (rather than the other way around).

Our approach is to find a way to make substrates flexible enough that programmers can reach in and adjust selected parts of the substrate implementation, without having to involve the substrate designers or implementors.

Another way of looking at this is that we are opting for making systems smaller by including fewer features and considering fewer special cases, deciding instead to make substrates extensible. This is in contrast to languages like Ada which are trying to be complete from the start.

It is a simple intuition, but…

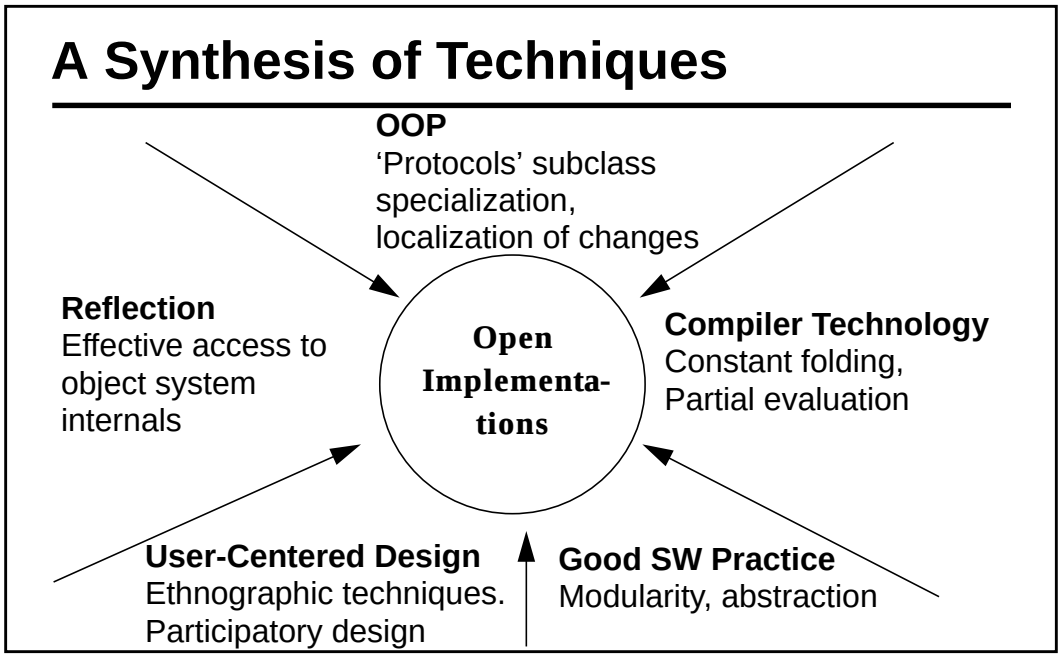
Some Questions That Come Up

- Harder to design
- Leads to bad uses (unreadable code, and so on)
- Harder to implement efficiently

Yes, but…

There are legitimate concerns about the approach of opening implementations. One is that open implementations are harder to design. This is true. Another concern is that open implementations can lead to abuses. Our answer to this is that proper protocol design can be used to control this problem. Specialization rules and checkable result constraints for user-defined extension methods were examples of this.

Another concern is that open implementations are harder to implement efficiently. This is true. But we have shown that a variety of technologies, such as manual partial evaluation techniques, are at this point powerful enough for the challenges posed by open implementations.



In fact, we showed that the open implementation techniques presented were borrowed from, or enriched by a variety of fields, most of which have actively been researched over the years.

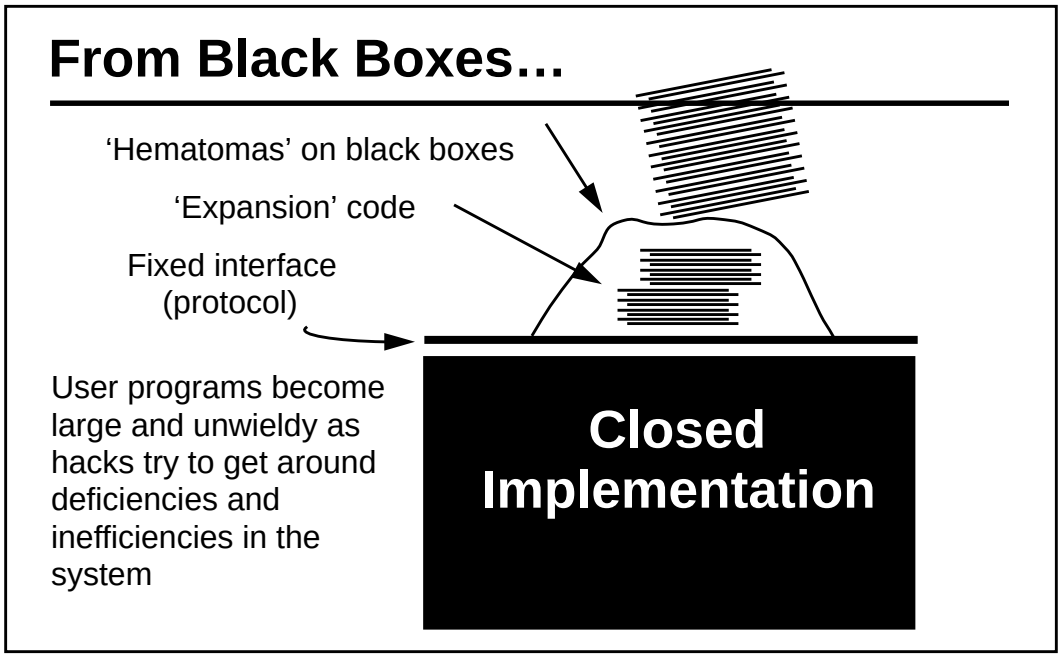
The question is whether the additional work of opening substrate implementations is worthwhile. The answer to this is in part economical. Consider operating systems as an example type of substrate. There are only a handful of commercially significant operating systems in circulation. This compares to thousands of applications written on top of them. Any additional work invested in the operating systems to make application writing easier yields a large payoff.

Success Metrics

- Reduced complexity of user's code
- Reduced size of user's code
- Increased reuse of system by user's code
- Interoperability benefit

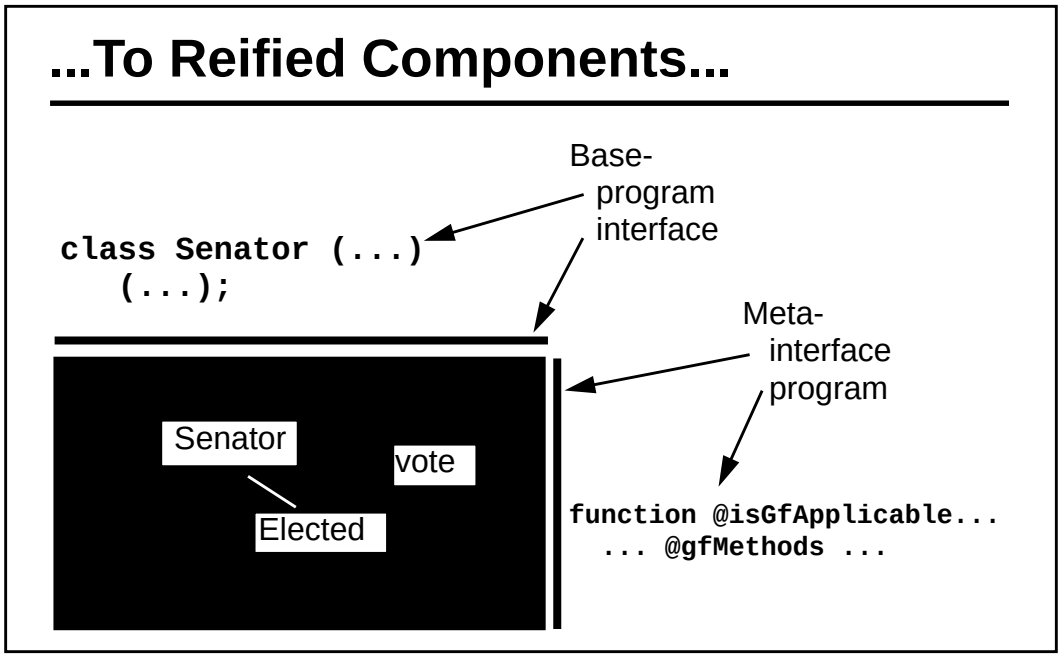
A key point is that success metrics must be user-based. The question is less whether substrates are easier to build, but whether support for programmers writing applications on top of those substrates is improved.

These arguments are strengthened if we look at what happens when substrate implementations are **not** opened up.



The truth is that if programmers really need to make something work, they will find a way. If they cannot do it cleanly because their substrate implementation is closed, then they will code 'hematoma' atop their substrates, bloating the application code and making it less portable. There are cases where programmers add missing substrate features, and others where they replace entire portions of the substrate to gain efficiency for their special needs. This can be observed in the database industry, where operating system functions are often duplicated in the database, because the OS versions of that functionality are too general, and therefore too slow.

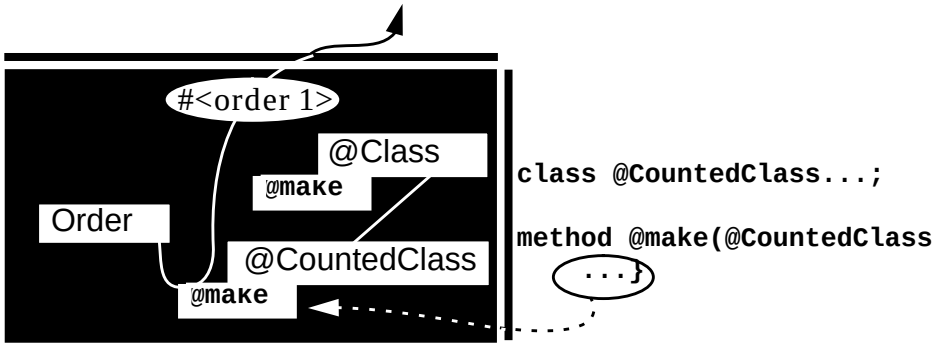
Our dynamic slots example made this clear as well: without the ability to adjust TinyObjects to the special case of sparse object structure, programmers would have had to write their own little language on top of TinyObjects, which provided a new data structure and then dealt with its allocation, inheritance and so on. At a later time they probably would have wanted sparse objects to be known to browsers and inspectors, which would have required more code still. Ensuring efficient compilation would have involved additional efforts. Code hematomas are not a problem to create initially. It is the follow-on effort that is expensive.



The first step towards making these hematoma unnecessary was to enable introspection. This involved the reification of selected components in the TinyObjects system, providing introspective facilities for programs to use and build on. This enabled programs to construct browsers, or to build new functionality, such as tests for method applicability, or the compilation of program statistics.

...And an Effective Interface to Them

```
class Order (...) (...)
  @CountedClass();

new Order()  --> #<order 1>
```



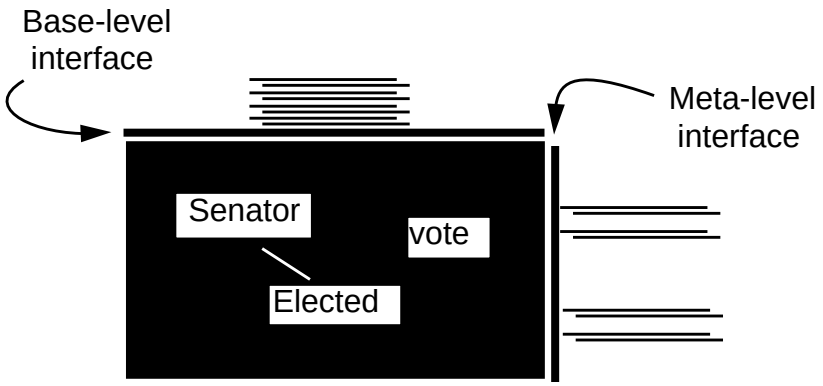
Then we added an effective interface to these reified components. This interface allowed programs to effect operations at runtime which restrictive base-level syntax prevented them from executing before. We called this explicit invocation, and one example was the creation of objects from classes known only at runtime.

This ability was later enriched by the additional capability of injecting state and behavior into TinyObjects implementations, as exemplified by classes that automatically select efficient subclasses to instantiate when programmers provided an expected usage profile for the object to be created.

The use of object-oriented programming allowed us to introduce an important measure of scope control, which limits how far the effect of a modification can be seen. Programmers could, for instance, control which classes retained information about their author, and which did not. More importantly, the classes that did not need authorship information to be retained did not have to pay the price for this capability.

Object-oriented programming also allowed programs operating with a modified object system implementation to co-exist with other programs that used the default implementation. We called this interoperability.

Resulting in Open Implementations



Simple meta-programs tailor the base interface to meet special needs, keeping base programs understandable

All these steps together finally gave us an open implementation of TinyObjects, with a base level interface that programs are written to, and a meta level interface for adjusting selected parts of the TinyObjects implementation.

This separation of concerns allows programmers to control the added complexity of implementation tailorability: it allows for a different working style, where the substrate is adjusted first, so that applications can then be written more easily. We showed how programmers can first write sample code in their "dream object system," which is then implemented by a separate process of writing simple meta level programs.

We showed also that the design of metaobject protocols is an iterative process, fed by programmers attempting to use the evolving MOP to solve their problems.

In the chapter on performance, we showed that implementations can be efficient in the presence of a MOP. A key technique in this context was the currying of protocols, which allowed substrates to avoid unnecessarily repeating computations during time-critical phases at runtime.

To Think about

- Adding a method dispatch MOP to TinyObjects
- MOPifying your favorite language. How much smaller would C++ be? Ada?
- Which programs have you written that would have been simpler if your language had a MOP?
- Do your users want more flexibility in the system you are building?

The purpose of this book is to interest the reader in building their own substrates as open implementations. Here are a few closing questions that may prompt thoughts in this direction. The last question in particular, points away from object systems. The technology we presented for object systems is applicable to other substrates, and research in this direction is ongoing.

Flexibility is an issue for users of databases, operating systems, or distributed environments much the same as for users of object systems. This book provides a framework that can be used to examine application requirements and open implementation opportunities for these other systems as well.

Index Entries - no page references (see)

<$nopage>Class precedence list:accessing,<Emphasis> See<IndexCode> @classCPL;
<$nopage>Inheritance, multiple: <Emphasis>See<Default Para Font> Multiple inheritance;
<$nopage>Multiple inheritance: <Emphasis>See also<Default Para Font> Class precedence list;
<$nopage>Reification:<Emphasis>See also<Default Para Font> Metaobject;
<$nopage>CPL, <Emphasis>See<Default Para Font> Class precedence list;
<$nopage>Methodology.<Emphasis> See<Default Para Font> Design cycle;
<$nopage>Polymorphism:<Emphasis> See<Default Para Font> Class precedence list;
<$nopage>Polymorphism:<Emphasis> See<Default Para Font> Multiple inheritance;
<$nopage>Protocol<Emphasis> See<Default Para Font> Metaobject protocol
<$nopage>Reflection:<Empahsis>See also<Default Para Font> Reification
<$nopage>Class:reification:<Emphasis>See also<Default Para Font> Metaobject
<$nopage>Generic function:reification:<Emphasis>See also<Default Para Font> Metaobject
<$nopage>Method:reification:<Emphasis>See also<Default Para Font> Metaobject
<$nopage>Multiple:<Emphasis>See<Default Para Font> Multiple inheritance
<$nopage>Metaclass:<Emphasis>See also<Default Para Font> Metaobject
<$nopage>Acessing:<Emphasis>See<IndexCode> @classCPL

List of Slides in Presentation Order

Introduction

But I Could Make It Run Better for My Application…

Index