Alphabetical List of Slides

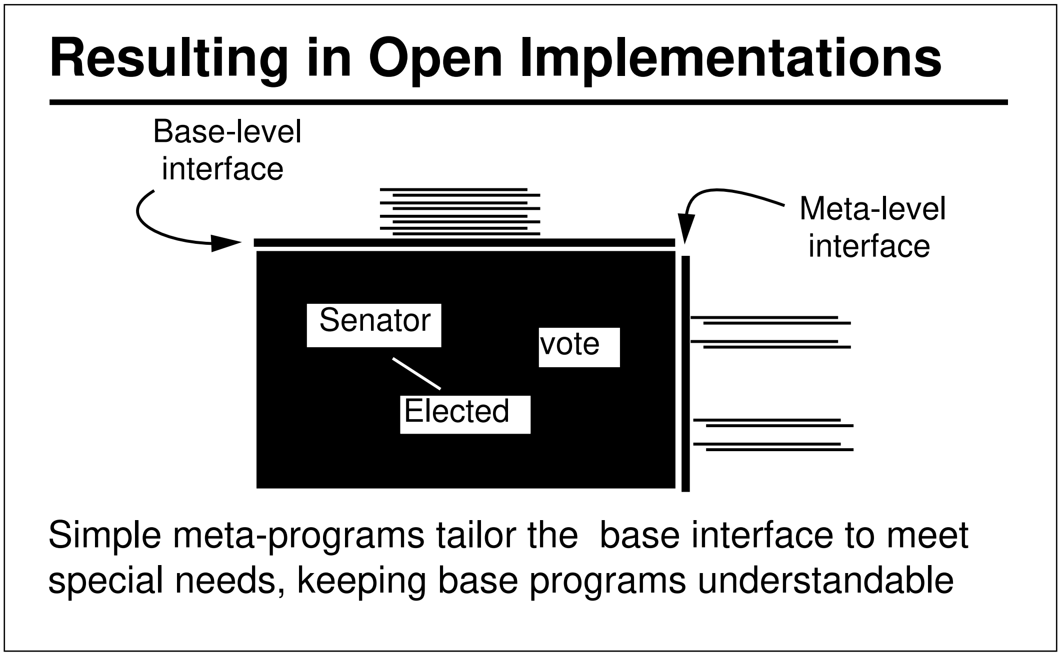
List of Slides in Presentation Order

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.



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.

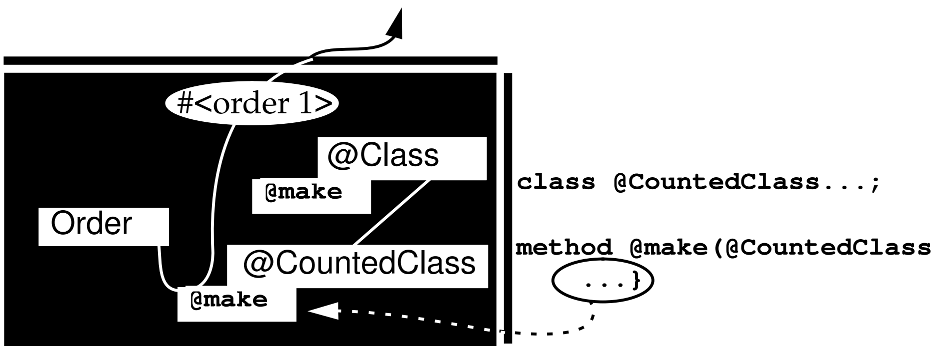
This separation of concerns allows programmers to control the added complexity of implementation tailorability: it allows for a different working style, where the substrate is adjusted first, so that applications can then be written more easily. We showed how programmers can first write sample code in 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.

...and an Effective Interface to Them

```
class Order (...) (...)
  @CountedClass();

new Order()  --> #<order 1>
```



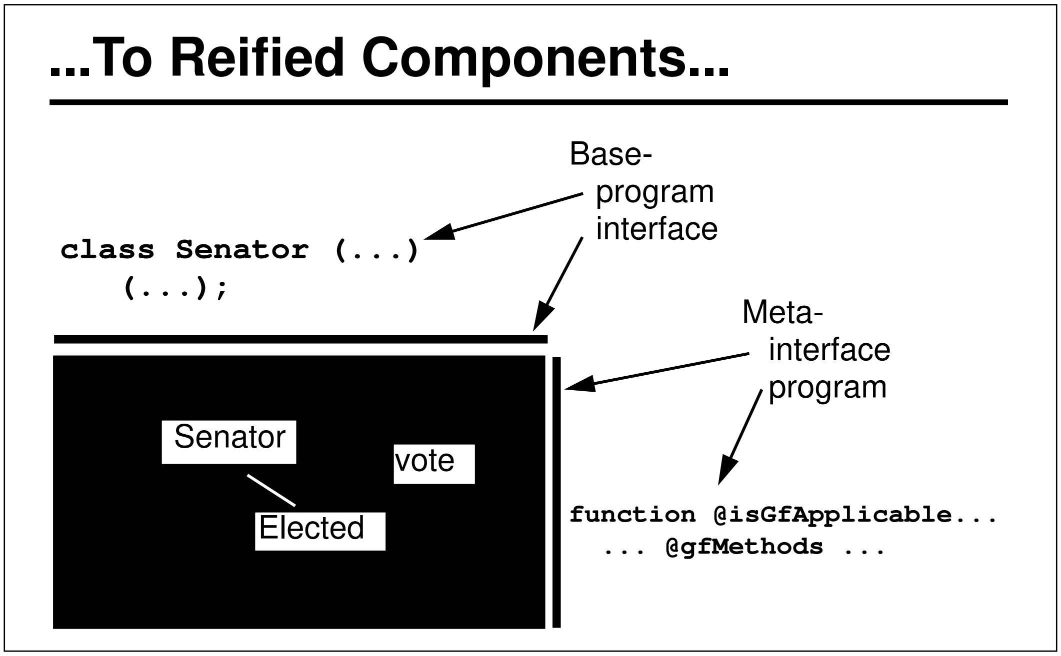
Then we added an effective interface to these reified components. This interface allowed programs to effect operations at runtime which restrictive base-level syntax prevented them from executing before. We called this explicit invocation, and one example was the creation of objects from classes known 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.

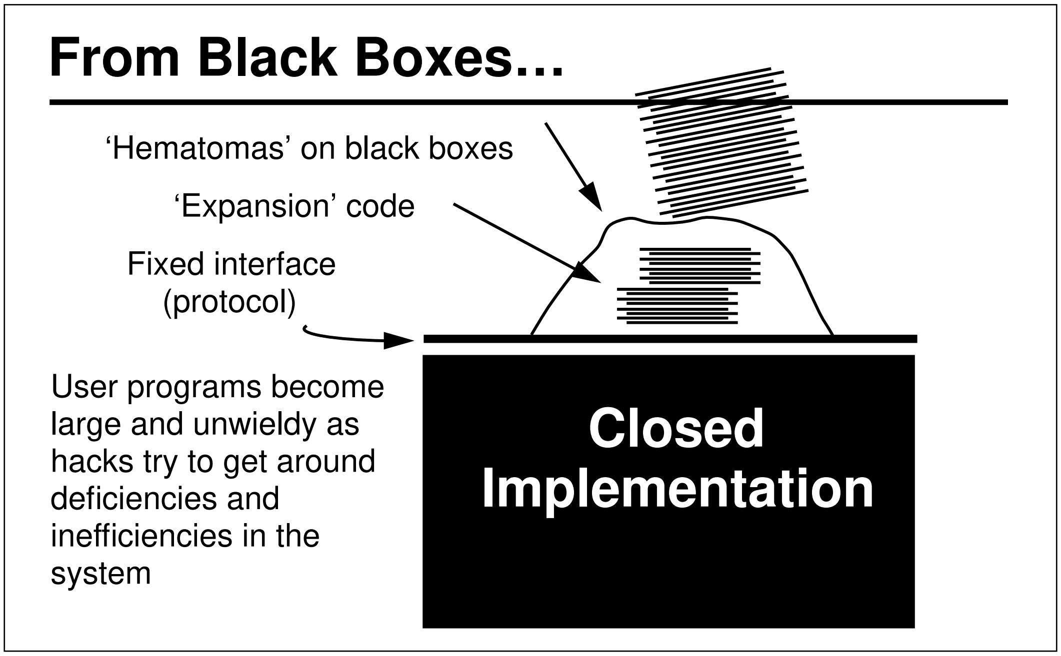
The use of object-oriented programming allowed us to introduce an important measure of scope control, which limits how far the effect of a modification can be seen. Programmers could, for instance, control which classes retained information about their author, and which did not. More importantly, the 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.

Conclusion.frm



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.



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.

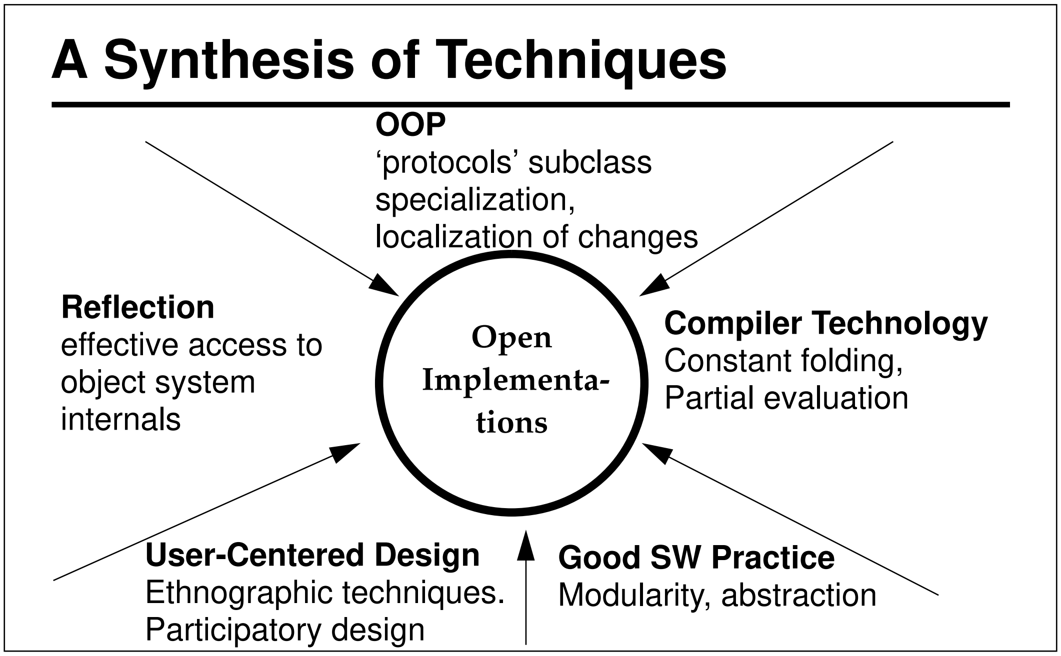
Our dynamic slots example made this clear as well: without the ability to adjust TinyObjects to the special case of sparse object structure, programmers would have had to write their own little language on top of TinyObjects, which provided a new data structure and then dealt with its allocation, inheritance 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.

Success Metrics

- Reduced complexity of user's code
- Reduced size of user's code
- Increased reuse of system by user's code
- Interoperability benefit

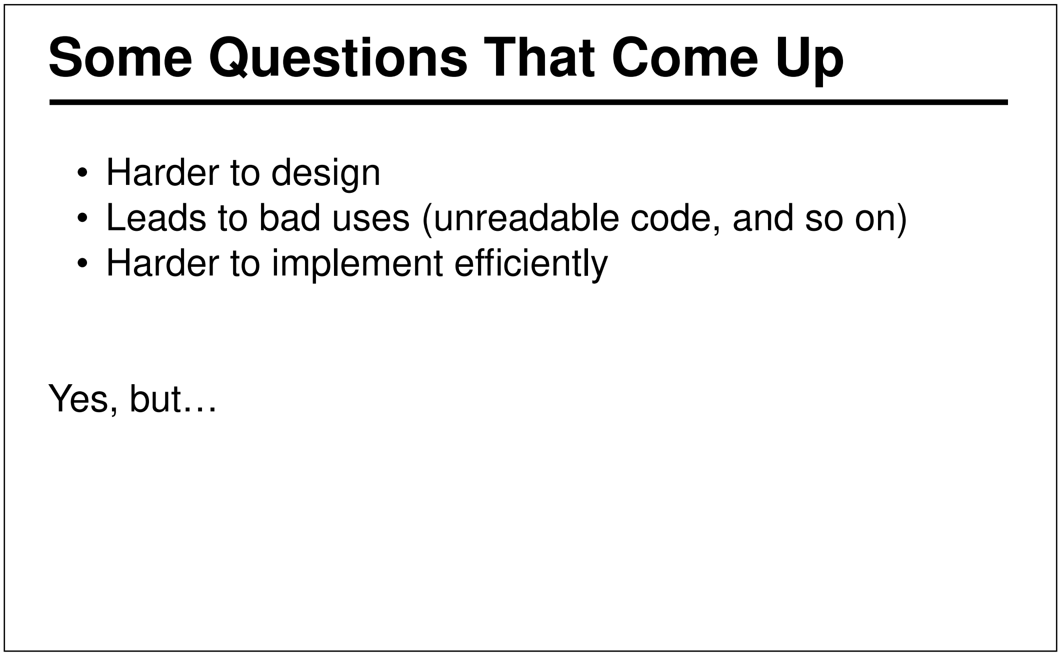
A key point is that success metrics must be user-based. The question is less whether substrates are easier to build, but whether support for programmers writing applications on top of those substrates is improved.

These arguments are strengthened if we look at what happens when substrate implementations are **not** opened up.



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.



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.

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…

The Problem We Addressed

There are many cases where a system is “almost right” to meet a user’s needs.

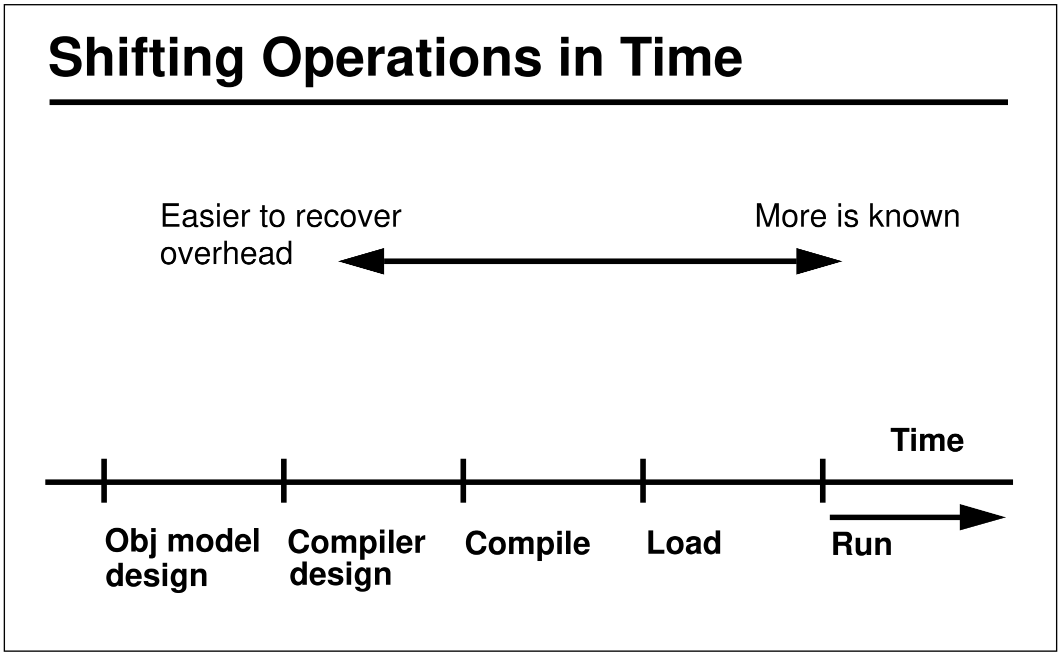
- Missing features
- Locally different behavior
- Inconvenient for some frequent action
- Too slow in particular cases

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.

Chapter 8 Summary and Directions



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.

Review: What Was All 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.

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.

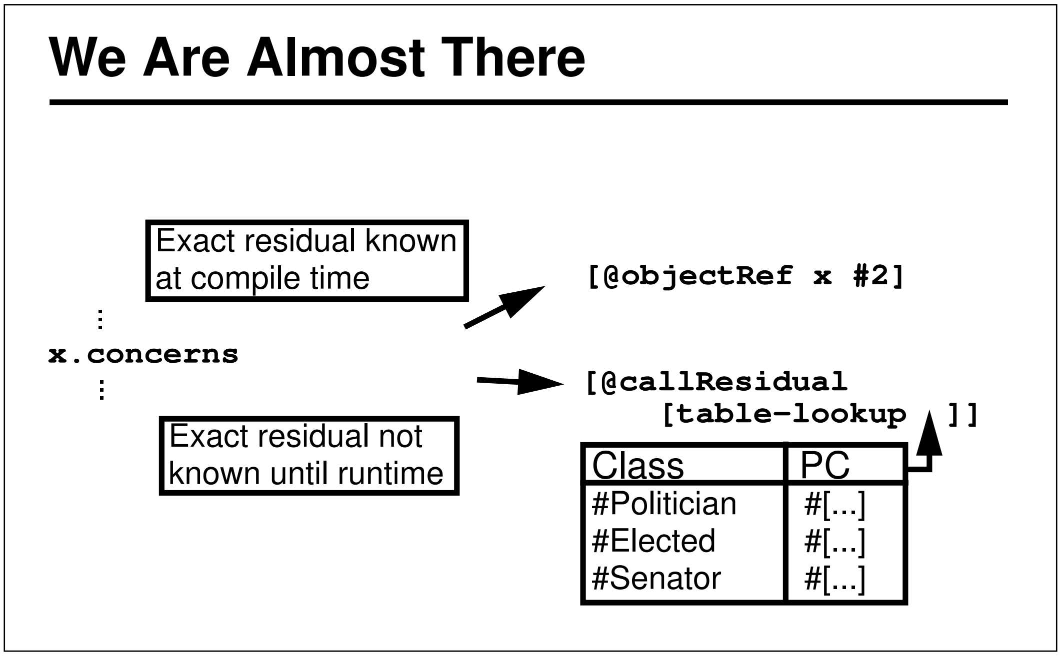
Remaining Issue: Calling Residuals

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.



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 280).

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.

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.

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)
```

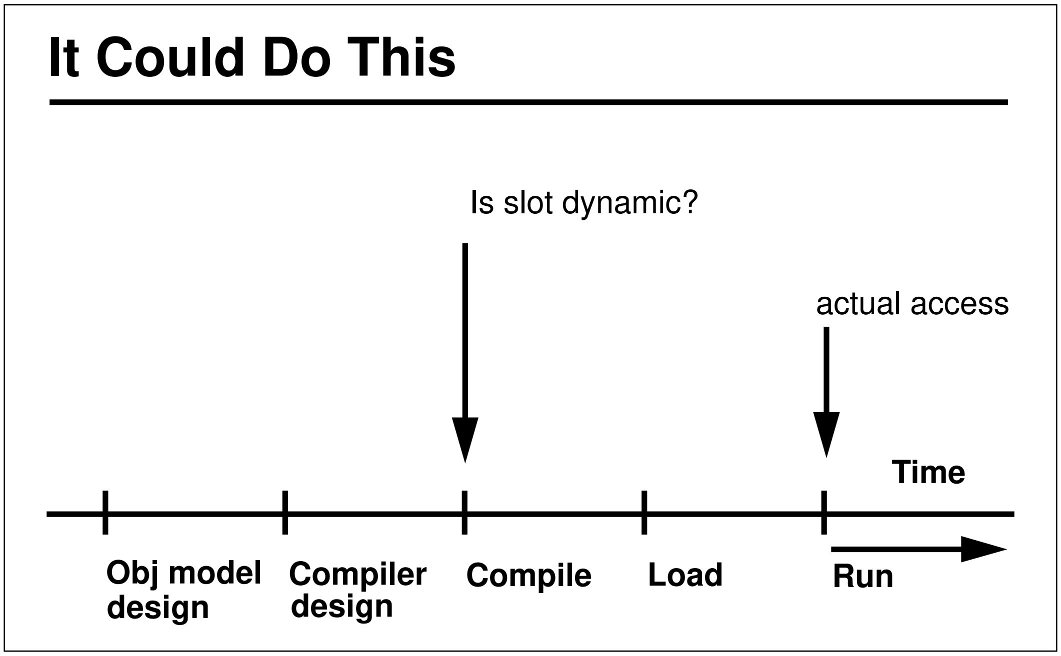
A curried protocol provides programmers with the opportunity to think about their code and make it as efficient as it can be. The underlining technique used earlier comes in handy again in finding a better **`@slotRefResidual`**.

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`**).

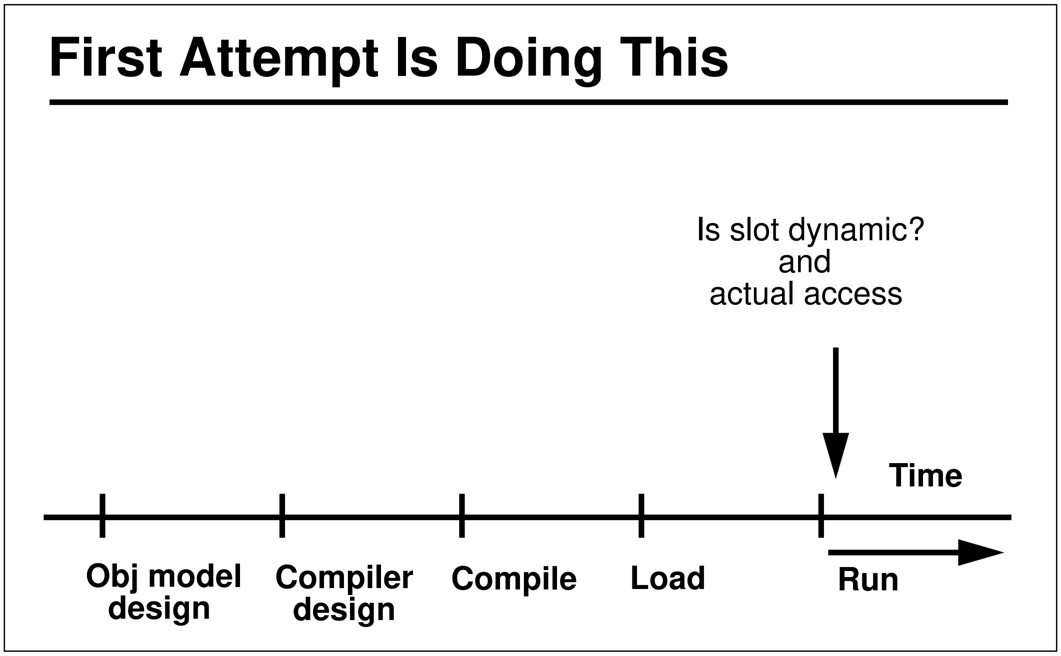
The location of the dynamic values in objects of **`c`** can be retrieved from **`c`** before runtime as well (**`c.dvalLoc`**).

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.



The check for whether the slot to access is dynamic should happen early.



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.

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.

Dynamic Slots Under the Old Protocol

The old **`@slotRef`** protocol had programmers do this:

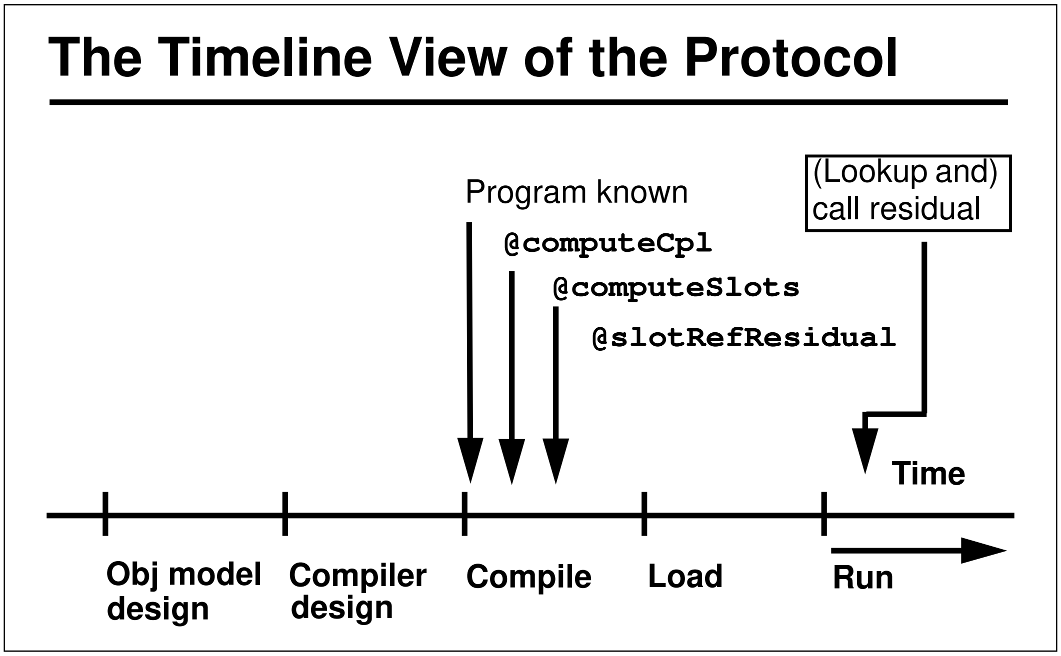
```
method @slotRef (@DynamicClass c,
                 obj,
                 slotName) {
  if (@isDynoSlot(c,slotName)) {
    var dvalLoc = c.dvalLoc
    var dvalues = @objectRef(obj, dvalLoc);
    return lookup(@slotName, dvalues);
    }
  else
    return callNextMethod();}
```

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.

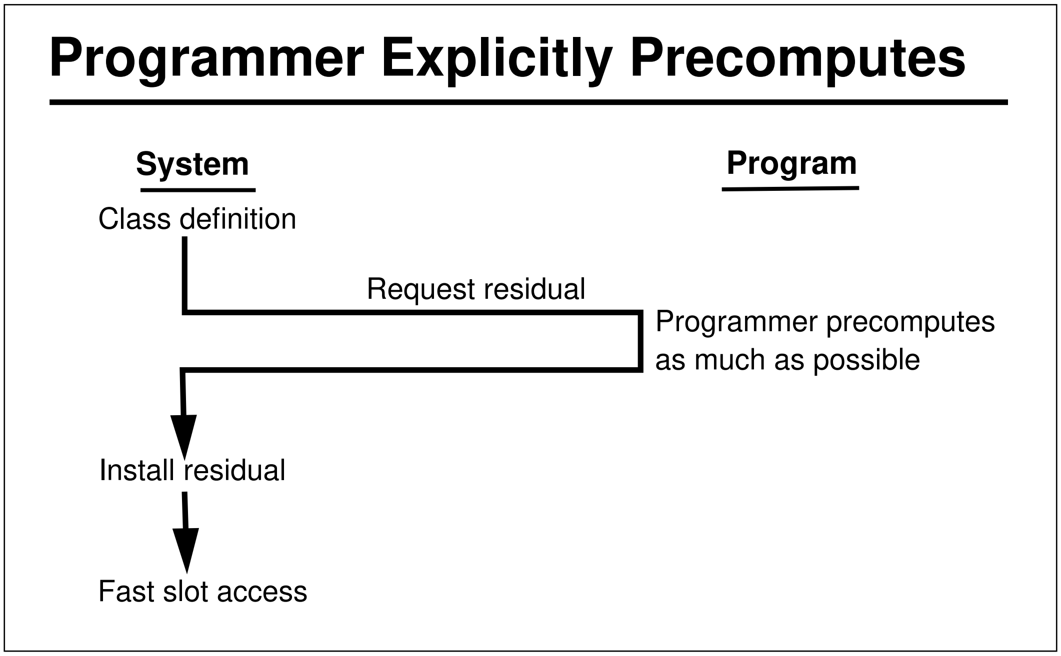
Dynamic Slots Need to Be Reworked

- 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.



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.



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...

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.

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 l**oc** variable within it will have been replaced with the actual location.

So, let us have our MOP designers document this new protocol.

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 TinyObject system method would look like under this new protocol.

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 analyis 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.

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 258 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's 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.

Manually Pre-Computing Slot Access

If class is known to be **Senator**
so class of class is **@Class**

```
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);}
```

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.

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.

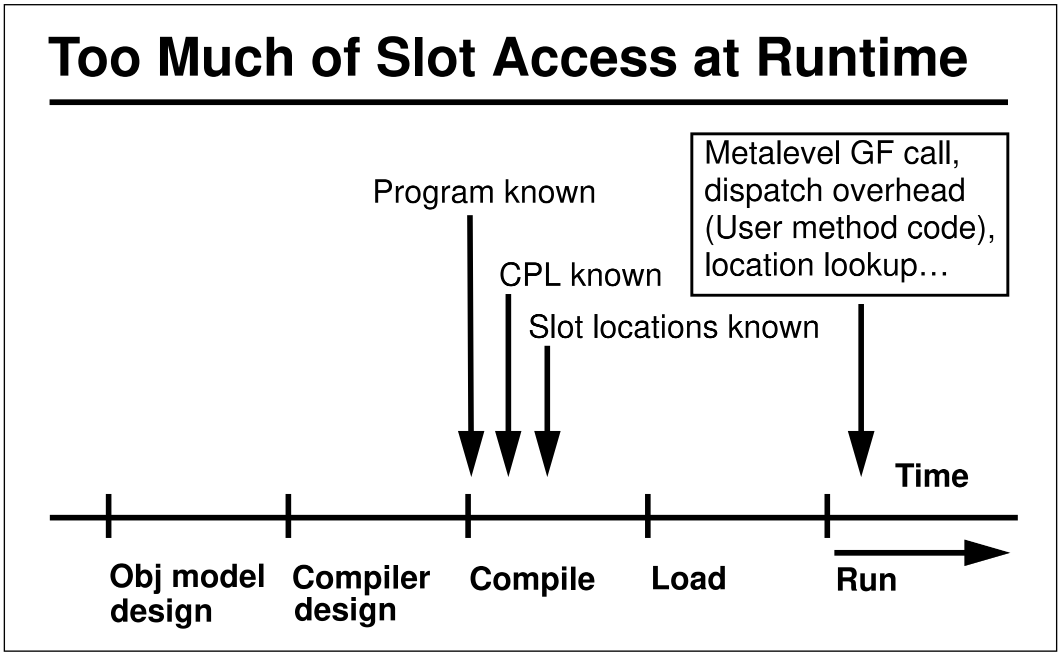
A Closer Look at Slot Access

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 280.



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.

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**'s **@slotRef** is complete, the—potentially lengthy—**@isDynoSlot** runs. If this test comes out negative, the **callNextMethod** produces additional dispatching costs.

HUGE Overheads

```
   ⋮                    @slotRef(...)
x.slotName    →         method dispatch
   ⋮                    if  some-test
                            special-case
                            callNextMethod()
                        more dispatch
                        @objectRef(...)
```

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.

We Committed to Huge Overheads!

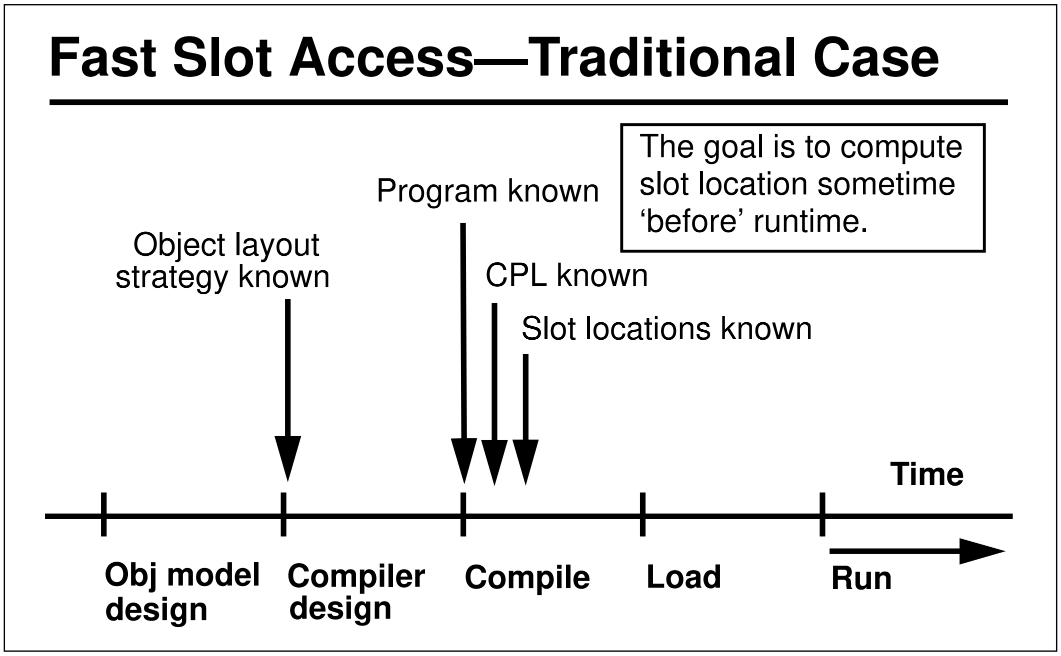
The specter of `@slotRef`

Recall this spec:

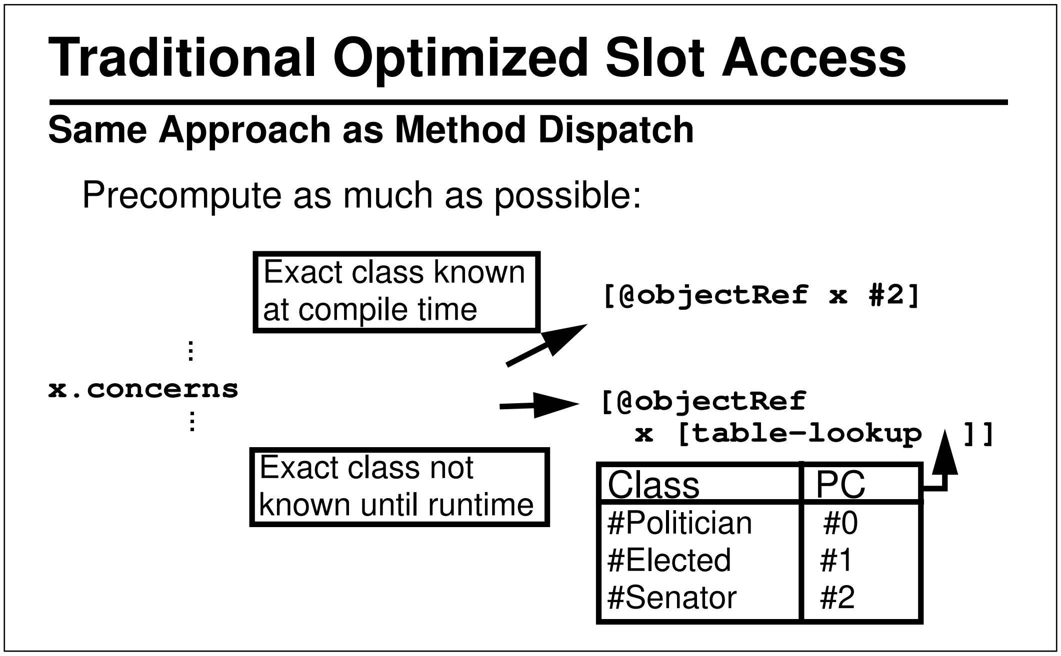
> **`@slotRef`** is the generic function that is called to implement slot access. It receives as arguments the class of object, the object and the slot name.

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.



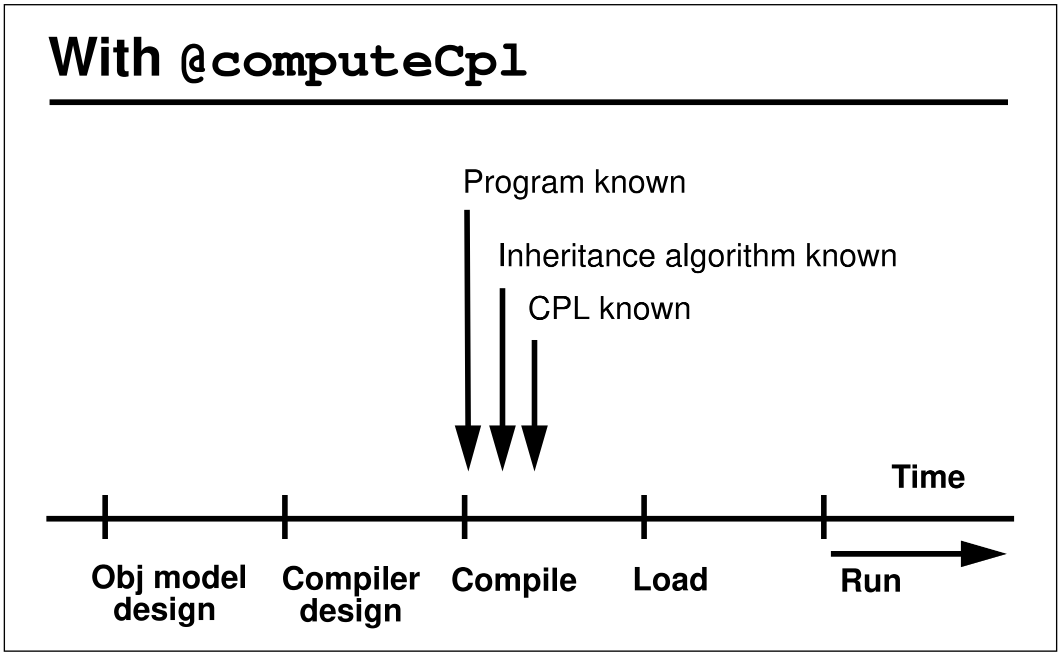
This works because in traditional object implementations the strategy for managing storage for objects is known at compiler design time. Once the program and the CPLs are known, slot locations will therefore be known as well, and a class uniquely determines where in one of its objects a given slot can be found.



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.



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 193).

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 275. 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 193 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.

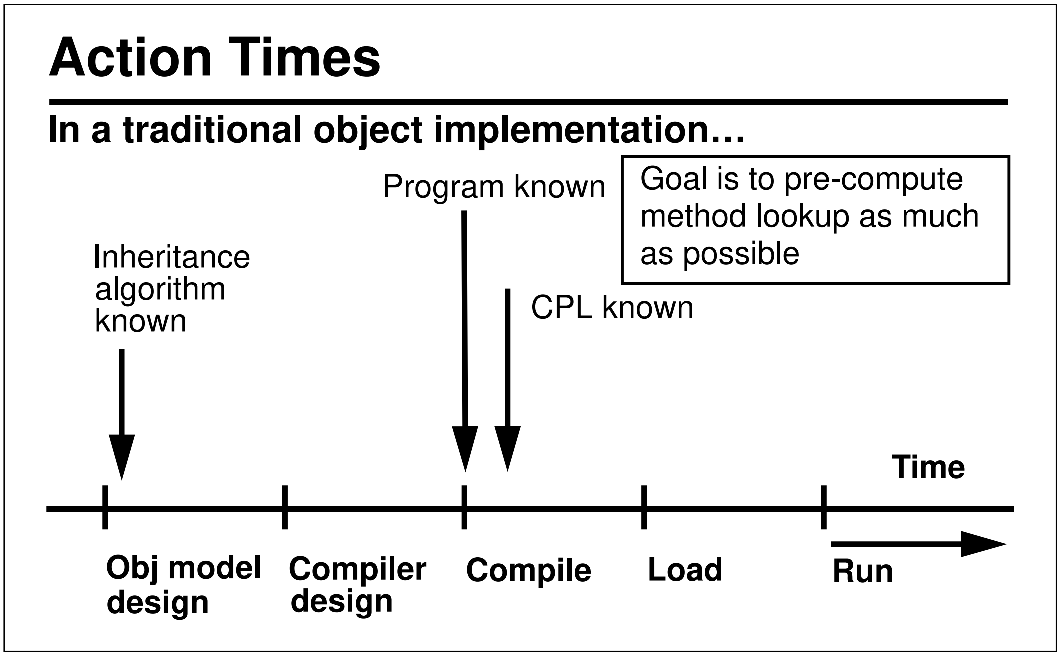
But What About `@computeCpl`?

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 275.

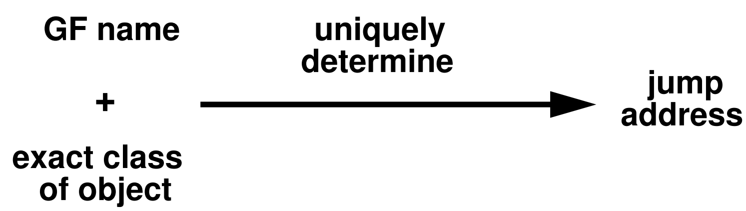
The answer is "no" because the timeline that takes **`@computeCpl`** into account is almost exactly the same as before...



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's 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.

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.

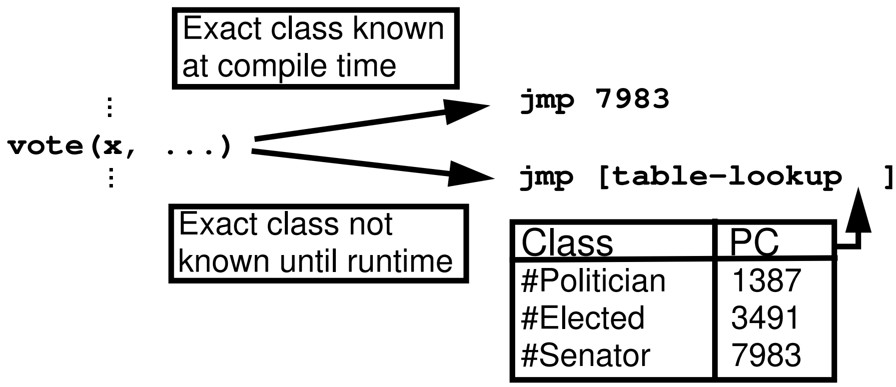


Why are compiler writers allowed to take this shortcut around method dispatch? They may do it because the combination of generic function name and the class of the argument uniquely determine which method to run. In other words, method dispatch does not depend on the time of day or the phase of the moon, but just on these two quantities.

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 techniqes 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.

Performance

Critical parts of TinyObjects

Method dispatch:

```
    ⋮
vote(x, ...)        ➡  compiles to ???
    ⋮
```

Slot access:

```
    ⋮
x.concerns          ➡  compiles to ???
    ⋮
```

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.

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.

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
- ☑ • Performance adjustments

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.

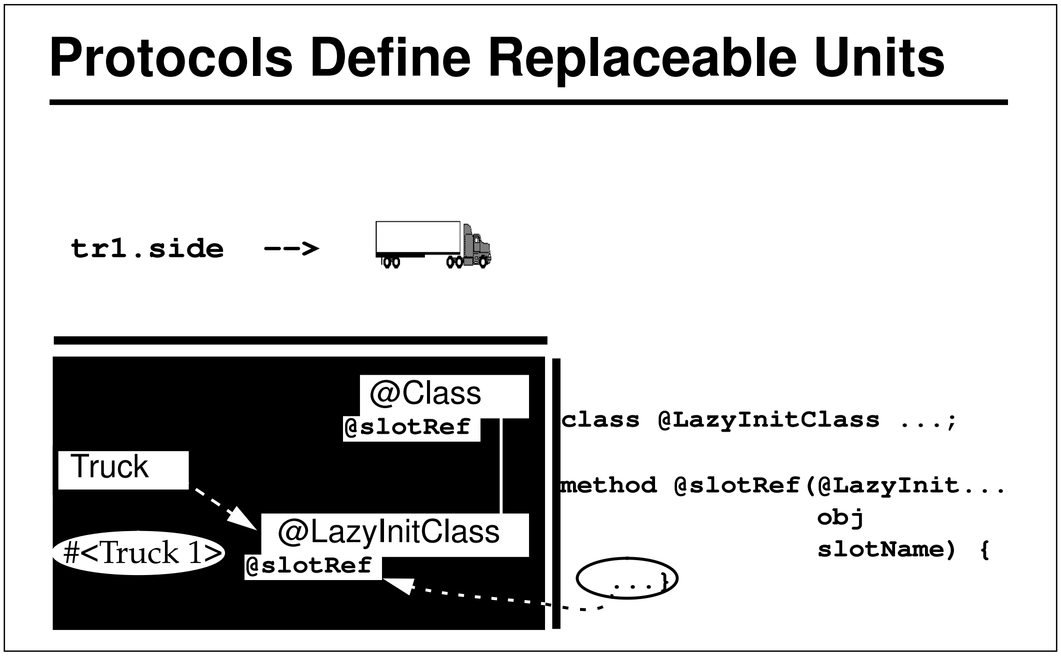
The following chapter will therefore take a good look at this central aspect: how can MOPs be designed to allow efficient implementations?

Intercession Rules

Reliance on basic SE principles

- Rules ensure that modules fit together
- They guarantee independence from implementation details

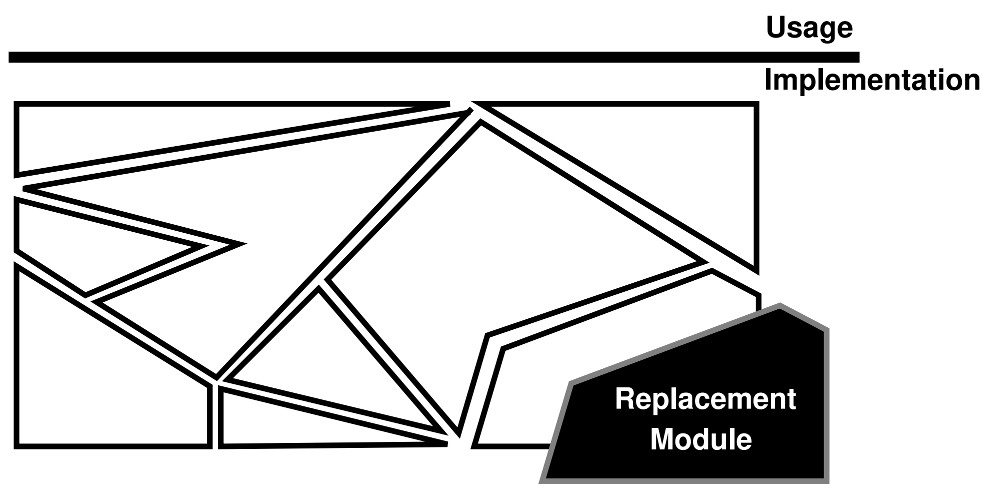
The rules associated with interecession ensure that the substrate modules will still fit together, after some of them have been modified or replaced.



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 is Software Engineering



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.

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.

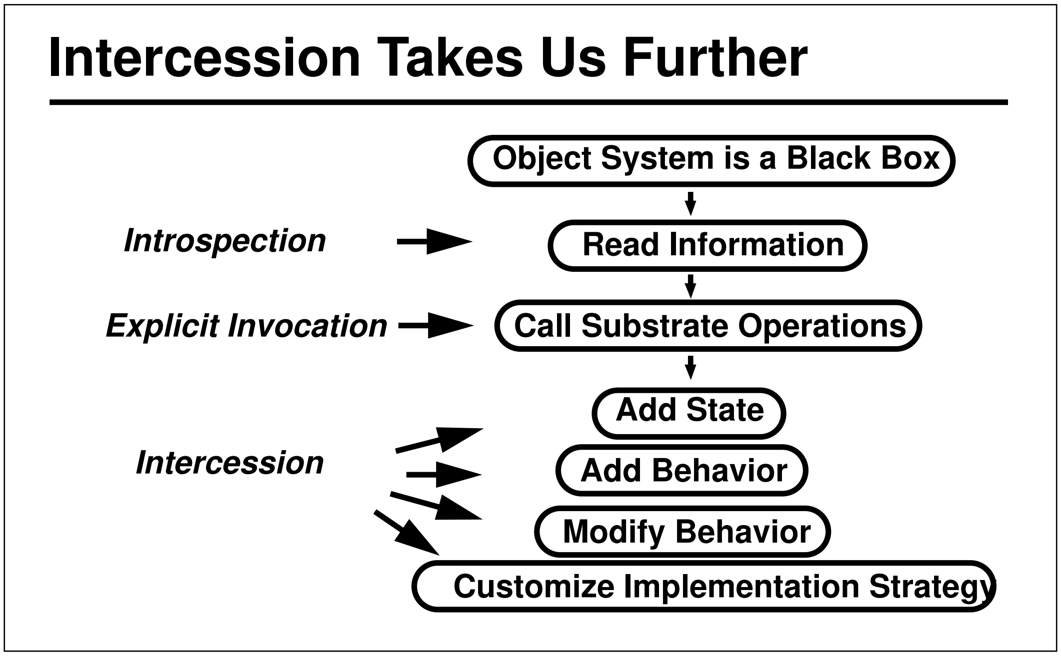
What About Our Rules?

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 221.



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.

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 264 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.

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.

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 236: 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.

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.

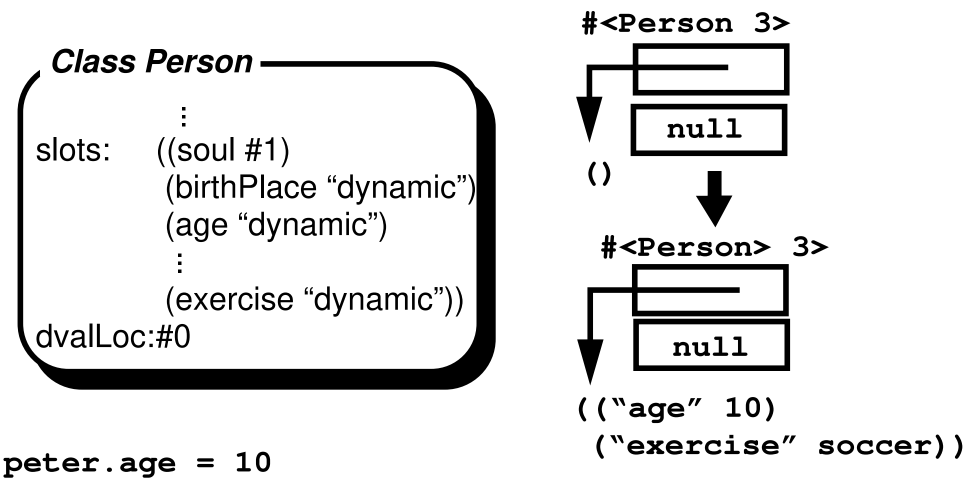
A different design is possible. This predicate could even be a generic function itself.

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"`**.

Marking the Dynamic Slots




```
peter.age = 10
peter.exercise = “soccer”
```

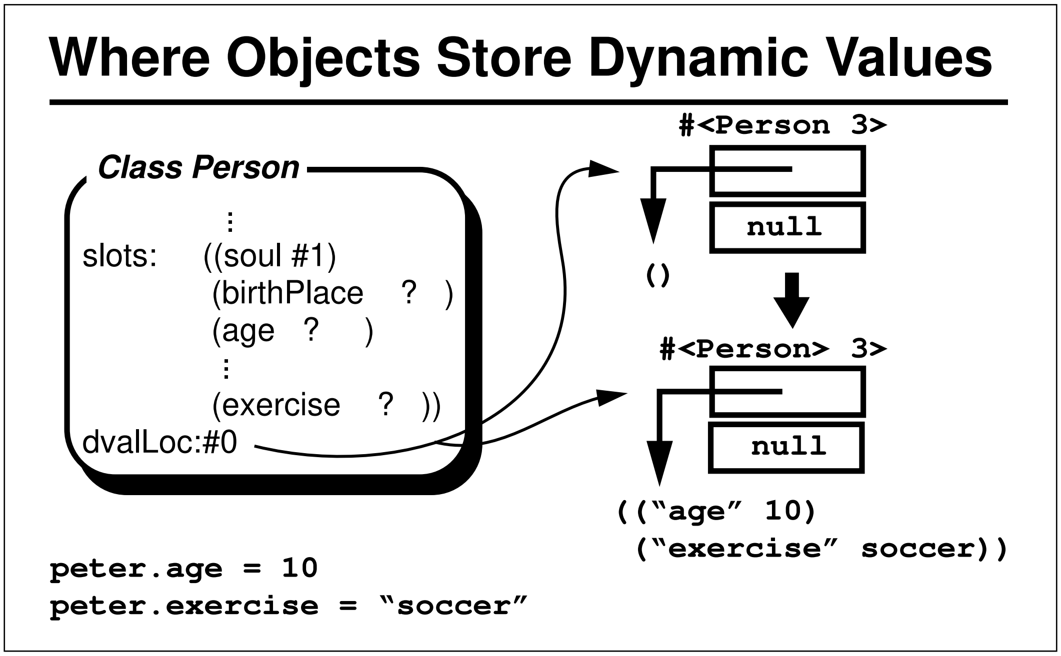
Since dynamic slots have no normal locations, they need to have a non-standard appearance in the storage map so that they can be recognized when slots are accessed at runtime. Instead of a storage location, the value part of entries for dynamic slots in the class will therefore contain the string **`“dynamic”`**.

Building the Storage Map (1/2)

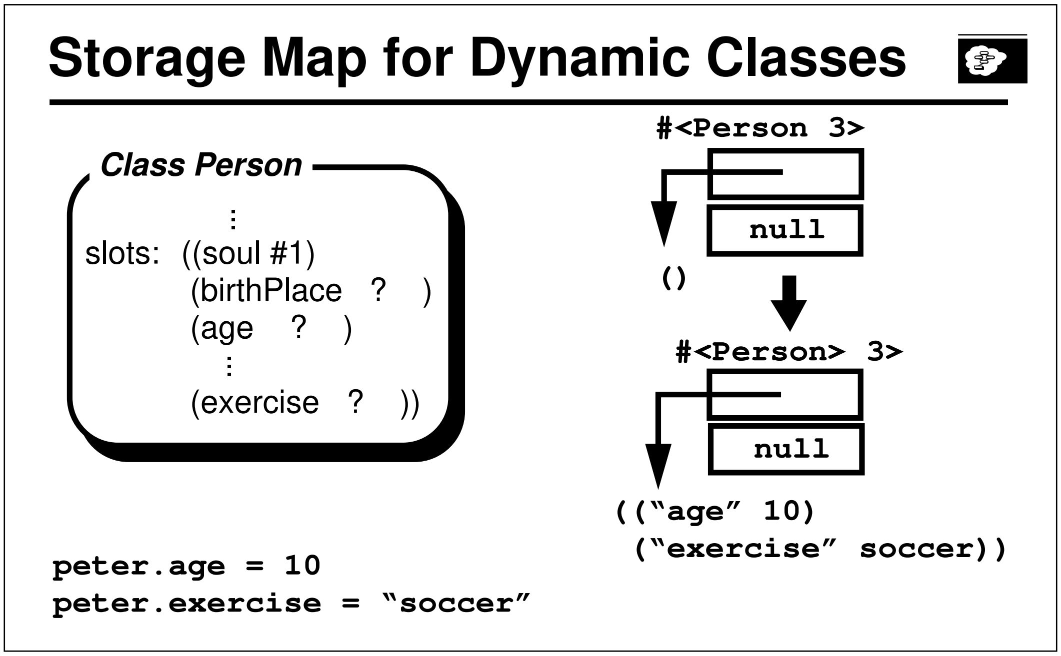
```
class @DynamicClass (@Class)
    (dvalLoc);

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.



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.



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.

How to Prevent Storage Allocation

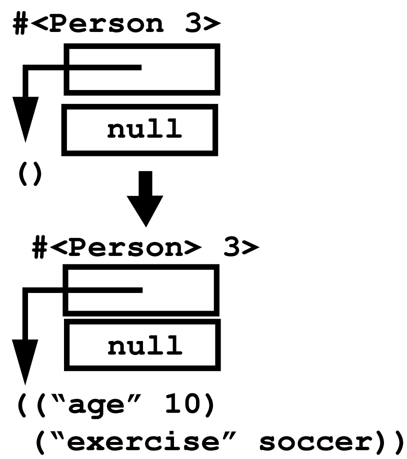
There is a need to…

- 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 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.

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.

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.

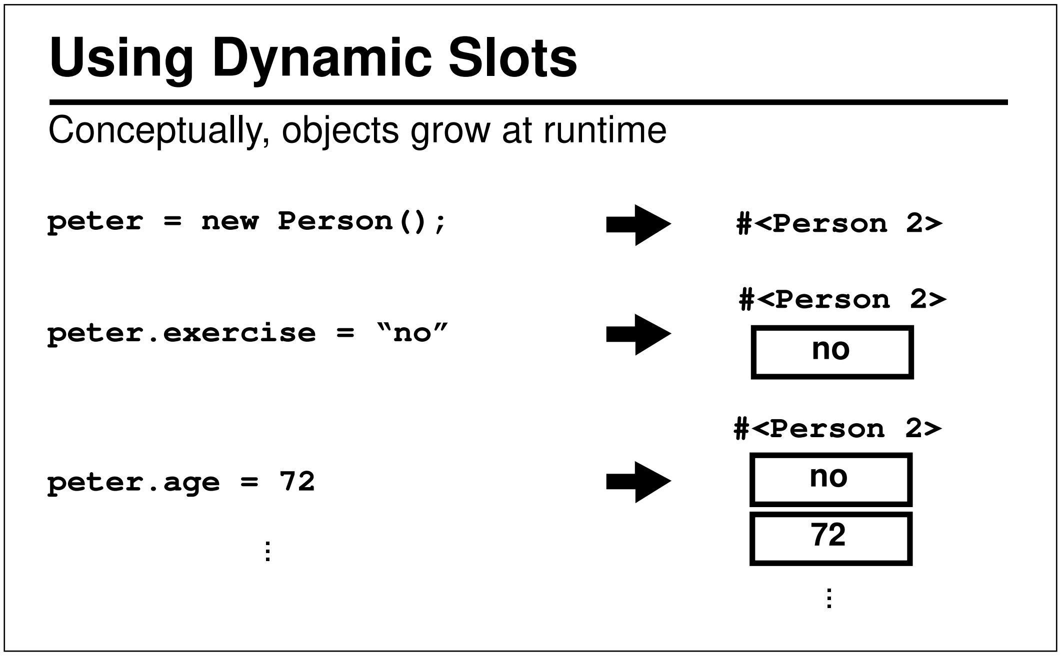
On the first point, the programmer decides for the simplest solution: All the slots defined on a dynamic class will automatically be dynamic. More fine-grained solutions are, of course, imaginable. One could, for example, introduce options at the slot level, which would allow individual slots to be marked as dynamic.

Here is how dynamic slots could look to the programmer.

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.



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 run-time.

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***

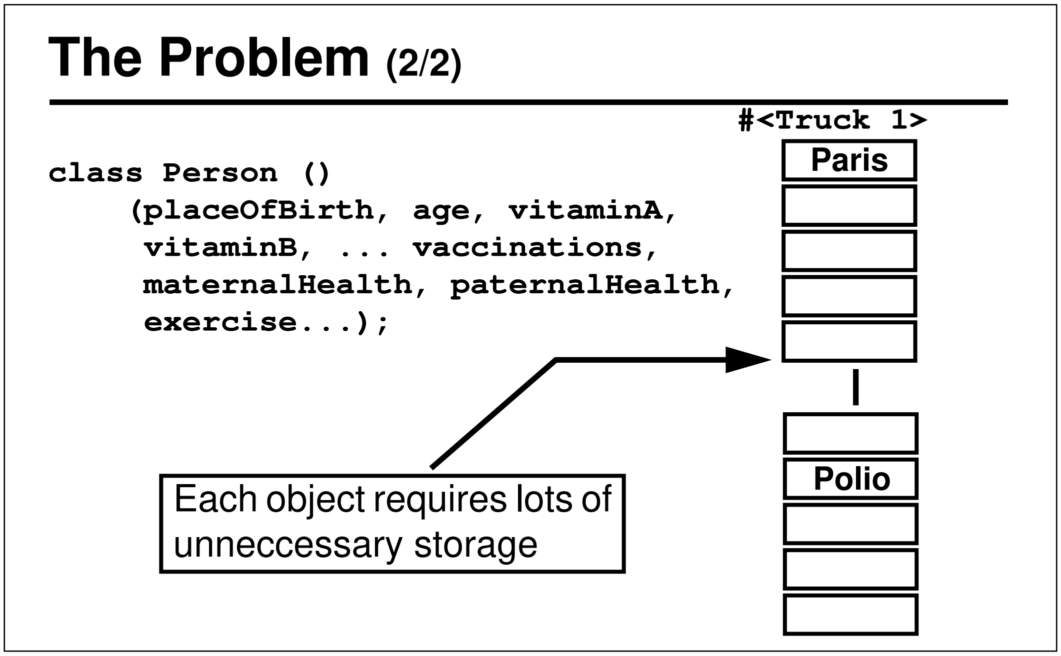
Ideally, objects would be created without any storage. Room for slot values would be allocated on demand. From the outside, this behavior would be transparent: the objects would behave the same way as regular objects.

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.

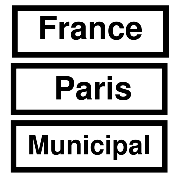


The problem lies in the classes with many slots, only few of which are used in any given object. This might be true because the information is not available, or because the objects represent sparse matrices, or for any number of other reasons. The point is that each object is very large, because the entire set of—mostly empty—slots is allocated each time. This is wasteful.

The Problem (1/2)

```
class Place ()
    (country,
     city,
     hospital);
```

#<Place 1>



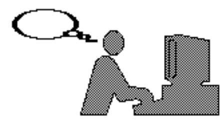
For each slot, the default implementation allocates one location in the object.

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.

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?

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.

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. Intercessionary 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 240, for instance, does not expose how object storage is allocated and where. This remains for each implementation to decide.

Accessing Attributes

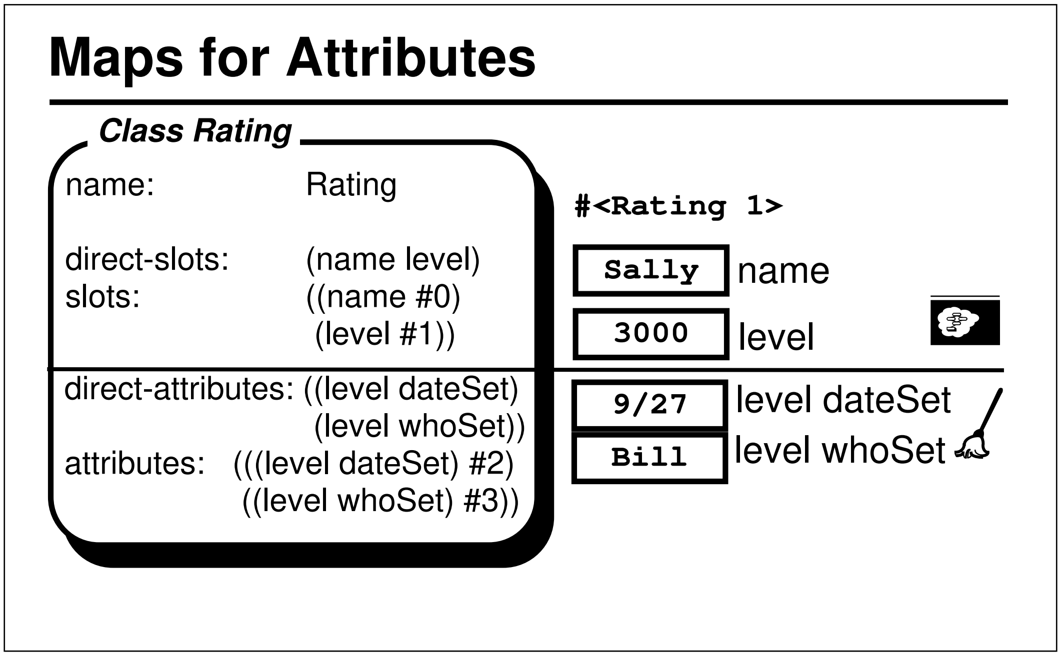
```
function attributeRef (obj, slot, attribute) {
  return @attributeRef(@classOf(obj),
                       obj,
                       slot,
                       attribute);}
method @attributeRef (@AttributesClass c,
                      obj, slot, attribute) {
  var loc = lookup(list(slot,attribute),
                   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.

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.



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.

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.

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.

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.

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.

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 235 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 locatoins 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.

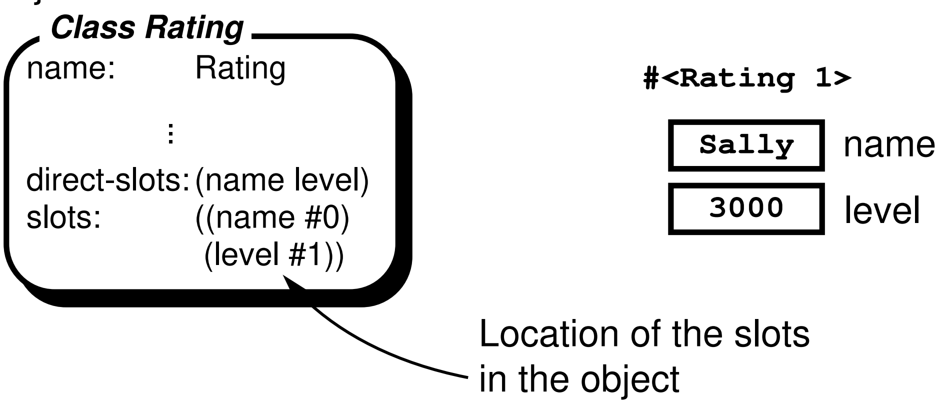
`@slotRef` and **`@slotSet`** are built on top of more primitive functions which take a location specification and access the respective slot in a given object.

Each of these will be examined now.

Intercession2.frm

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.

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.

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 violatated because, even though the programmer is specializing on a default (built-in) metalevel class, he is just *adding* a generic function, not overriding one.

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?

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 228.

Remember Class Initialization

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

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

direct properties

```
  updateDirectSupers(directSupers, c);
```

backlinks

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

derived 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…

Keeping Information About Attributes

Intercession again

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

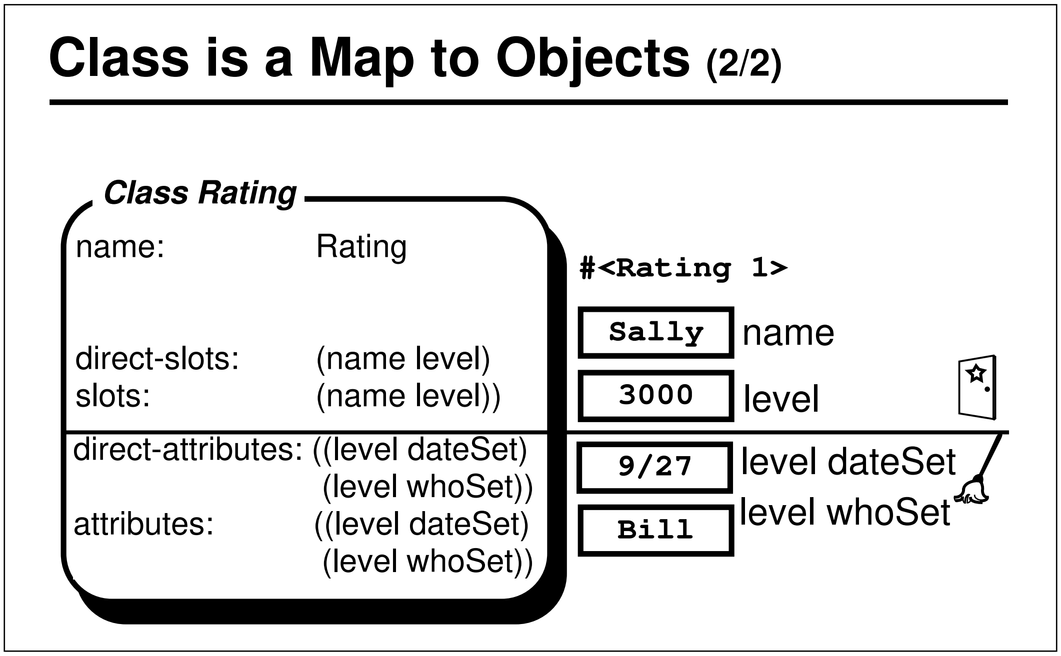
like direct-slots

like slots

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.



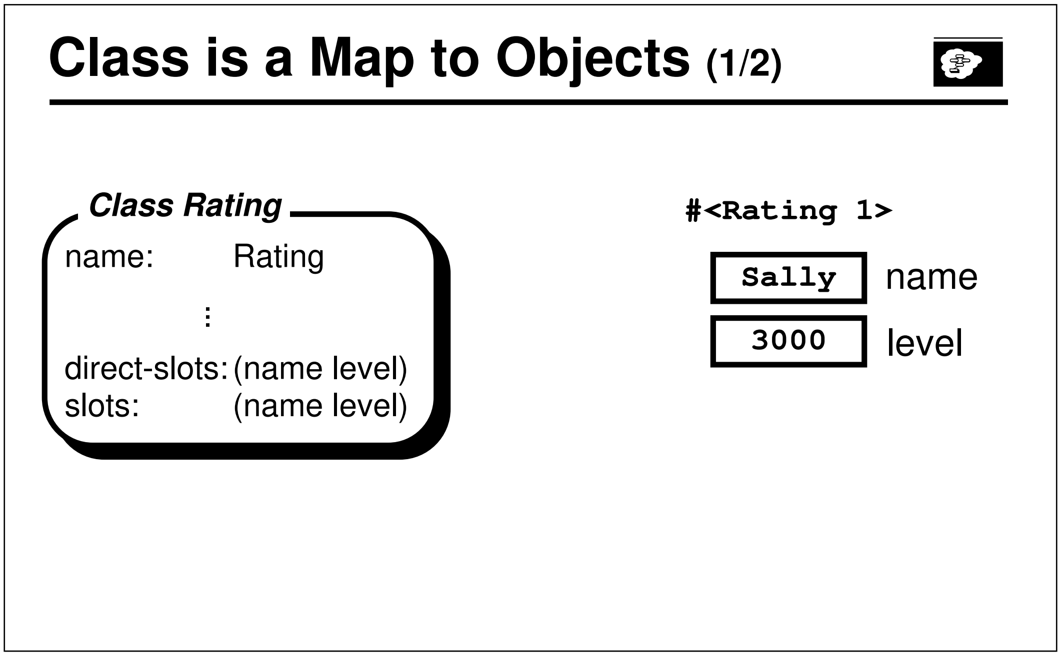
One way could be to just generalize from how slots are handled and to give the programmer appropriate abstractions into the mapping functionality.

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.



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?

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 intercessionary 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.

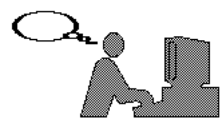
This Programmer Wants to Write

```
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.

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 others' 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.

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.

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 214.

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, **@lazySlotInitValue** 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.

Intercession2.frm

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 metaclasses are more appropriate for modifying slot access behavior. A second reason for using the metalevel 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.

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}
```

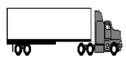
More about this shortly

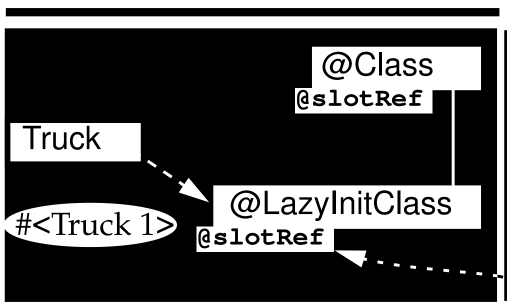
Before really documenting the new proposed piece of protocol, the MOP designer needs to examine whether it can be implemented in such a way that old code will not break, and whether code taking advantage of the new protocol is *interoperable* with code using the default protocol: can they exist in the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

Making `@slotRef` Extensible (2/2)

tr1.side --> 

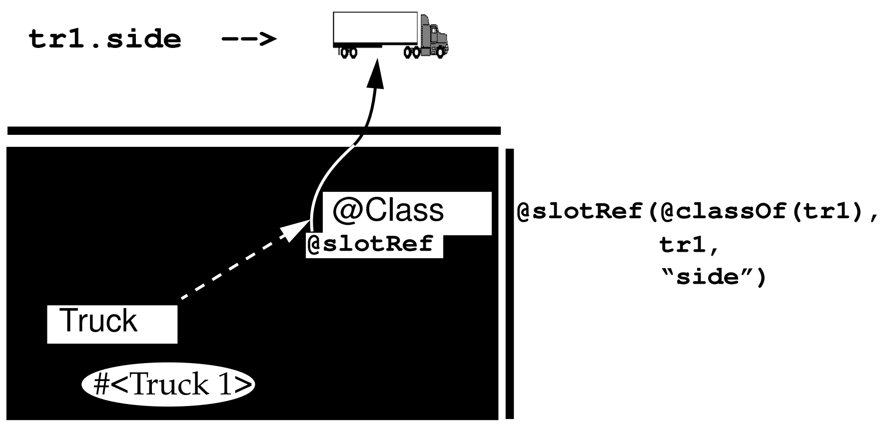



```
class @LazyInitClass ...;

method @slotRef(@LazyInit...
                 obj
                 slotName) {
  ...}
```

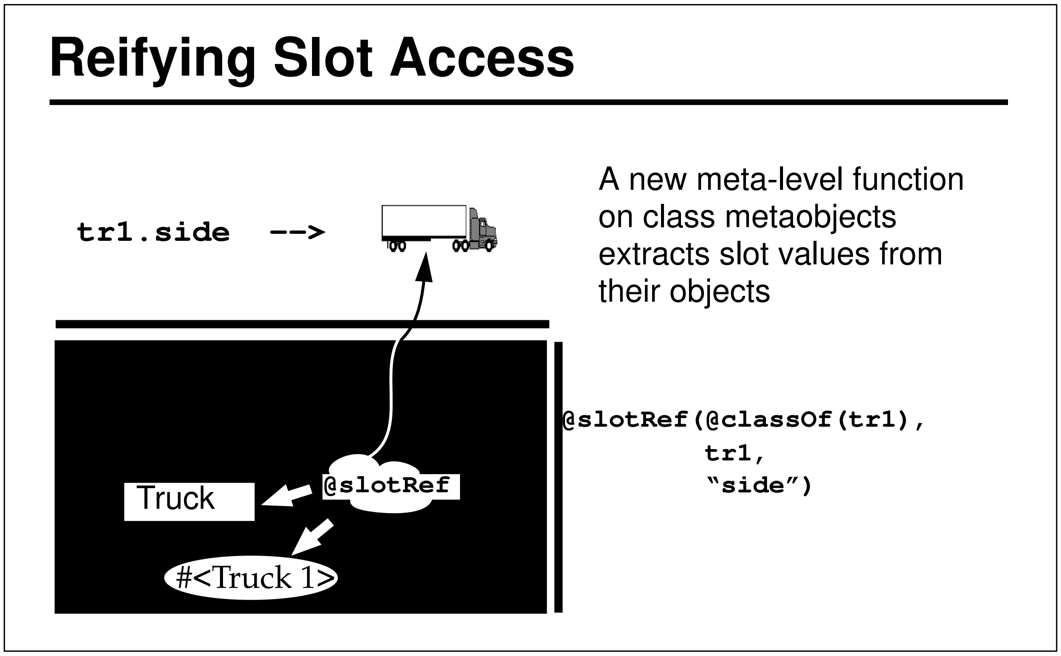
...to which the programmer adds his own methods, specialized to subclasses of **`@Class`**.

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

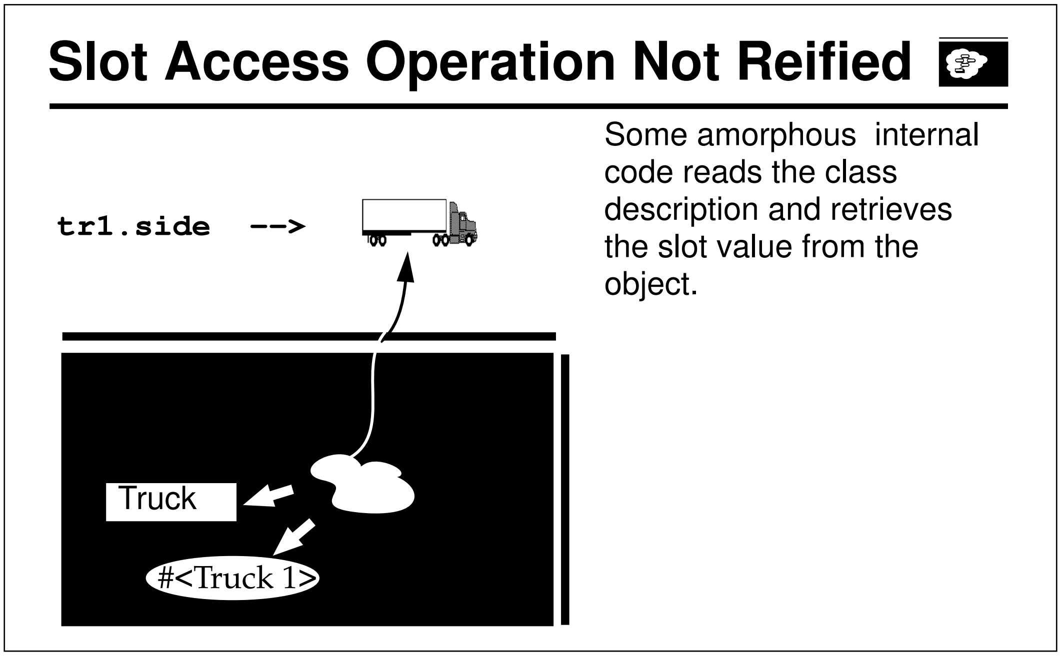
Making `@slotRef` Extensible (1/2)



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`**...



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 **`@make`**, 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.



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 120, 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...

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.

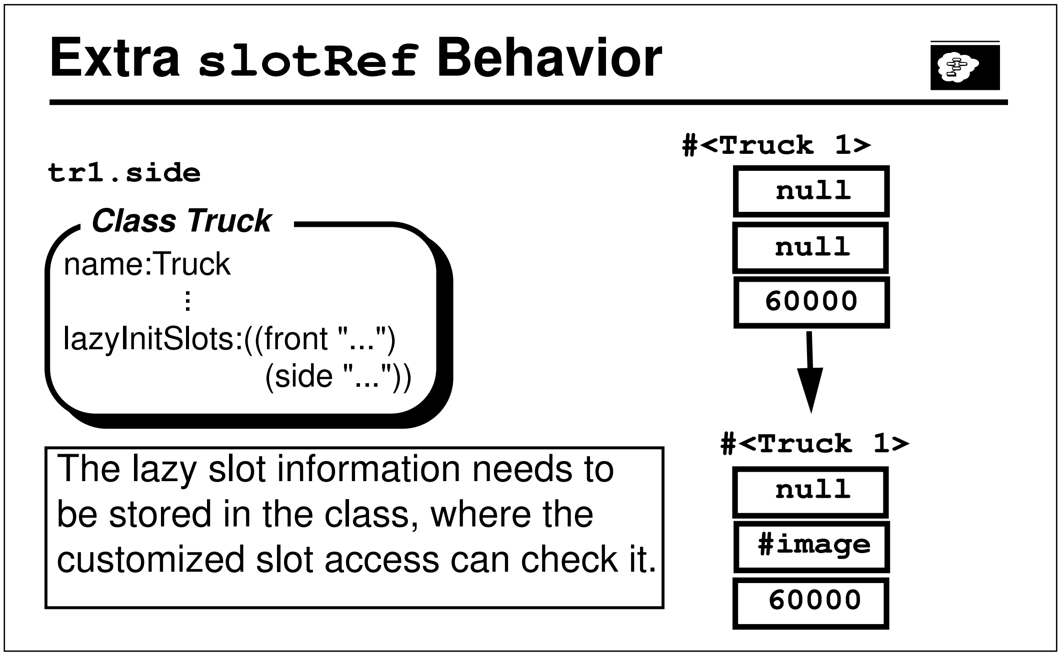
The obvious place to intercede with slot access is the function which is responsible for handling operations like **`tr1.side`**. The problem is that this operation is not reified. So the programmer cannot cleanly provide any facilities that add or change functionality around slot access.

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.

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.



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.

How to Make This Work?

The programmer needs to do two things:

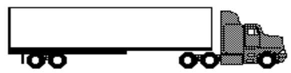
1. Classes must have information on which slots have lazy initializations—this is simple intercession.
2. The behavior of slot access must be augmented—more MOP will be needed to do this.

The first of these is straight-forward. 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.

The Programmer Would Like to Write

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

trl = new Truck(60000);

trl.side -->
```



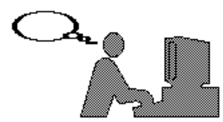
This slot access should load the image, and save it away in the slot.

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 154. The **@LazyInitClass** field after the slot list is a specification of which class metaclass **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...

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.

Chapter 6

But I Could Make it Run Better For My Application…

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
- ☐ • Performance adjustments

Anything Else to Try?

Well? Can you think of times when you wished in retrospect that you were 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?

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.

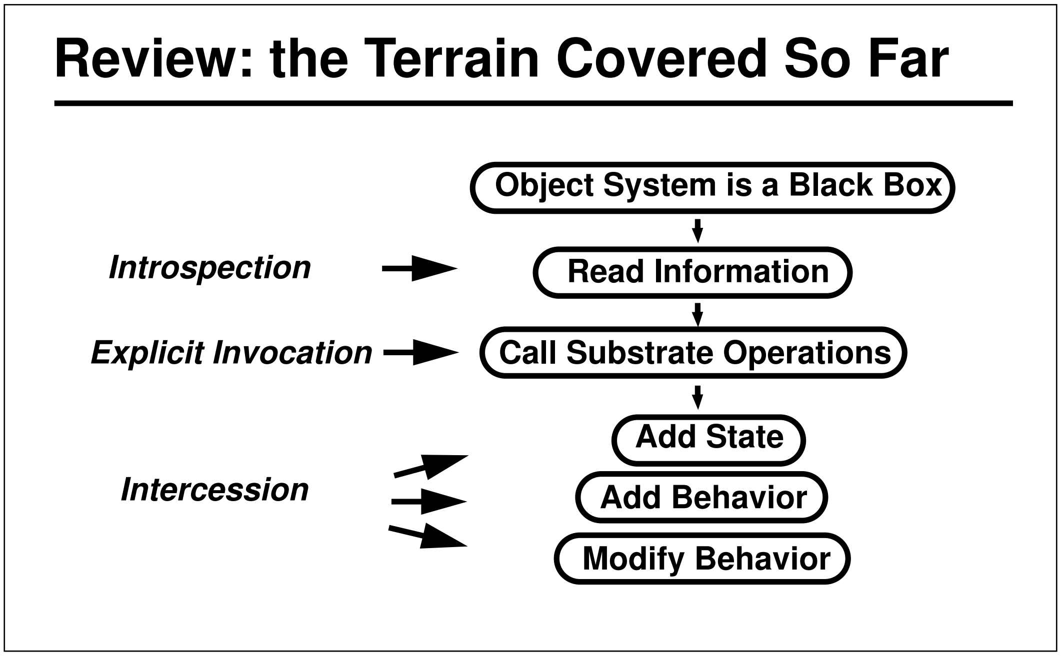
Review: How and Why

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

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.



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.

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.

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.

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”.

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.

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.

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.

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.

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 implemention
  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.

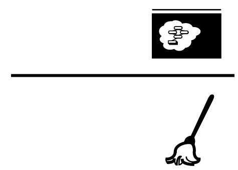
But what about coherence? Does programmer control over CPL computation jeopardize system usability?

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.

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]

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.

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`**.

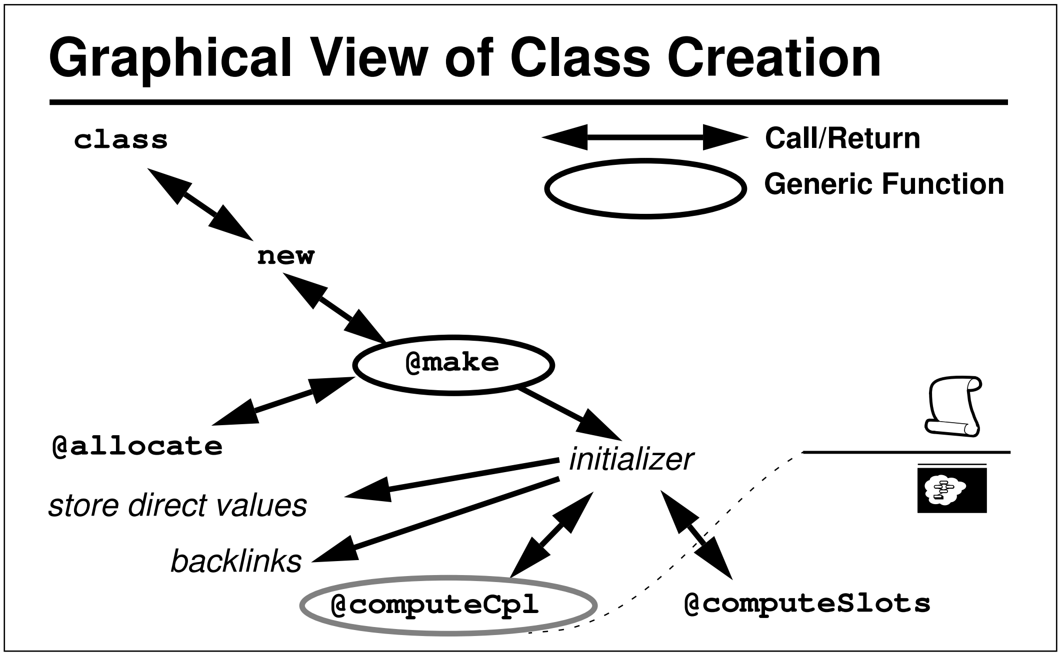
Programmer Control Over the CPL

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.



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 **`@make`**. 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.

Details of Class Initialization

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

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

  updateDirectSupers(directSupers, c);             backlinks

  c.cpl = @computeCpl(c);                          derived
  c.slots = @computeSlots(c);}                     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's CPL. Finally, she has found the choke point a protocol extension might use to allow the programmer into the CPL construction process.

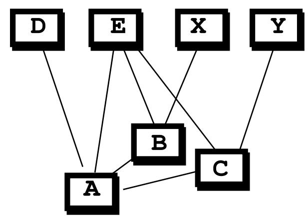
Looking for CPL Construction Site

```
class A (B, C, D, E) ();
    ⋮
    new @Class(...)
        ⋮
      @make(@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 132). And it is known that **`new`** calls **`@make`** (page 168). 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.

Remember Class Creation (2/2)

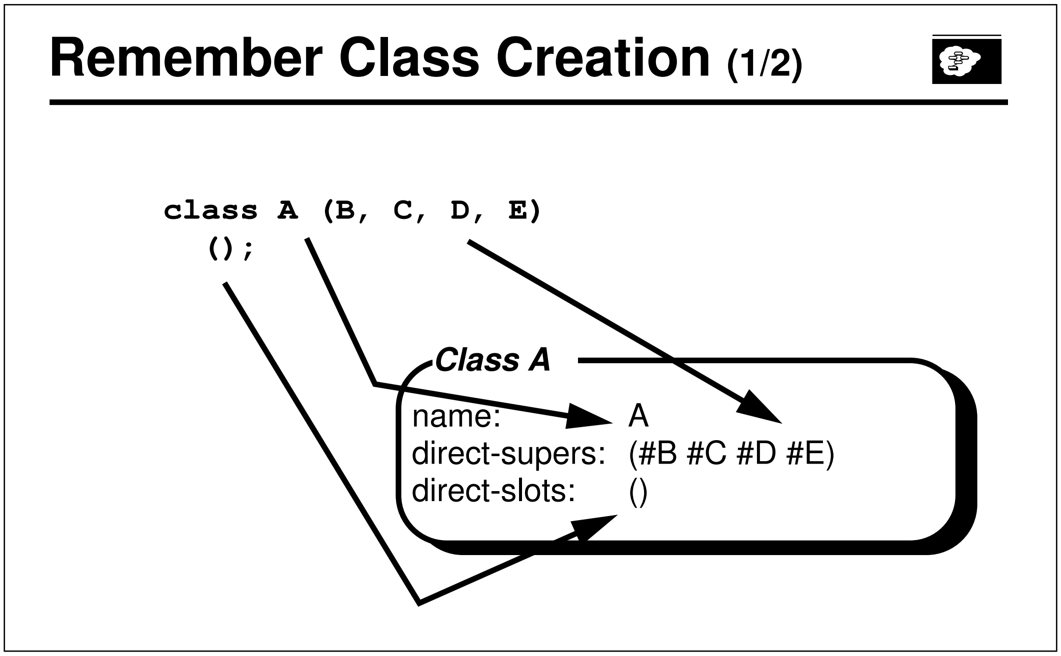


Class A

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.



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.

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's 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?

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.

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?

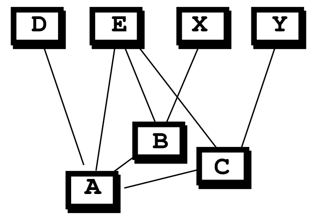
The first point to note is that the class precedence list is the key to inheritance. This was made clear on page 45 and through the subsequent examples there: when method dispatch is computed to execute a call to a generic function, the CPL of the class of the first argument is the determining factor for deciding which methods to run and in which order.

How could this insight help understand what this programmer needs?

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.

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?

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.

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 intercessionary 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].

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.

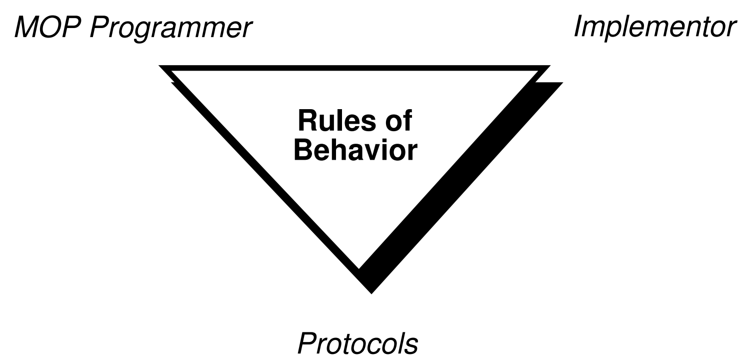
Rules for Introspection

These rules ensure integrity of data structures and implementor freedom:

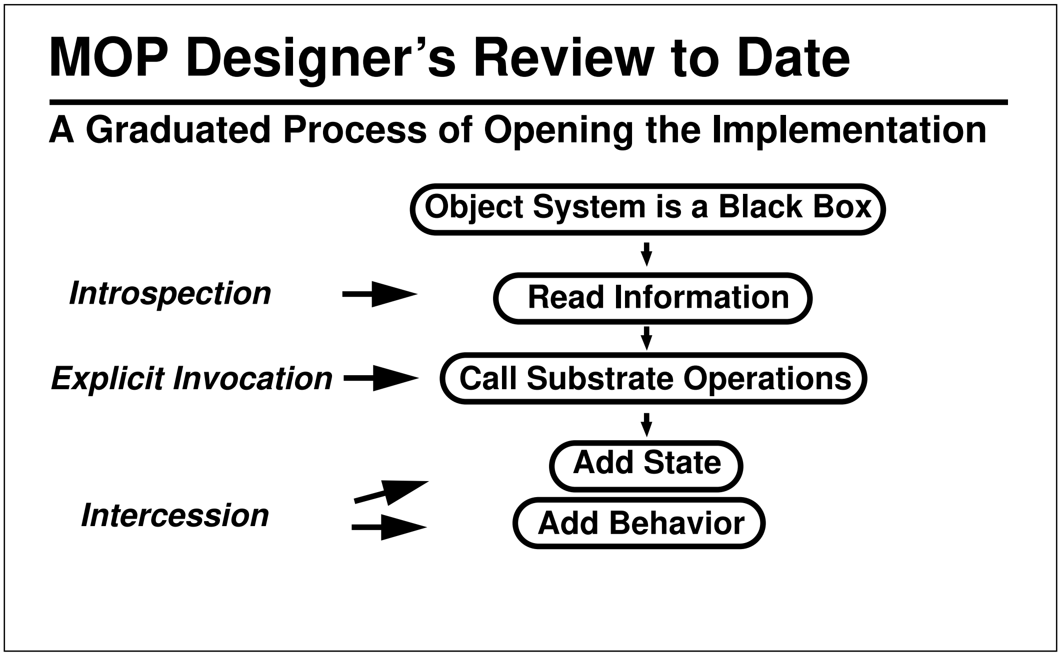
- No assumptions about ordering in sets (for example, **`@classDirectSubs`**)
- Do not modify lists or sets
- Successive reader results may or may not be identical
- Lists and sets may change if program changes (loading/reloading)

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 Mediate



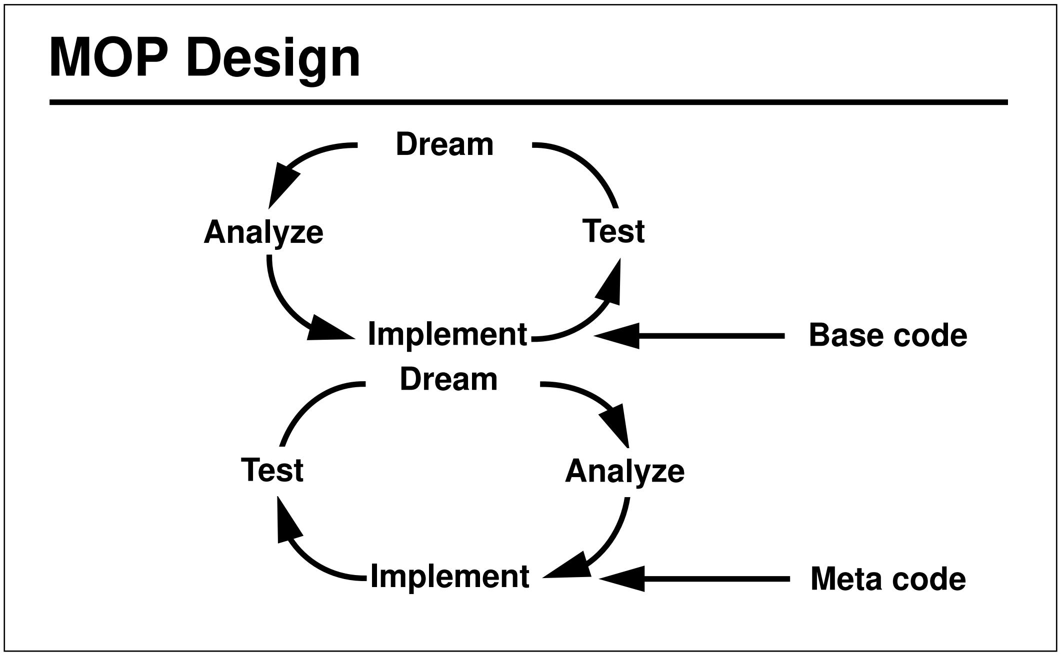
What we have implied as well throughout is that when programmers work on the meta interface, their behavior must be moderated by "rules of behavior", as is customary for software engineering in general.



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 **`@make`** 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.



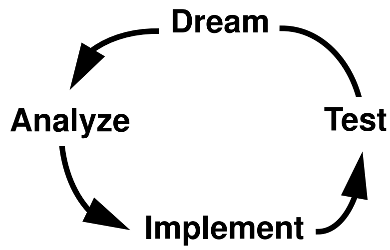
Here is the whole picture. Programmers select or construct substrate configurations that best suit their needs. If they cannot implement a new configuration they need by using the existing MOP, their problem becomes the subject of the MOP designers' concern. The application programmers' meta interface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

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.

Review of This Extension (2/2)

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


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

Concurrency Update frequency Size

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

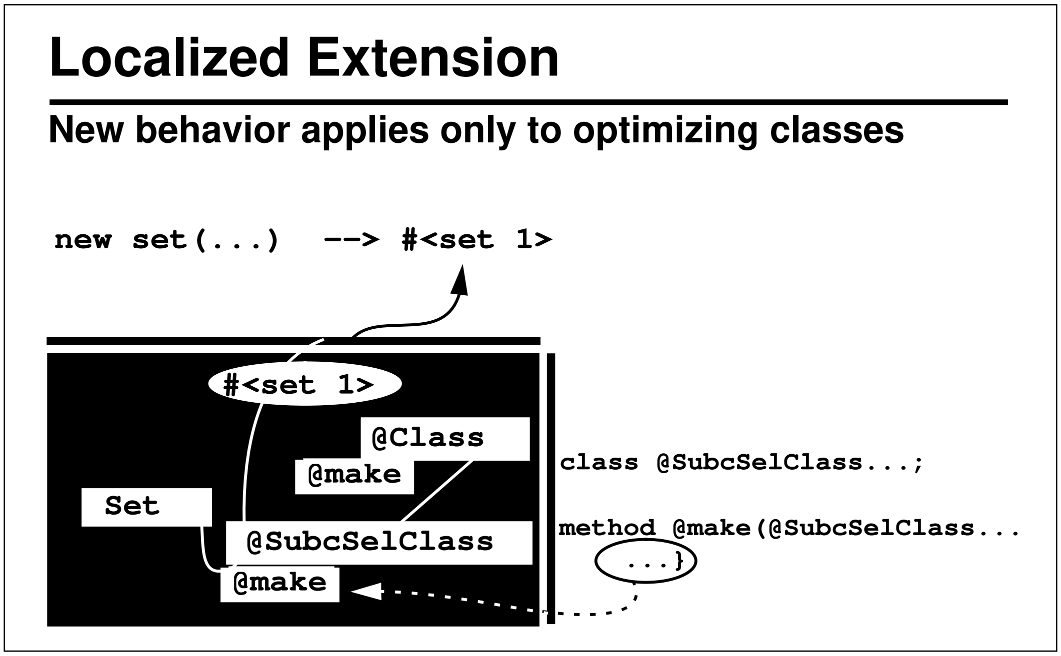
And here is what base object system programmers can now do.

> 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 **`@make`** method finds each of **`Set`**'s subclasses, obtains its evaluation function and calls it. Finally, **`@make`** instantiates the subclass who's evaluation function returned the winning bid.

Review of This Extension (1/2)

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

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

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

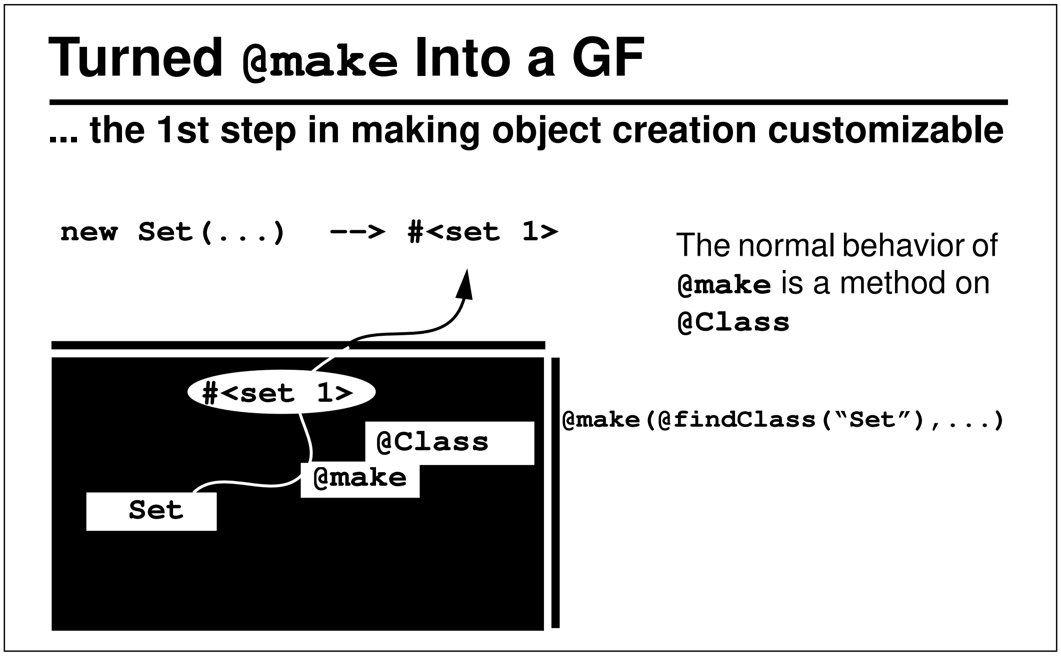
Here is all in one place the code programmers wrote on the meta interface to implement the optimizing subclass selection extension.



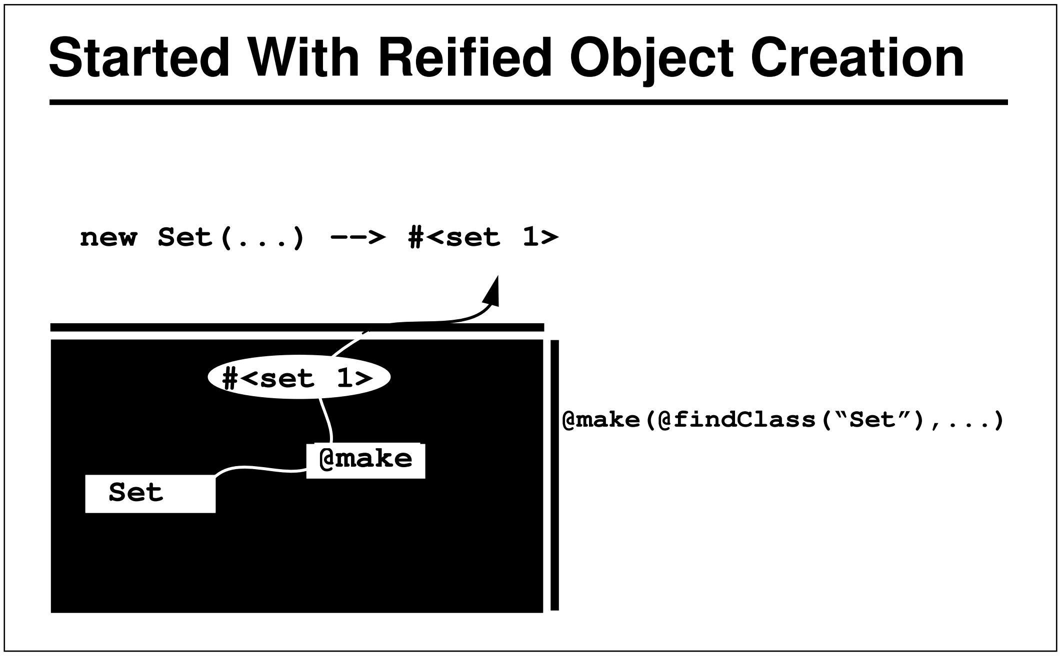
Since the protocol already allowed subclassing of metaclasses, no other extensions to the MOP were required, other than explicitly allowing meta interface users to add methods to **`@make`**.

Note that the ability to add an **`@make`** 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 **`@SubcSelClass`** 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 straight-forward.



The next step towards allowing the implementation of automated subclass selection was to turn the **@make** 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.



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 **`@make`** function was introduced in Chapter 4. It worked with classes, such as **`Set`** to create objects with the correct number of slots.

Documenting This Protocol

`@make` 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 **`@make`** 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.

Selecting the Subclasses

Now the MOP programmer could do this:

```
method @make (@SubcSelClass c, initargs) {
  var subc = applyFn(c.selectFunc, initargs);
  return @make(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 **`@make`** 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 **`@make`**?
>
> 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 metalevel 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`**.

Fitting it Into the Metalevel Model ?

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

```
method @make (@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 **`@make`** were called, then the protocol would ensure the existence of a “choke point” through which all object creation had to pass. If, furthermore, **`@make`** were a generic function, then programmers could add behavior to the object creation process by adding methods to **`@make`**.

Playing this out, it is clear what the **`@make`** 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 **`@make`** would be for class metaobjects. They would not be base level methods. For example, programmers would not write an **`@make`** method on **`Giraffe`** but on **`@findClass("Giraffe")`**.

Notice that this is more than what the protocol on page 122 provided, where programmers were first allowed to call **`@make`** to trigger object creation. What goes beyond that specification is (i) that even the implementation is guaranteed to call **`@make`** when creating objects, and (ii) that programmers can write methods on this generic choke point function.

OrthoSpec.frm

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 155. The **`@SubcSelClass`** would be a metaclass 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.

Analysis

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

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

Both **`new`** and **`@make`** 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.

What is Being Asked for?

Programmers using the set functionality should be able to write:

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

Concurrency Update frequency Size

```
--> #<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.

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.

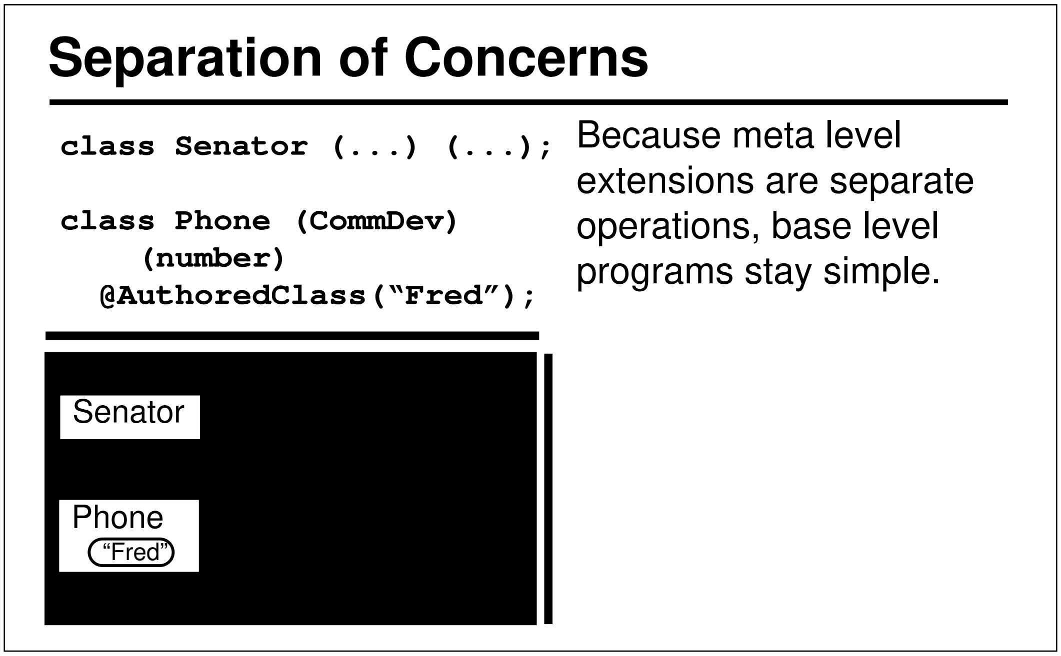
The programmers again start by imagining an ideal object system for their purposes. Here is a way this could look.

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 metaclass called **`@ClassVarsClass`**. Now reimplement the class author facility using class variables. Did you define a new metaclass or a new base class to do this?

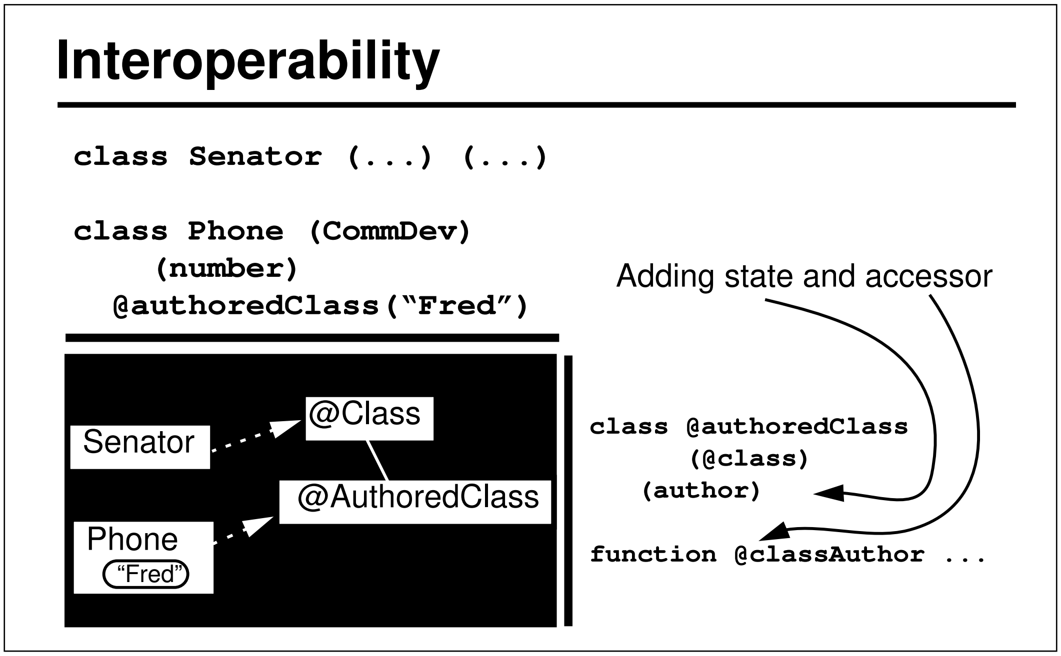
In addition to class variables, class *properties* are often a useful facility to have. They differ from class variables in that they do not contain state. One occasion for their use would have been the example on page 117 and page 118 where animal species were modeled with properties such as furry and 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 unecessary 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.



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 metalevel 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.



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.

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 metalevel 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, intercessionary 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.

Supporting Extension

- Programmers may define subclasses of **`@Class`**.
- The class definition primitive **`class`** supports an option which selects the metaclass 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 metaclasses 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 applicaton. 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.

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.

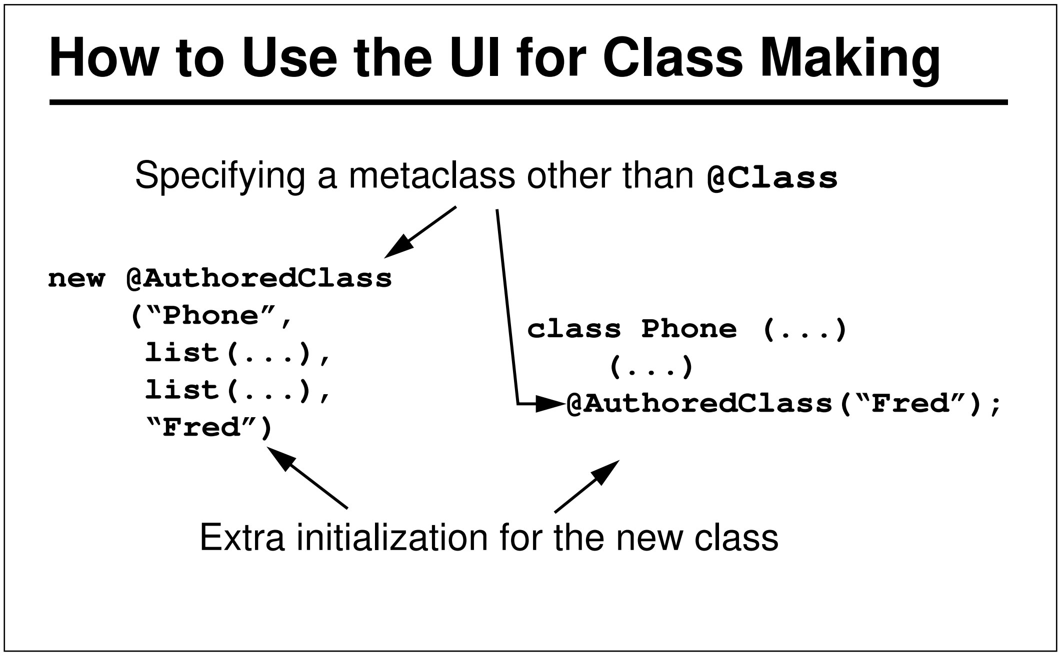
Having ensured that the new metalevel capability of subclassing would work for the programmer’s request, it needs to be documented as part of the protocol...

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.



Here is a new syntax for the **class** statement which allows an additional argument that determines the metaclass 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 148. The author is not introduced by a keyword. Instead, a fourth argument to the **class** primitive takes the non-standard class metaclass 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.

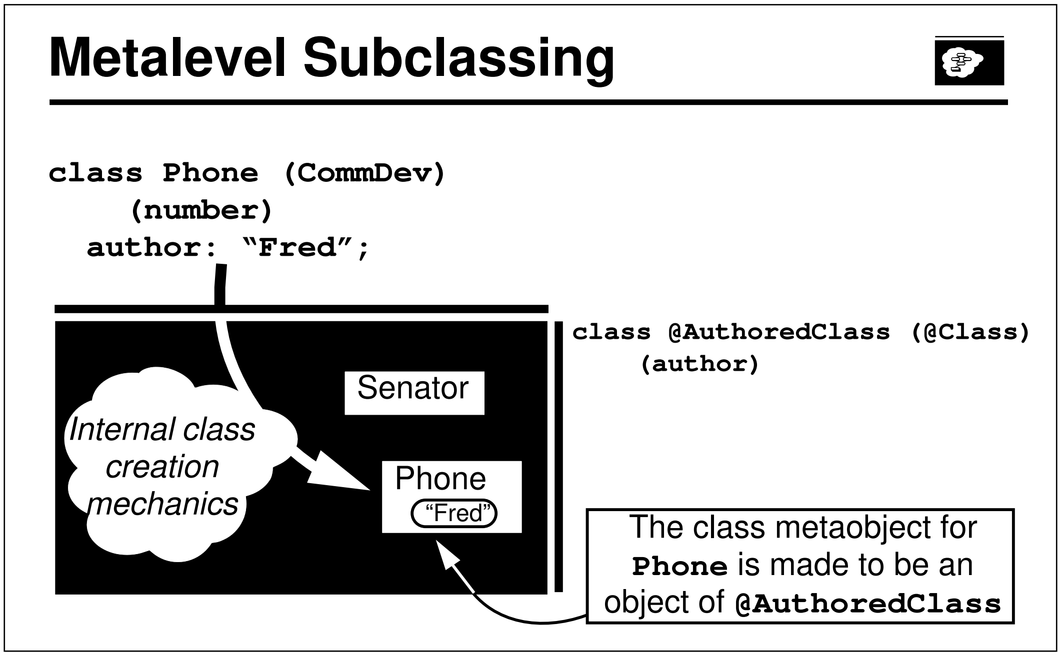
Defining an Authored Class

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.



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?

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.

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.

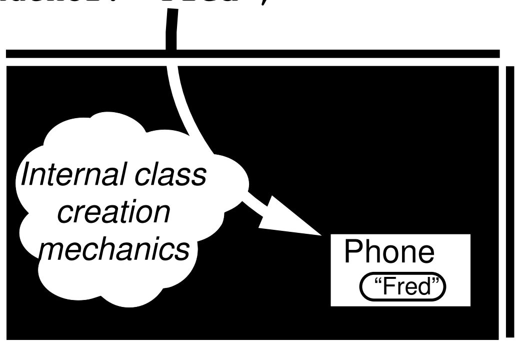
One possibility would be for the MOP designers to add the concept of class authorship directly into all class metaobjects and to provide a respective reader and initialization mechanism. But that would be inappropriate, because it would solve much too narrow a problem. The next day, other programmers 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.

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.

Analysis

This programmer wants to write something like this:

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

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

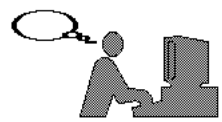
To address this request, the MOP designer sits down with the programmer to collaboratively work out a solution. In doing so, she employs a useful trick to organize the dialogue: they begin by focusing on what the desired functionality would have looked like if it had been part of TinyObjects in the first 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.

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.

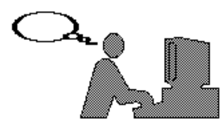
> Exercise: Using the facilities available in your environment, implement this functionality for yourself. You will want to design a special syntax for annotating a class definition with the author, and then have a preprocessor that goes through a file collecting the author information and storing it 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.

ciencies 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 metaprotocol—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.

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 Tiny-Objects 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 Tiny Objects to provide some *additional or different* functionality. Rather than asking the programmer to "code around" the defi-

Summary

- [x] **Introspection**—examination of selected system internals.
- [x] **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
- [ ] • Performance adjustments

This chapter has introduced a small but important addition to TinyObject's metaobject protocol. While the MOP this chapter started out with only allowed programmers to find out certain details about particular components of the substrate implementation (introspection), they can now write programs that explicitly cause substrate actions to happen at runtime, based on information acquired by the program at that time.

In the next chapter, we will examine how MOP designers can satisfy programmer needs for *extending* the substrate.

Anything Else to Try?

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

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

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.

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.

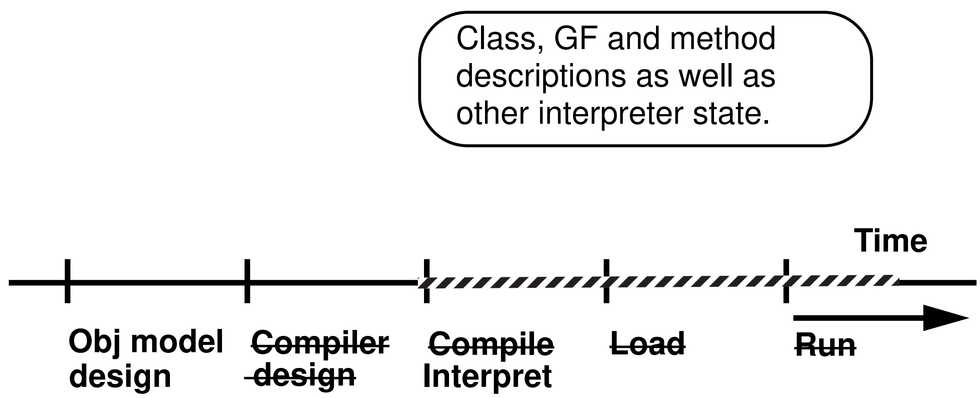
Runtime usage checking might be necessary.

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 before any metalevel interface came into existence. Any given implementation will be no more or less resilient to such activities now that the MOP of this chapter has been introduced.

When operations are made explicitly invocable which had not previously been accessible from the base interface at all, more care may be needed to avoid introducing new failure modes. Note, however, that any 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.

Interpreter Model



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, 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 by incremental compilers. The reader is asked 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.

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 implemention
  Can implementation be made robust?
  Can it be efficient?

MOP designers have shown in these previous pages how their new extension would be useful, and how the extension cleanly builds on existing notions: they build on the existing metaobject concept by allowing programmers to create those metaobjects. This helps preserve coherence of the system.

Regarding implementability, we first note that, again, MOP designers are not asking implementors to provide any new operations. All they ask for is that programmers be allowed to trigger operations already performend in all implementations. But let us acknowledge a question that may come up for readers mostly familiar with compiling environments.

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. Here are some examples. Apart from the obvious applications of browsing, editing and debugging, program transformations or applications in the area of knowledge representation come to mind.

Example of GF and Method Making

```
new @Gf(“makeNoise”, list(“animal”))

@addMethod(@findGF(“makeNoise”),
           new @Method(@findClass(“Bear”),
                     @findFunction(“bear_growl”)))

function bear_growl (b) {
  openMouth(b);
  generateGrowl();}

@applyGf(@findGf(“makeNoise”), list(smokey))
```

Here is an example of how programmers could now write programs that create generic functions and methods for them, and how their programs could 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`** call retrieves the new generic function metaobject and creates a new method metaobject specialized on the **`Bear`** class, with the function **`bear_growl`** as its body. The **`@findFunction`** takes a function name and returns a pointer to that function.

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

It is clear how a less messy-looking **`@makeAGf`** or **`@makeAMethod`** could be written from these primitives. What is important to note is that by not simply providing an **`@makeAClass`** function, and instead going deeper and making the formerly purely representational meta interface effective, MOP designers have provided the conceptual foundation for generic function and method creation for free.

DirectManip.frm

Methods and GFs Are Analogous

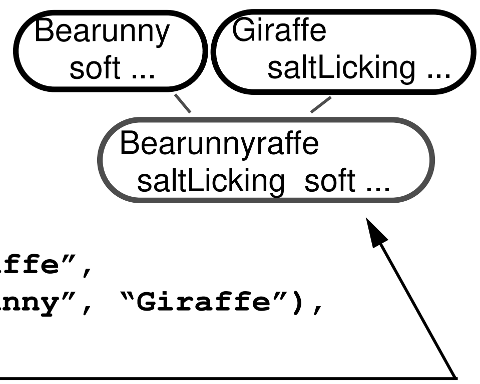
- The programmer can make objects of class **`@Gf`**. Initargs are *name* and *arglist.*
- The programmer can make objects of class **`@Method`**. The initargs are *specializer* and *function.*
- The programmer can call **`@addMethod`** to attach a method to a generic function.

All these will result in the creation of the corresponding base level components, or operations on them, respectively.

Remember that MOP expansion should look beyond the solution to a particular problem. Our MOP designers therefore proceed to make the obvious analogous extension for generic functions and methods. They 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 underlying implementations.

We will not go through the detailed explanations for these, providing instead the final result in the picture above. Programmer requests similar to the one for class creation can drive the design of generic function and method creation protocols.

Making Custom Classes (2/2)




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

Here is a graphical view of how a call to **`@makeAClass`** will work.

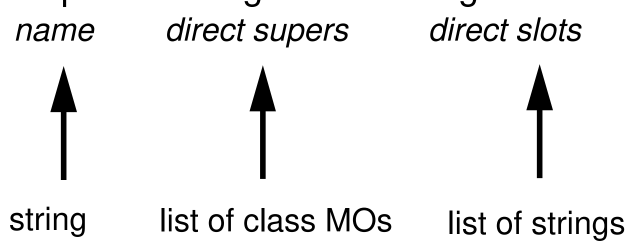
Making Custom Classes (1/2)

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

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

This is the function programmers have requested. It uses the new piece of metaobject protocol. Since **`new`** requires a list of class metaobjects, the helper function **`@findClasses`** is introduced which simply produces a list of class metaobjects from a given list of class names.

Documenting Class Making

- Programmers may instantiate **`@Class`** to create class metaobjects.
- Creation of a class metaobject will cause implementations to create a corresponding base level class.
- Required initargs for creating class metaobjects are:



Because programmers are now able to make (meta-)objects of **`@Class`**, and because this action causes base level class creation, they can now create such base level classes even when their names and superclasses are not known at compile time.

> Question: Are you making guarantees about how the implementation actually works? Are you saying that every class actually is an instance of **`@Class`** on the inside?
>
> Answer: No, MOP designers are not requiring that classes really have to be instances of **`@Class`**. What they have 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 metaprograms calling **`@make`** on **`@class`** will cause a new base level class to come into existence. This means that implementations remain free to implement classes any way they want. 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. But they will (i) ensure that the metalevel abstraction will always reflect what is happening in the implementation and that (ii) metalevel operations truly are first-class operations that have their intended effect in the implementation.

As an exercise to check whether this piece of protocol fills the bill for at least these programmers, here is an implementation of **`@makeAClass`**.

Programmers Making Class MOs

`@Class` could be the published class of class MOs.
Initargs for making class metaobjects could be:

Name: **`"Bearunny"`** ← **String**

List of class metaobjects

Direct supers:
```
list(@findClass("Bear"),
     @findClass("Bunny"))
```

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

The MOP designers will document that class metaobjects are made from **`@Class`**. When programmers use **`@make`** to create class metaobjects from **`@Class`**, they get to provide the obvious initialization arguments.

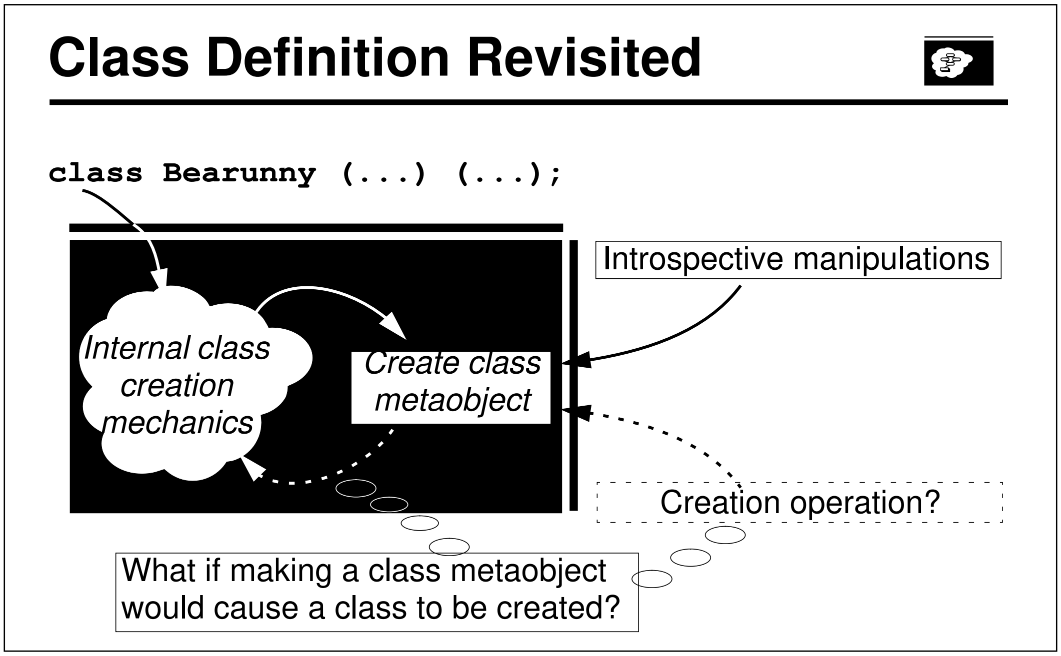
Just like **`margins`** and **`concerns`** were initialization arguments for objects of **`Senator`**, so **`directSupers`** and **`directSlots`** will be initargs for objects of **`@Class`**. The **`@makeAClass`** function of page 128 would simply have to perform such a metaobject creation. But first, the MOP designers need to make this expansion of the protocol official...

just provide that **`@makeAClass`** function and be done with it. But this book, and therefore the characters within it, have larger and more heroic ambitions than such local solutions. As will become clear, this piece of protocol will go much further than an **`@makeAClass`** function would.

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 implementational 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.

But before getting carried away with the cleverness of this MOP designer, let her do some more leg work to see how this proposal will look in practice.



When a class is defined, implementations perform all their internal work which might be different from one implementation to another. As indicated by the solid arrow out of the code cloud in the figure above, all implementations also have to create a class metaobject, because the introspective protocol of Chapter 3 requires them to do that.

The fact that the internal mechanics of creating a base class also create the corresponding class metaobject gives the MOP 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?

The MOP designer thus decides that there will be a published metaclass from which the programmer is allowed to create class metaobjects. Such instantiations will then be the signal for implementations actually to go ahead and make the corresponding base level class.

This might seem less direct than the **`@makeAClass`** function first contemplated in conversation with the programmer on page 128. If the MOP designer wanted only to solve this current problem, she would probably

Analysis

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

```
@makeAClass(“Bearunnyraffe”,
            list(“Bearunny”, “Giraffe”),
            list())
  -->  #Bearunnyraffe
```

New class

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

Object of new class

But right now, only **`class`** knows how to construct classes.

Again, the metaobject protocol is not powerful enough to allow the programmer this level of control.[1] The MOP designer now employs a technique that is very useful. She sits down with the programmer, and they invent the piece of interface that would be most convenient for him. In this case, they ask which kind of facility the dream version of TinyObjects would have to fill his needs. This approach will be used again.

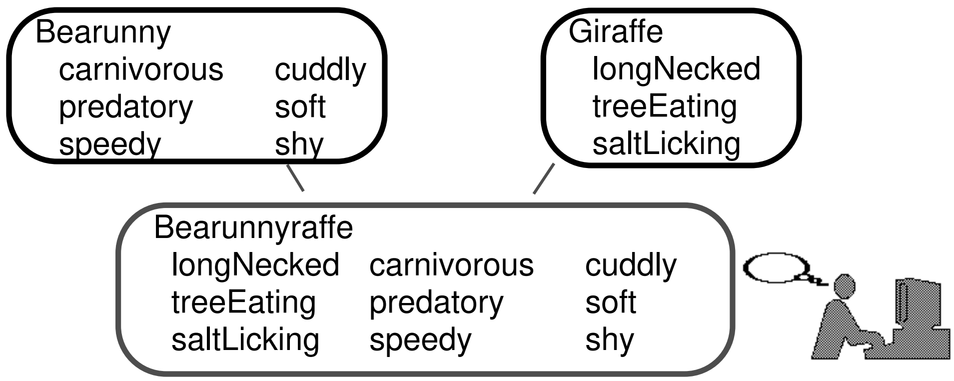
In some form or other, this programmer is asking for a function such as **`@makeAClass`**. The function would take similar arguments as **`class`**: a class name, a list of superclasses and a list of slot names.

Once this basic approach is decided on, the MOP designer needs to look at which parts of the class creation mechanism would be common to all implementations, so that she knew how to provide a properly abstracted access to the class creation operation, without making life too difficult for implementors.

1. Note that these frequent shortcomings of the MOP are due to the fact that for the purpose of teaching the material,TinyObjects was started with no MOP at all, using the 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.

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?



Since this worked so well, the programmer comes back and says he would like to allow children to make new species of animals. When they click on Bearunny and Giraffe, the programmer would like a new species of animal to come into existence, with all the inheritance done right.

This sounds like a problem similar to what the MOP designer just solved with **`@make`**: the programmer is asking for the ability to explicitly cause an event to occur in the implementation. The base level comes close to being sufficient, because the **`class`** operator does the right thing—except that it assumes knowledge of the desired superclasses when the program is written. In this case, those superclasses will only be known at runtime when children start clicking on animals on the screen.

Another example showing how explicit invocation can sometimes be desirable, independent of what is available through the base interface, comes from the world of operating systems: the swapping in of pages is normally triggered indirectly by base level activity. But situations are imaginable where the programmer of real-time systems would like to invoke paging operations explicitly from his programs to avoid page faults during critical periods of time. [1]

1. Some operating systems allow programmers to specify that a process never be swapped out at all, once it is swapped in. That is a very rudimentary version of this facility.

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.
- Effect events not accessible through the base level.

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

Another way of looking at the capability introduced by the previous example is to say that application programs may now sometimes bypass the syntax prescribed by the base interface. That syntax was designed to be convenient for most cases. A good example is the **`new`** mechanism taking the name of a class, instead of a class metaobject. Nobody would regularly want to write **`new(@findClass(<className>))`**. So the base interface syntax is fine. But in this case, it was too restrictive, and the additional piece of MOP helped the programmer get around this restriction when the need arose.

The ability to explicitly invoke substrate operations can sometimes involve more than a bypass of inconvenient syntax. This happens when some operation is not even accessible through the base interface at all. For example, in object models with dynamic class redefinition, such as CLOS, updates of all instances whose class is changed are normally done lazily. Since lazy update is semantically indistinguishable from eager update, there is no provision for triggering instance updates through the base level. But for performance reasons, the programmer might want the ability to force update at a convenient time for given sets of objects.

Roadsigns to Remember

Note that this sign was just used in a new context. It marked material showing a process that inherently occurs in all substrate implementations.

Previously this sign only marked information inherently maintained in implementations.

On page 59 this symbol was introduced as marking illustrations with information all implementations of a substrate inherently need to maintain. On page 120 the symbol was used in a slightly broader context. There, it marked an illustration showing the representation of a process that inherently needs to happen in all implementations. The symbol will be used for both of these purposes from now on. Sometimes the material marked with the symbol will look like a piece of code. In those cases, it is to alert the reader that this code is not necessarily the program actual implementations use. It is intended solely to communicate what inherently needs to happen in all implementations.

Making Those Animals

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

Now the question marks in the program of page 119 can be resolved. The code above uses the new **`@make`** access to object creation functionality in its loop.

Making Objects

Another Protocol Expansion

Given a class metaobject, the programmer may create objects of the corresponding class by calling **`@make`**. Like **`new`**, it accepts a list of initargs.

The following calls are equivalent:

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

```
new Bunny()
```

Here is the MOP designer's latest proclamation, signing this capability into law. It from now on allows the programmer to write programs which create instances of classes known only at runtime.

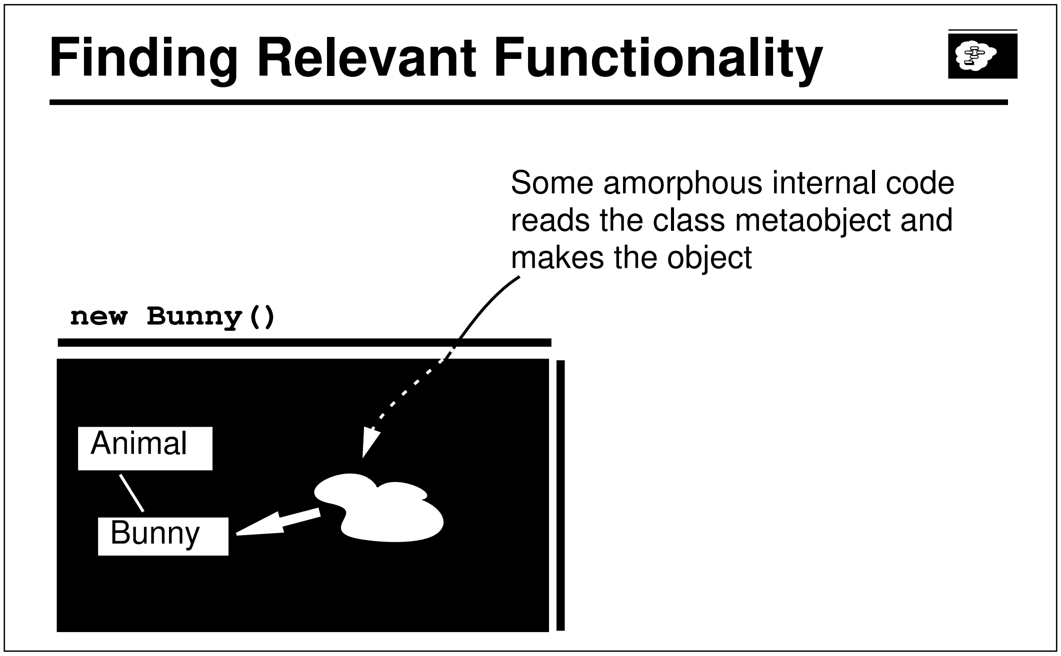
Note that this piece of protocol is qualitatively different from the ones in the previous chapter. Those pieces of protocol enabled introspective capabilities. This piece allows the programmer to effect events in the substrate—in this case the creation of objects.

Reifying Object Creation

A new meta-level function operates directly on class metaobjects to create objects

The new **`@make`** function above becomes part of the meta interface in the figure. It takes a class metaobject and returns an object of the corresponding class. How any particular implementation actually accomplishes this remains hidden from view.

`@make` is simply a function the programmer may now call with a class metaobject to have an object of the corresponding base level class created. Note that the MOP designer is not even requiring implementations to call **`@make`** for their own object making activities.



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.

Clearly, what the MOP designer needs to do is to enrich the meta interface to include 'running the cloud'. Checking against the needs of implementors, she realizes that every implementation must somehow have a version of the cloud that does the work of object creation, which conceptually is expressable in terms of a more primitive operation on class metaobjects.

Here comes the step that makes the cloud concrete.

But That's Easy...

Remember: this is
a class **metaobject**

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

Programmer has a class metaobject, but the special form **`new`** takes the name of a class. What to do?

Sounds easy. Here is the code—almost. It makes use of the **`@allSubs`** function written earlier. 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...

As shown in the figure, this runs into a snag: The primitive **`new`** takes the name of a class at compile time, not a class metaobject. 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 is a hole in the base interface. Note, however, that implementations certainly do create objects at runtime. This programmer is therefore not asking for anything impossible or even particularly difficult. Unfortunately, since the meta interface introduced in the previous chapter is limited to introspective facilities, the meta interface is not powerful enough to help out either.

This is therefore a case where the object system (and therefore meta interface) designer must get involved. Here she is, going to work.

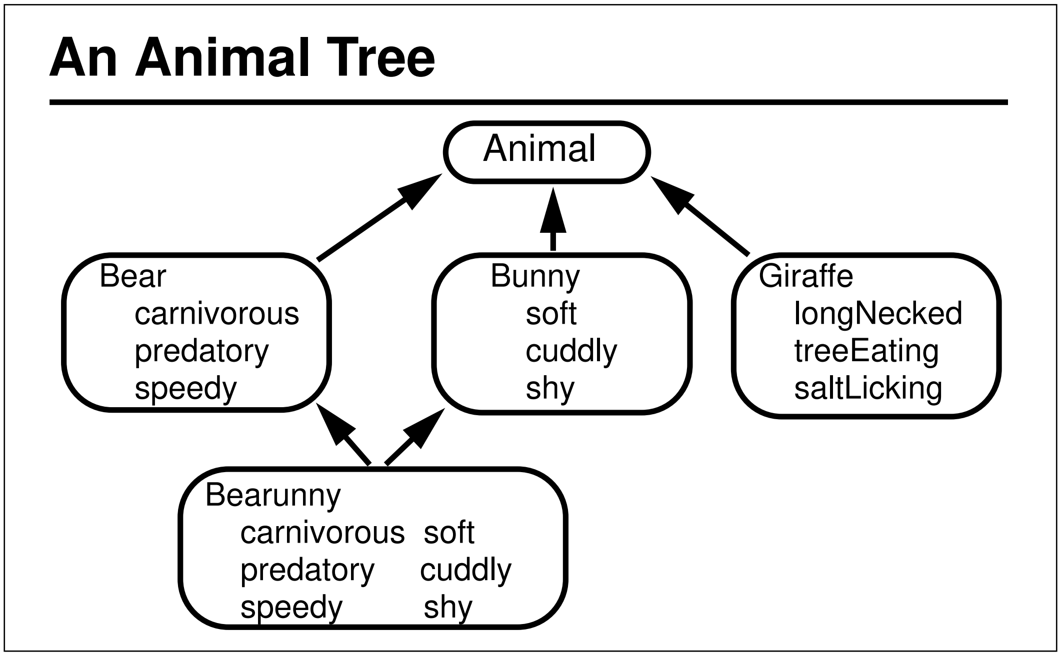
Species Are Represented as Classes

```
class Animal () ();

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

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

One obvious way for the programmer to proceed is to model species of animals as classes. Since each species has attributes, the programmer needs some way of keeping that information with each class. In the figure above, this is done using slots, even though this means distorting the slot concept. Slots are meant to hold state in each object of a class, not attributes that are valid for each of the objects. If this feels wrong, that feeling is a good intuition. A better solution will be found for this later.

This programmer knows how to write a program to generate the picture of page 117 from these classes, right? 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`**).

If a child clicks on **`Bunny`**, objects of **`Bunny`** and **`Bearunny`** are to be created. Programs to do this would get a hold of the respective class metaobjects. Given those, they would find all subclasses and would finally create objects of these.

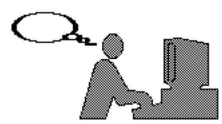


Graphically, the program in question looks like this:

Kids should be able to click on an animal. The system would then create one animal object for each of the animals that are below of and reachable from the one the child clicked on.

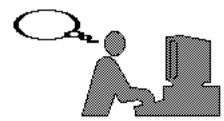
Going to the Zoo…

My computer game uses cross-breeding of animals to teach children about object-oriented programming.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.



This programmer clearly has a rather interesting approach to teaching object-oriented programming. But rather than getting into an argument with him about the similarities between cross-breeding and multiple inheritance, here is an analysis of how the base and meta interfaces to TinyObjects live up to this challenge.

Chapter 4 Alternative Interfaces to Existing Functionality



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

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

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
- ☐ • Performance adjustments

The following material will show how other kinds of programmer needs can be addressed by providing other kinds of access. But if the programmer using some substrate needed nothing beyond the introspective capabilities developed in this chapter, this would be a coherent place for him 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.

MOP Approach to Obj System Design

Be open-minded about meeting programmer needs by giving him access to previously hidden substrate state through appropriate abstractions. This will allow him to solve many of his problems himself. Introspective access provides solutions to “what’s going on” questions.

Programmers need not allow themselves to be this easily dismissed. The interaction between programmers and MOP designers is fruitful for both parties. The same is true for implementors who can make their systems more useful to more people by being able to respond to programmer requests.

Pretty Smooth Going, But...

Normally, protocol design requires several iterations, with designers, substrate implementors and application programmers involved.

All this protocol development and implementation went smoothly. In general, more iterations are required to get everything right.

Normally, the designer of a substrate like TinyObjects would construct a design with a regular base interface and an initial, suitably abstracted meta interface to parts of its state as shown in the preceding examples. The programmer would then use the base interface as intended. If his programs required information for which that interface was inappropriate, he would turn to the meta interface for access. If that interface was not powerful enough, he would need to return to the substrate designer and ask her to provide a more powerful solution. The designer would evaluate whether accessibility of the additional information compromised encapsulation and implementation efficiency concerns. Unless such objections were raised, the interface would be appropriately enriched and tested. Eventually, these iterations would cease, and a practically useful design would ensure substrate implementations that were open to principled inspection.

Often when programmers ask implementors for access to substrate information or behavior, they are dismissed with blanket statements about efficiency, or assurances that "the designers are very clever people."

Readers Would Be

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

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

These functions could provide the meta interface for this implementation.

Access to the information about classes is nicely encapsulated with these functions. For example, if the information were not kept in slots of class metaobjects, the interface would still work.

> Exercise: Take a moment to think about the tradeoffs that were made, and how much is really being asked of implementors. For example, some implementations 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. That means implementors might have to do some work to present the interface our MOP designers have documented.
>
> Now put yourself squarely 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 extensions we have made to TinyObjects? Try to implement the procedures **`@isGfApplicable`** and **`@methodGenealogy`** of page 86 and page 87 without them.
>
> 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...?

Objects for Encapsulation

How might the protocol be implemented?

```
@className
@classDirectSupers
...
```

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

The protocol allows a variety of implementations. One of them simply introduces a class **`@Class`** whose instances are the class description metaobjects.

Note again that implementors could organize their internal data structures any number of ways, including the use of non-object-oriented structures. While the programmer working with the metalevel interface can count on the existence of class metaobjects with their associated readers, he cannot count on classes actually being implemented as those metaobjects in every implementation.

Rules for Using Readers

- Do not modify data structures obtained through the meta interface, such as classes, methods, sets of items, etc.
- The data structures returned by successive calls to a reader may or may not be identical.
- No assumptions must be made about ordering in sets (for example, **`@classDirectSubs`**).

Rules for meta interface programmers are necessary to help implementors do their jobs. One way to make their implementations robust, is to be very conservative on the spectrum of page 105. This approach can be expensive. Enabling them to opt for cheaper solutions requires help from the protocol specifications. This is what rules of a protocol are for.

Essentially, they say that programmers must not stomp on data structures, that structures may or may not be copies of underlying information and that the implementation may update structures to reflect changes in the program.These rules ensure that implementors may—but need not—share any data structures directly.

Protocols thus contain not only the functional interfaces to representations of some implementation state, but also a set of rules about how those should be used. These rules make the specification of the central part of the protocol more concise. Example: The MOP designer does not need to specify “don't stomp on the result” for every reader she includes in the protocol. There are the readers, and the “no stomping” rule covers them all.

Throughout subsequent material, the MOP designer will add not only specific protocol, but also additional rules that govern its use.

Introspection2.frm

Robust and Efficient Protocol Design

- Protocols must preserve a range of implementation choices.
- Standard principles of design can produce clean, well-encapsulated interfaces and rules that govern the use of the results returned through them.
- Such rules enable, but do not require a shared-all approach.

Meta interface design is no more magic than any other kind of interface design. Standard techniques and requirements apply. These include the choice of information transfer method alluded to on page 105 (by reference, by value), care to provide minimal primitives that provide maximal power without being too inconvenient, appropriate rules regulating how interface clients are allowed to manipulate results, or which assumptions they are allowed to make about them.

> 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.

In order to prevent the introspective MOP from limiting implementation choices on the spectrum of page 105, the MOP designer publishes a set of rules for the programmer writing programs to the meta interface.

Desired Implementor Freedom

A Range of Data Structure Sharing Policies

Shared-nothing

No data structure sharing
➡ Provide only copies of internal structures.

Prevent shared structures from being changed.
➡ Read-only structures

Ask for shared data structures to be respected.
➡ Need clearly enumerated rules

Shared-all

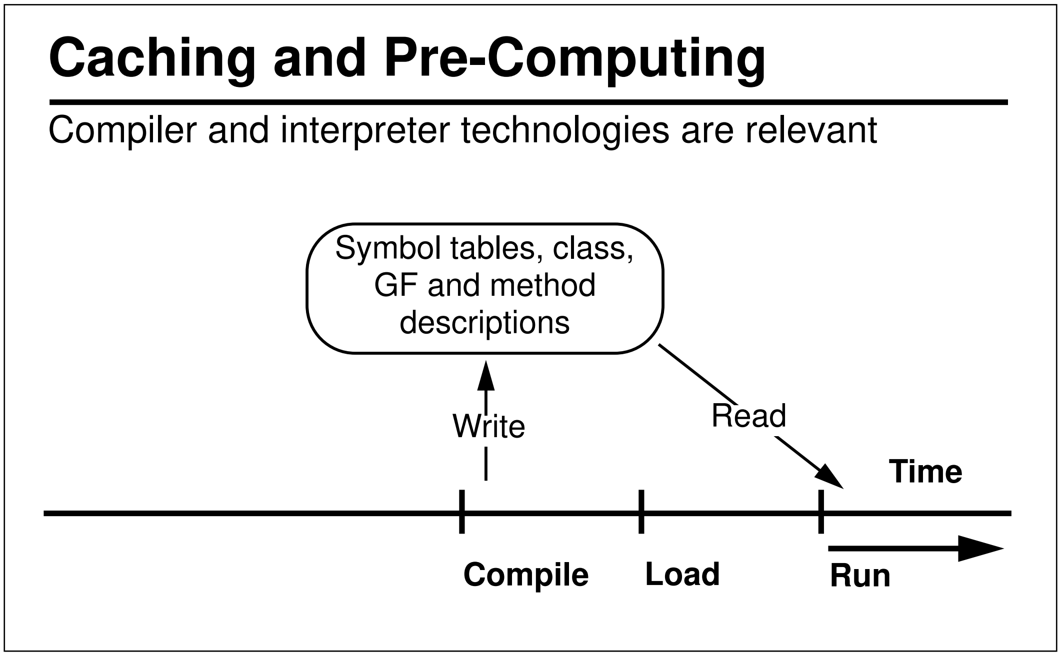
What about the question of robustness? Can this introspective MOP be provided without making the system fragile? Robustness, it turns out, is interrelated with questions of implementation efficiency.

As with other data interfaces, there is a spectrum implementors deal with for the data structures containing the information that is being made accessible. They might want to have the structures they hand across the interface be copies of internal structures, or, to save space and time, they might want to simply share their internal structures, relying on programmer care and cooperation. That choice will depend in part on the sensitivity of the data in question and the use the implementation makes of it.

This is true for all systems. For example, in a window system similar questions might arise about who allocates/deallocates buffers, when they can be read and written, and so on.

One example in the MOP introduced so far is whether the programmer can accidentally modify data structures like the one returned by **`@classDirectSupers`**. The answer will be determined by the implementors' choice of sharing technique on the spectrum above. The MOP designer must design protocols that give implementors freedom on that spectrum.

Introspection2.frm



Regarding limitations on caching and pre-computing, note again that the basic technology for supporting metaobject protocol interfaces is well developed.

Thinking in terms of a compile-link-load model, the metaobject protocol makes compile-time information available to the programmer at runtime. The techniques required to do this are not new, since this is not so different from existing compilers that provide symbol tables for debugging at run-time. Providing descriptions of classes, generic functions and methods is just one step further. In fact, a good TinyObjects implementation would no doubt already have such facilities to support the debugging environment.

Note that in order to save space in the runtime image, the information could even be in a file, retrieved at runtime on demand, just like symbol tables.

If instead one thinks in terms of an interpreter model, providing the metaobject information is even easier. It will be 'floating around' in the interpreter already.

Feasibility of Implementation

Have implementors been overconstrained?

Robustness and efficiency:

- Are contents of internal data structures overprescribed?
- Are undue limitations on caching/pre-computing imposed?
- Is the safety of implementation data structures compromised?

The MOP designer needs to be certain that the new functionality will not be too burdensome to implement: Will it prevent important optimizations or otherwise interfere with performance—speed or space? Will it make implementations less robust? Or does it just take too much work to implement?

These questions are essential to an appropriate balance between gain to programmers and cost to implementors.

The first point of limitations on internal data structure contents is in many cases intrinsically addressed by the design approach taken throughout this book. The MOP designer always takes advantage of elements that are inherently already present in implementations. This minimizes the danger of conflict between what protocols require and the contents of implementation data structures. It is also important to remember in this context that the meta interface is just that: an interface. Its data structures do not need to have identical shape or contents to implementation data structures. This also helps preserve implementor freedom.

Coherence and Complexity

- The base-level interface remains unchanged.
- The meta-level interface is a simple and elegant reification of already existing concepts.
- A clear separation of base and meta means people know when to work in one vs. the other.
- The only new aspects introduced were ones relevant to the programmer requests.

A primary concern is whether adding all of this new functionality is going to bloat TinyObjects and thereby make it too complex. As was shown, the new functionality is of a somewhat different sort than the original one, and this distinction allows the substrate designer to separate interfaces to the two kinds of functionality, thereby alleviating the bloating effect.

The base interface remains tailored to the world of applications the substrate is serving. Information about the substrate's service and about the base interface itself is dealt with through the meta interface. This ensures that the base interface remains focused.

Further control over coherence and complexity is afforded by the use of object-oriented technology in the metaobject protocol: its encapsulation facilities allow the meta interface designer to ensure that the interface really does provide just information, rather than the data structures or other idiosyncrasies of some particular implementation. The ensuing chapters will show that other aspects of object-oriented technology also help control the complexity of meta interfaces.

Finally, the MOP designer was careful not to expose anything extraneous. Only material relevant to programmer requests was included in the MOP.

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?

In evaluating a proposed new functionality, the questions above are important to address. Our examples so far were driven by programmer requests, which provides a good indication for how the resulting new capabilities would be useful.

Through the next several pages, the other questions of complexity/coherence and feasibility of implementation will be examined. This is an exercise which will be repeated throughout the remainder of the book as further extensions are added.

Introspective Protocol

An *introspective* meta protocol allows selective examination of substrate state, 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 does not force implementations to expose the actual data structures they choose to use for their internal representations of classes and other components that are the subject of the meta interface.

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.

tocols for substrates that have nothing to do with objects, like virtual memory systems or relational databases. They might not even be implemented using object-oriented programming technology.

Inversely, while this book teaches an object-oriented approach to meta interface design, one could build meta protocols that are not object-oriented.

Metaobject protocol design is part of substrate design. In general, the substrate designer would therefore also produce a substrate's MOP. From now on, the terms "MOP designer" and "substrate designer" will therefore be used interchangeably.

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

A protocol on (interface to) metaobjects is called a **metaobject protocol**.

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

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*. In TinyObjects, there are three kinds of metaobjects: class, generic function and method. The class of any kind of metaobject is called a *metaclass*.

Traditionally, the term "metaclass" has been used only for the class of class metaobjects. This restriction is best thought of as a historical accident, and the use presented here transfers much better to meta interfaces for substrates other than object systems.

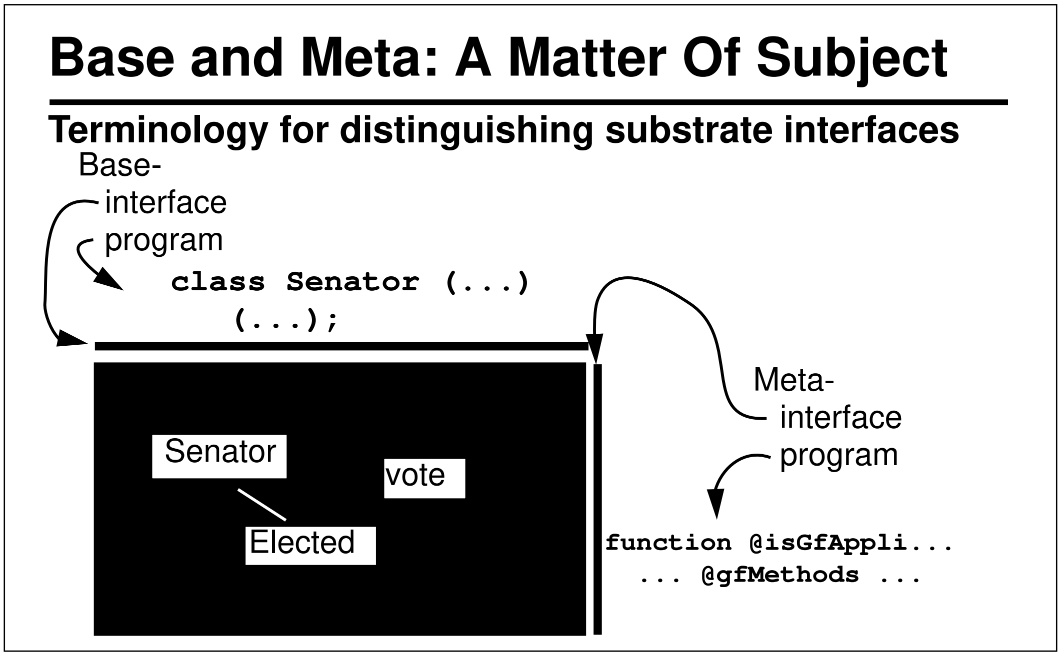
The interface to metaobjects is called a metaobject protocol (MOP). The term protocol is used, rather than the term interface, because it better fits with capabilities introduced later on, when the programmer will be involved in richer interactions with metaobjects than just the extraction of information from them.

The *object* in metaobject protocol indicates that the meta interface is object-oriented. It is not a statement about the kind of substrate the metaobject protocol is an interface to. One could, for example, design metaobject pro-

The Use of ‘@’

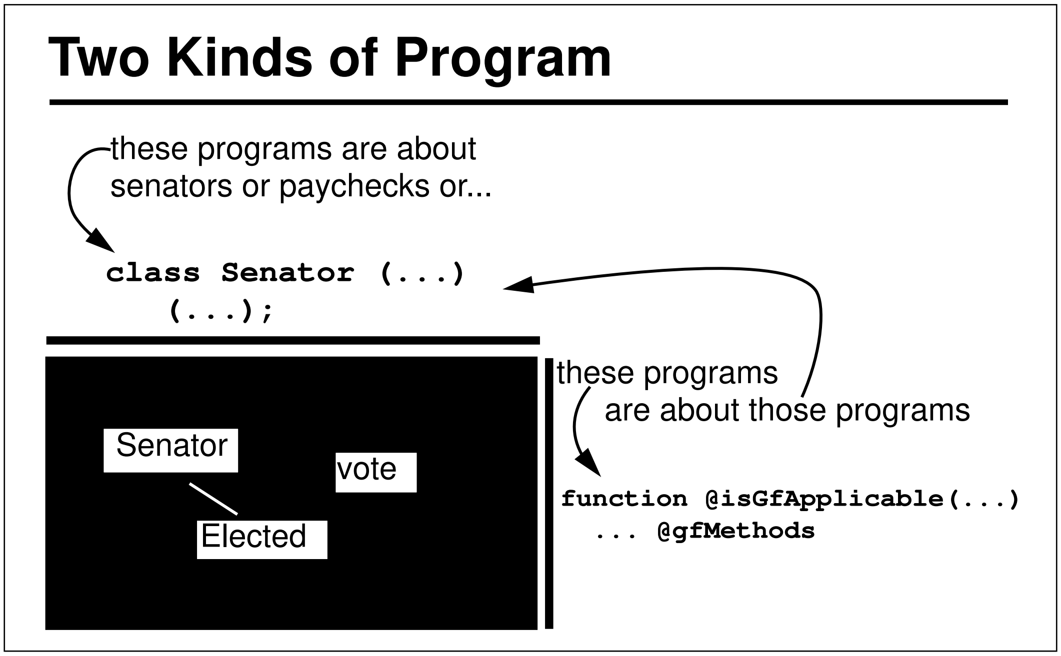
Names of functions (and later classes and GFs) at the metalevel begin with an ‘`@`’ sign.

The names of meta-level functions, both built-in interface functions and programmer-defined functions, are prefixed with an ‘`@`.’



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 programmers are writing code for, 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.

This terminology now makes it possible to clarify and refine the design process of page 52. The substrate designer designs the substrate, a base interface and a meta interface to it. The programmer normally programs to the base interface. Often, this will be all that is needed. Sometimes, however, he will want to solve a problem that requires information about the substrate that is not obtainable through the base interface. He, or one of his more sophisticated colleagues will therefore turn to the meta interface to see whether it lets him get to the information he needs. If so, he writes an often small meta program he can then use. In this example from the domain of object systems, **`@isGfApplicable`** or **`@slotGenealogy`** were examples of this. In a way, the programmer thereby enriches the base interface to his substrate to make it more suitable to his needs.

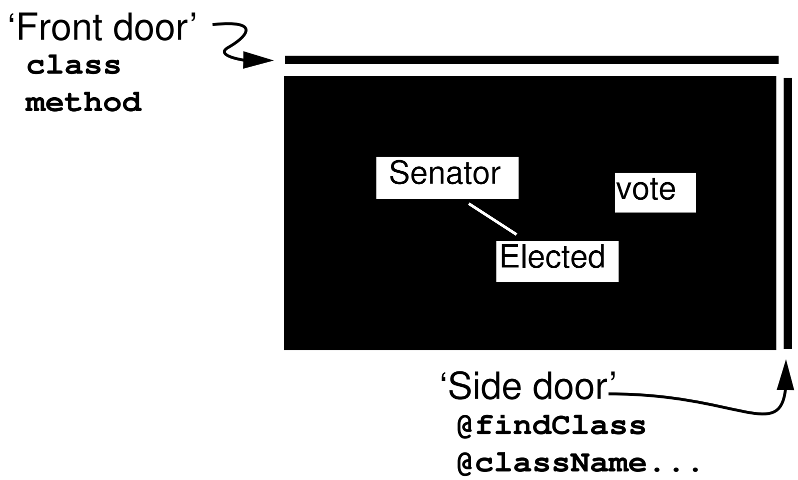


To draw out the difference between the old and new functionalities, it is important to focus on the programs that use each one.

Programs written using the original TinyObjects interface are concerned with senators and voting and the like. (Other programs might be about paychecks or switching systems or ...).

On the other hand, programs written using the second interface are concerned with *other programs*, in particular the programs written using the first interface. Another way of seeing this is that the objects manipulated by the first kind of program represent senators, or whatever the primary subject matter might be, whereas the objects manipulated by the second kind of program represent aspects of the supporting substrate. In the case of this tutorial, that substrate is the TinyObjects object system.

Reification of Internal State



The original TinyObjects system was simple, clean and easy to understand, and it is desirable to preserve those properties. First, notice that the TinyObjects box is still mostly black, and that the original interface has not been changed at all. Any old TinyObjects program will still run. All the object system designer has done is to *reify* some of the box's 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. Which 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 object model that could not be manipulated, the programmer can now grab a hold of their representations as concrete objects.

An Abstraction of the Implementation

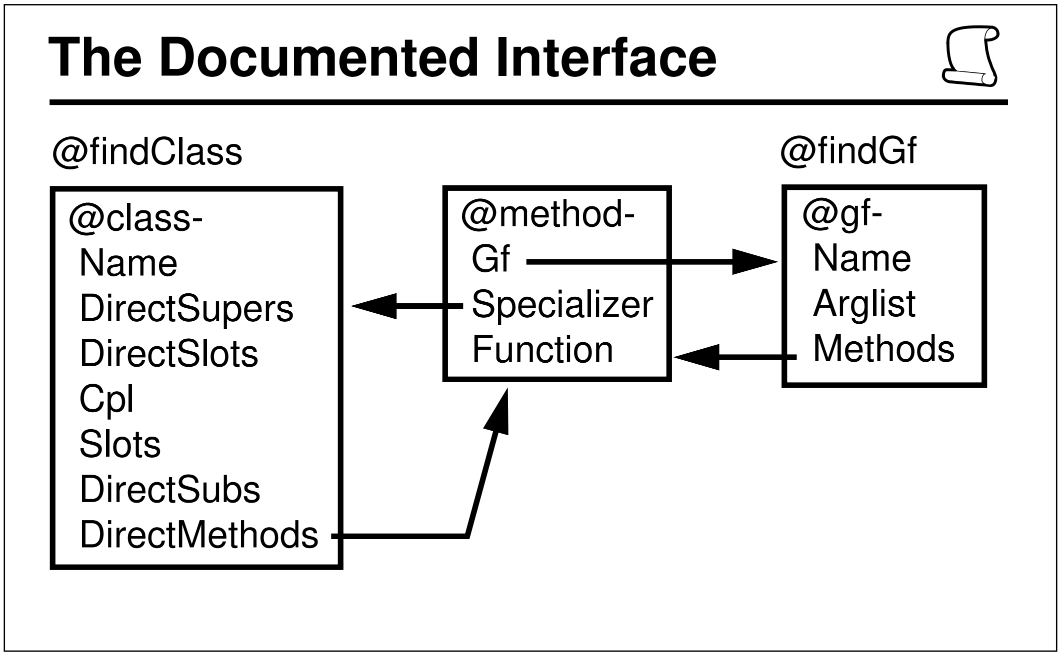
- The object system designer defined an **abstraction** of the relevant substrate state, providing the programmer with a clean representation of his programs.
- It allows the programmer to inspect selected aspects of his programs in useful ways.

All that has been specified is that a set of objects and associated readers exist. Nothing more…

…yet.

In short, the substrate designer solved the programmer's problem by providing him with an abstract representation of his programs. The only requirement this solution places on implementors is that they provide a set of objects describing the relevant program components. The programmer is required to use readers to access these description objects, so proper insulation between application programs and the substrate is always preserved, no matter how the implementors decide to realize those objects.

Here is how all this relates to the traditional black-box model.

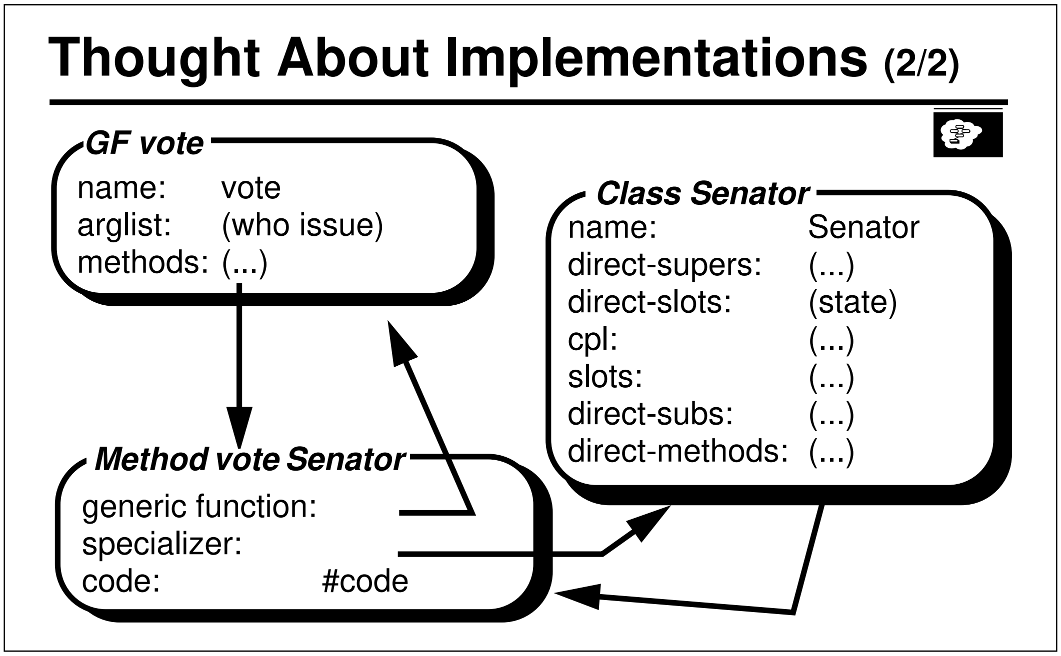


Instead, what was called for was a well-designed interface to the relevant pieces of substrate state.

So, the object system designer documented a general purpose interface that any TinyObjects implementation can support. She took advantage of the encapsulation properties of object-oriented programming to help achieve good modularity in the design.

This figure summarizes the extended interface she developed so far.

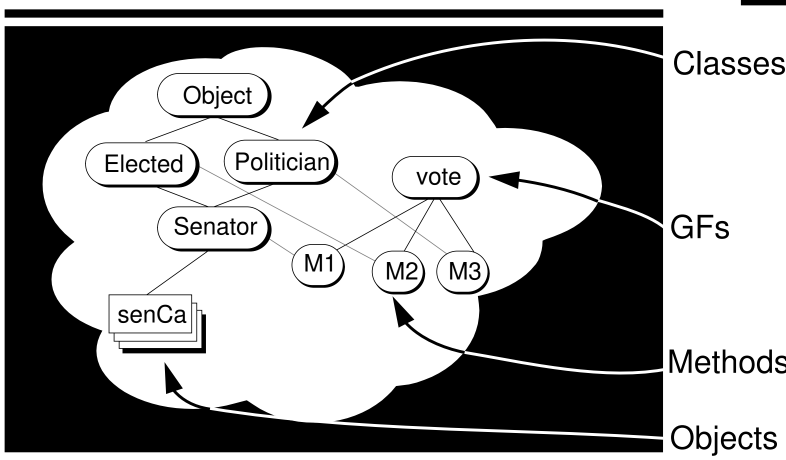
At each step along the way, she was careful to test the new functionality, to make sure that it properly addressed the problems the programmer was trying to solve.



...Which in more detail looks something like this.

While this internal representation was pretty much what the programmer was asking for, it was very important that the object system designer did not require the implementors to provide arbitrary access to their data structures, nor did she document idiosyncrasies of a particular implementation.

Thought About Implementations (1/2)



In response, the object system designer analyzed the questions to understand which set of similar questions they were instances of. Armed with this more general understanding, the object system designer thought about the information inherently maintained in all TinyObjects implementations and realized that a representation of programs something like the one shown here must be maintained in all of them.

Review

In response to programmer requests like:

- How can I find out how much I am using multiple inheritance?
- How do I track down where a given slot gets defined?
- How can I tell whether a given GF is applicable to objects of a given class?

...the object system designer wanted to give the programmer access to the implementation's representation of programs.

The programmer approached the implementors of his object system or other substrate with these kinds of requests. They are all about the programmer wanting to look at the structure of his programs, maybe trying to write little programs to explore those structures.

Anything Else to Try?

Again, it is clear that these accessors can be used to write a number of similar programs.

> Exercise: Write 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?

These examples illustrate how an open implementation could, for instance, be used to write special-purpose tools for displaying or analyzing characteristics of applications. Generic browsers can usually not be tailored sufficiently for this task. Note the additional important advantage of portability which is provided by the fact that the interactions with the implementation occur through a standard abstraction.

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;}
```

Go through given GF's methods. Collect all classes to which any of the methods are specialized and which are either the given class or one of its subclasses.

This is very nearly the same code. The difference is that it collects all the classes that provide a method for the generic function instead of just returning **`true`** as soon as it finds the first class that provides such a method.

> Exercise: One problem with this implementation of **`@methodGenealogy`** is that the order 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 (see "Methods Link the World" on page 82).

GF Applicability

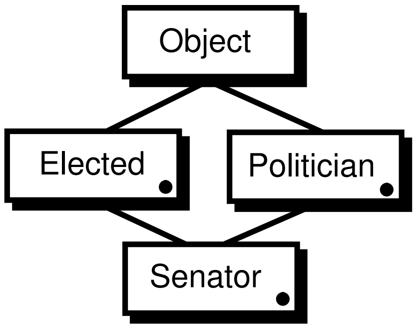
```
function @isGfApplicable (gfName, className) {
  var gf = @findGf(gfName);
  var c = @findClass(className);
  foreach (m, @gfMethods(gf)) {
    var spec = @methodSpecializer(m);
    if (@isSubclass(c,spec))
      return true;}
  return false;}

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

The code above implements this first version of the applicability test. It goes through the given generic function's methods. If one is found that is specialized to the given class or to one of its subclasses, the function returns **`true`** and terminates. The helper function **`@isSubclass`** tests 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`** in this case. Write such a function, called **`@isStrictSubclass`**.

Introspection2.frm

Is Given GF Applicable?




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

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

• denotes **`vote`** Methods

Also just as with class accessors, it is now time to test the new functionality to ensure that it meets the programmer's needs. One problem the programmer had was that he wanted to know when a particular 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 this and another kind of question the programmer might want to ask.

In thinking about how a programmer can now implement these functions, it is important to think only in terms of the documented class, generic function and method description interfaces.

One implementation approach for **`@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 a superclass of the class, or the class itself.

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

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

Also: **`@classDirectMethods`**

Here, then, is the new piece of interface which all object system implementations will be required to provide.

Just as with the class accessors on page 64, to save space it will simply be said here that these accessors return the 'natural' values, e.g. that **`@gfName`** returns a string, and **`@gfMethods`** returns a list of method description objects.

Remember that the protocol designer has so far provided only information that is inherently present in every implementation. She has not asked for anything that did not already have to exist in all implementations. For full-scale systems, this assumption is sometimes not trivial to check. Implementors use a variety of optimization techniques that sometimes require some hard thinking about what really is maintained in all implementations. The protocol designer needs to ensure that her demands do not preclude these techniques, and it will be shown later how she can go about this.

Use the Same Strategy as for Classes

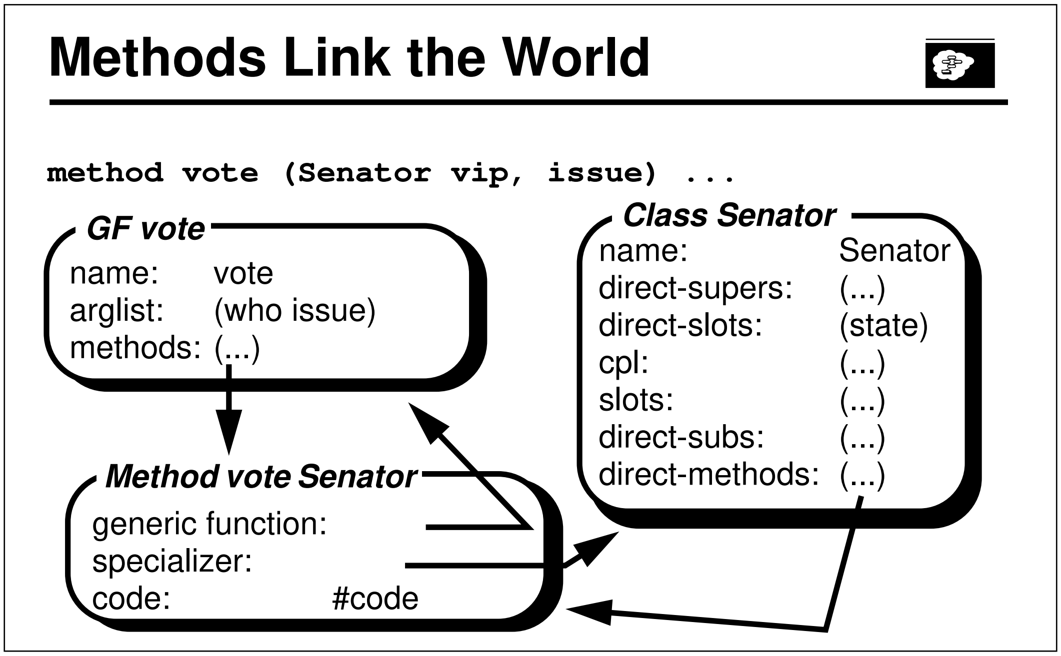
Since information about GF's and methods is inherently managed by all implementations, the object system designer can use the encapsulation properties of OOP to:

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.

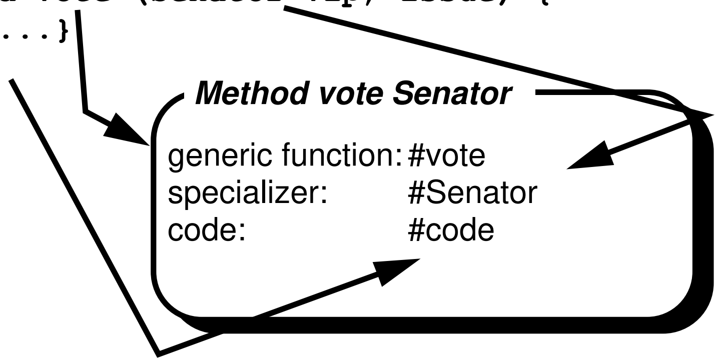
The same approach as for classes will be used to address these new problems.



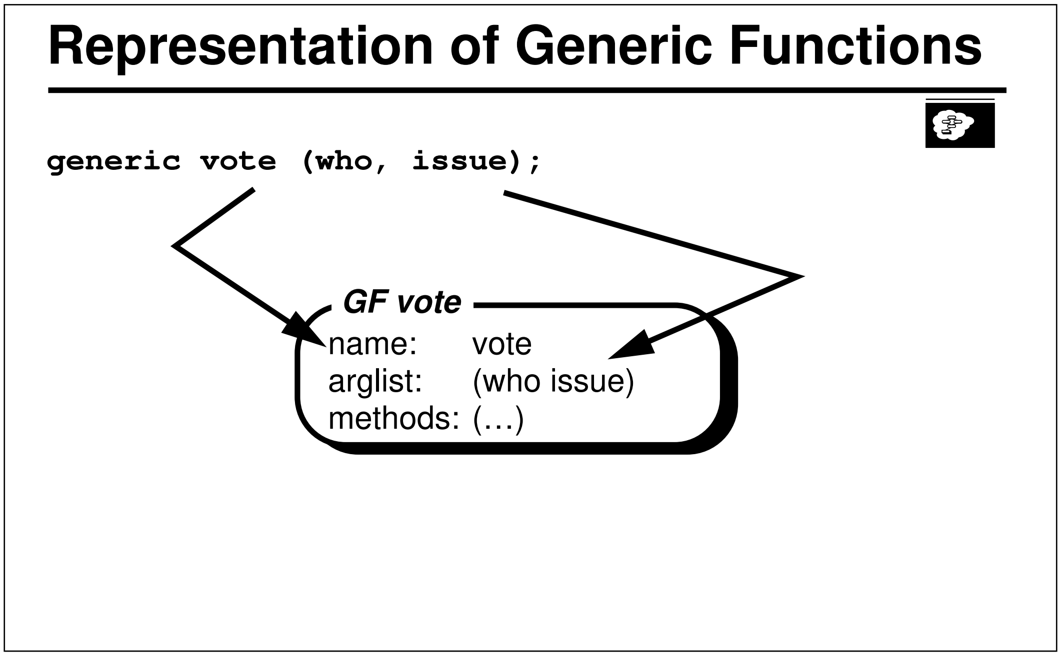
Notice that a method is linked to both a generic function and a class. In that way, the classes, methods and generic functions are linked in the way to be expected in object systems.

Representation of Methods

```
method vote (Senator vip, issue) {
   if ...}
```



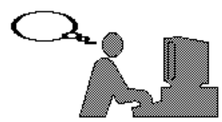
And the same will be true for methods.



The next step, again, is to understand what kind of relevant information every implementation of TinyObjects will inherently maintain. The black box marker on this slide indicates that once again, the object system designer is “speculating with conviction” about some inherent characteristics of all TinyObjects implementations. It is reasonable to expect that each implementation will keep descriptions of generic functions that are similar to those kept for classes, as displayed above.

I Want to Know…

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



Here is another programmer request. It will be used to demonstrate that providing an analogous interface for generic functions and methods can solve additional problems.

The interface to information about classes as defined so far is insufficient for a solution to the request above. So the application programmer cannot solve it independently, and responsibility falls on the object system designer to enrich that interface. Note again that this enrichment, to be fully justified, should solve a whole class of similar problems.

Stepping now into the substrate designer's shoes, the first step is to move back from the details of the problem at hand to see what the programmer really wants. In this case it is straightforward, he wants information about the generic functions and methods defined in his program.

Recap

A clean abstraction of the data inherently maintained in the state of all implementations allows the programmer powerful access to information about his program.

If the abstraction is indeed clean, it is not too much to ask implementors to support it.

Stepping back and viewing the TinyObjects object system as just one kind of substrate again, one can summarize as follows: the programmer needed some substrate (i.e. object-system)-'internal' information that was inaccessible. This information was in some way or other inherently being maintained by every implementation of that substrate. The substrate designer therefore decided that it was not too much to ask that all implementations provide access to it. But this access was provided via an *abstraction*, not via direct access to the data structures used by individual implementations. The substrate designer therefore developed a separate, well-encapsulated (object-oriented) interface to the information and documented it for the substrate implementors to realize, and for the programmer to use.

Anything Else to Try?

Having seen these simple examples, it should now be getting clear what kinds of programs can be written with the new functionality. Some other examples include:

- 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?

An important property of this new functionality 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, it is important to note that 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.

In addition, because they are a documented part of TinyObjects, programs that use the new facilities will be portable.

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 facility that returns the names of *all* the ancestors that define this slot name.

> Question: How do you decide whether to make a function accept a class name, vs. a class description object?
>
> Answer: In general, it will be convenient to have functions used interactively 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 74. Edit the program you wrote for that exercise to only report diamonds in which the class at the top of the diamond defines slots.

Slot Genealogy (1/2)

Where did slots in a class come from?

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

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

One of the QA 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.

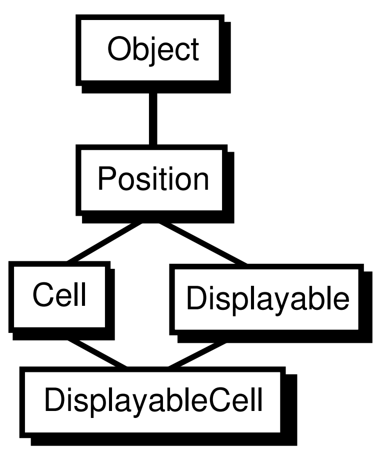
How Much Multiple Inheritance?

Which classes have multiple direct supers?

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

Here is another example of a program the quality assurance team might want to write: it gathers a list of all the 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, and return a list of leaf/root pairs.



Other Operations on Lists

```
var l = list("A", "B" "C")

first(l)  -->  "A"

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

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

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

All remaining list utilities uses in this book are listed above. Their functionalities are clear from their names and examples usages.

Collecting Results of Ops over Lists

`extend` tacks an item on 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, such as **`("N" "S")`**. As explained earlier, the **`extend(list,element)`** construct tags **`element`** onto the end of **`list`**. **`Union(list, sublist)`**, on the other hand, will append the new **`sublist`** to **`list`**.

Finding All Subclasses (3/3)

A First Use of Class Readers

```
@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 can 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.

Here are a couple of additional list manipulation functions we use.

Roadsigns to Remember

This little mop means that the material on the slide demonstrates the use of the Metaobject Protocol abstraction over TinyObjects implementations.

inherently maintained material	***documented MOP interface***	***use of the MOP interface***

Note the roadsign on the previous slide. The small mop means that the code on the slide is a program using the new MOP functionality. This code could be written by application programmers. In contrast, the decision of which information to make accessible to the outside, in which shape or form and under which conditions is reserved to MOP designers.

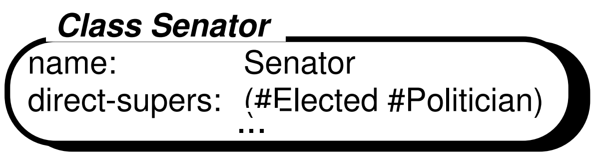
Be sure to remember that such code can only use all the regular facilities that application code can use, plus the documented functionality on slides marked with a scroll. It cannot assume any details that might appear on a slide with a black box marker.

Introspection1.frm

Many Views of a Class

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

Senator



#Senator

Note that over the past few pages, several graphical representations of classes have been used, depending on what aspects of the class was important in the context. All of those appear on this slide.

The first is the definition of a class as it would appear in the source text of a program. The second is the name of a class. The third is a visualization of the data which implementations would maintain about that class in their state, after the compiler/interpreter has processed the class definition. The fourth represents the documented structure programmers can access to acquire information about their program. This is shown as a box with square corners without any suggestion of possible internal information. The last, finally, is a concise way of referring to a data structure like the third representation.

All of these will show up again throughout the book, and shortly similar notations will be used for generic functions and methods.

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.

Here is the code that the programmer would write, using the new functionality, to implement the tree walk. Mostly, it looks like any other tree walking program. It recursively walks down the class graph, collecting each node as it goes. In the interest of brevity, code that removes duplicates from the returned list of subclasses has been left out. **`Senator`** will therefore appear twice in the result of calling **`@allSubs`** with the class **`Object`**.

This slide shows several of the built-in list operators that will be used: the statement **`var result=list()`** introduces a variable called **`result`** which is initialized to contain an empty list. The statement **`extend(result,c)`** adds the class description object **`c`** to the end of the list **`result`**.

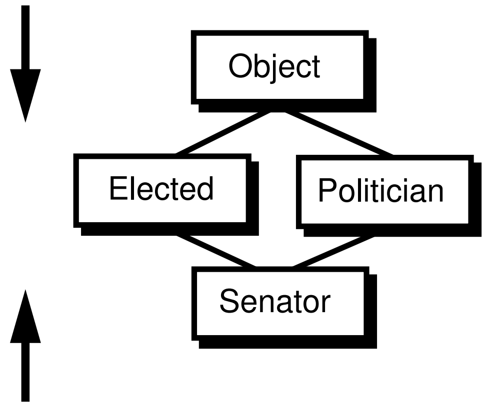
The **`foreach`** assigns successive elements of the list returned by **`@classDirectSubs`** to the variable **`sub`** and executes its body. When the list is exhausted, the loop terminates.

Finding All Subclasses (1/3)

Testing the new facilities

`@classDirectSubs`
returns down links

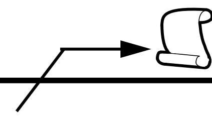
`@classDirectSupers`
returns up links



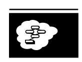
The fictitious TinyObjects implementors have added the accessors on page 64 to enable the programmer to solve the kinds of problems on page 54. Now it is time to go back and test the new functionality, to make sure it works as intended.

The first programmer question had to do with 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. Putting the programmer's hat on now, this figure shows the available interface and functionality. The complete list of subclasses of **`Object`** can be found by walking down the tree, collecting each node, calling **`@classDirectSubs`** to get to the next level down.

Roadsigns to Remember



This scroll means that the material on the slide is a part of the Metaobject Protocol specification that defines an abstraction of TinyObjects implementations.



inherently maintained material

documented MOP interface

So far, two kinds of roadsigns on have been used on slides. The first, a black box with a cloud inside, indicates that the material on the slide is representative of the inherent information in a TinyObjects implementation.

The second, a scroll, indicates that the material on the slide is a documented extension to the original TinyObjects functionality. It is part of the *metaobject protocol interface* (MOP) that is being developed. (The term metaobject protocol still has not been defined, but that will come shortly.)

It is very important to distinguish these two kinds of slides. It is wrong, for example, to let slides with a black box and cloud lead to the assumtion that a documented[1] class description object will have a slot named **`directSlots`**. Slides with a scroll are all the programmer can count on. The one on page 64, for instance, says that the only way to get the direct slots of a class is to call **`@classDirectSlots`**.

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

- Not how this information about classes is really represented in any particular implementation, or
- whether these externally visible class description objects share structure with any internal structures of the implementation.
- All that has been said is that these functions can be called and the natural values will be returned.

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(senCal)`**

Readers:

```
@className            @classCpl
@classDirectSupers    @classSlots
@classDirectSlots
@classDirectSubs
```

The result of this design process is a decision to extend the TinyObjects specification. It now requires that every implementation must make available an object for each class which will provide access to information about that class.

Note that all the accessors begin with an '**`@`**'. This is simply a naming convention. More will be said about it soon. To save space, the return values of these accessors are not specified in detail here, only that 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 a bit later.

This extension to the TinyObjects specification means that the programmer can write programs to solve the problems considered in this example. But, it is important to note that the designer has not changed the basic TinyObjects model. She has simply added an auxiliary interface that allows the programmer to get information about the classes his programs define. In considering the functionality that has been added, it is very important to remember some things that have *not* been specified:

Strategy

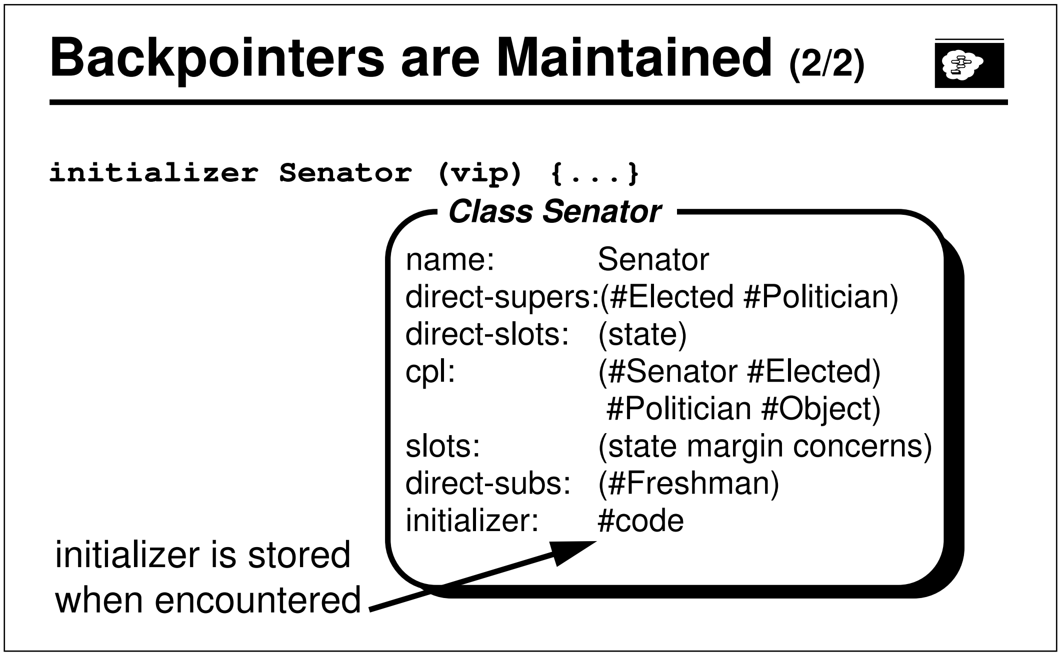
Since information about classes and slots is <u>inherently</u> maintained by all implementations, object system designers can:

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.

So far, two steps of the design cycle have been completed: (i) the object system designer now has a good sense of what additional functionality the programmer needs to address his problems. (ii) With an eye towards providing this functionality, the designer has thought carefully about what all TinyObjects implementations have in common that is relevant to these problems.

In this case, she can conclude relatively easily that because the desired information about programs is inherently part of any implementation, and because it would be useful for the programmer to be able to read it back, it is worth providing access to it—so long as it can be done in a way that is acceptable to implementors. At this point the object system designer need to design an interface that is clean and elegant, and that provides appropriate insulation between programmer and the implementation. The plan is to use the encapsulation properties of object-oriented programming to achieve this.



Similarly, when the initializer for a class is encountered, the class's description is updated to hold an appropriate code pointer.

Backpointers are Maintained (1/2)

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

Class Senator

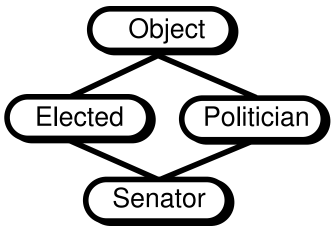
name:	Senator
direct-supers:	(#Elected #Politician)
direct-slots:	(state)
cpl:	(#Senator #Elected) #Politician #Object)
slots:	(state margin concerns)
direct-subs:	(#Freshman)

Subclasses are
noted as encountered

When a TinyObjects implementation encounters a class definition, it not only builds a description of the class itself, as shown previously, but it also updates the descriptions of the new class's superclasses to point down to the description of the new class.

> 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. If this concerns you, be sure to consider it when you do the exercise on page 109.

Derived Information is Also Stored



Class Senator

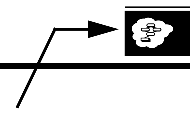
name:	Senator
direct-supers:	(#Elected #Politician)
direct-slots:	(state)
cpl:	(#Senator #Elected) #Politician #Object)
slots:	(state margin concerns)

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

In a TinyObjects implementation such class descriptions will also contain various kinds of computed 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 45, if you wish to review CPLs).
- The slots entry is a list of all the class's slots, those defined directly in the class, as well as those inherited from superclasses.

Roadsigns to Remember

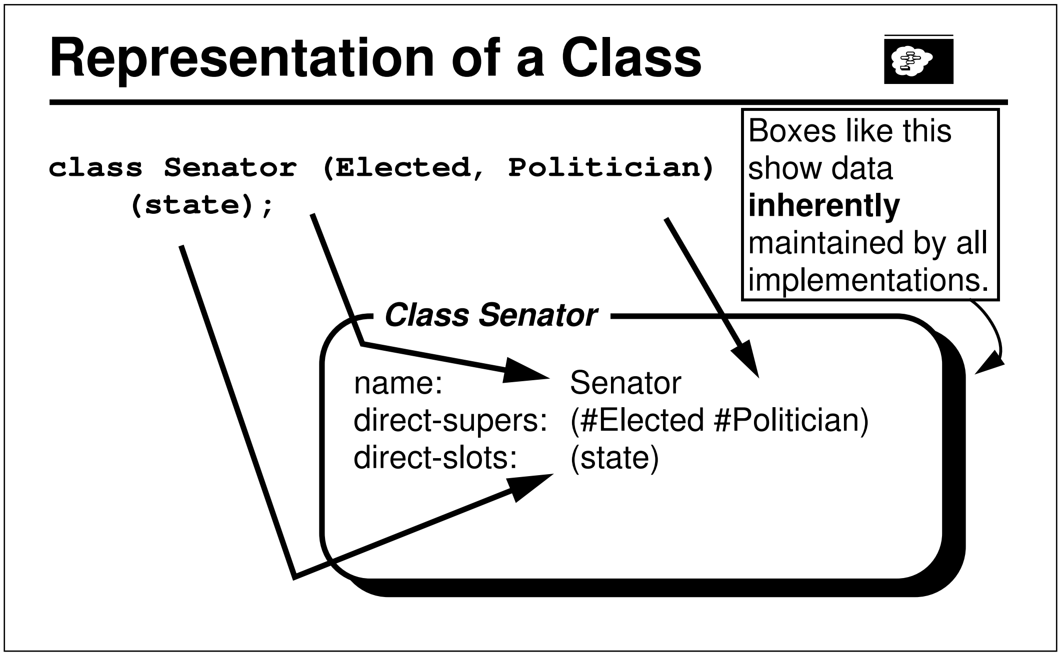


This black box with a cloud and some information inside means that the material on the slide shows information that is inherently maintained in some form in all TinyObjects implementations.

There will be two similarly-placed markers whose meaning will be introduced later.

inherently maintained material

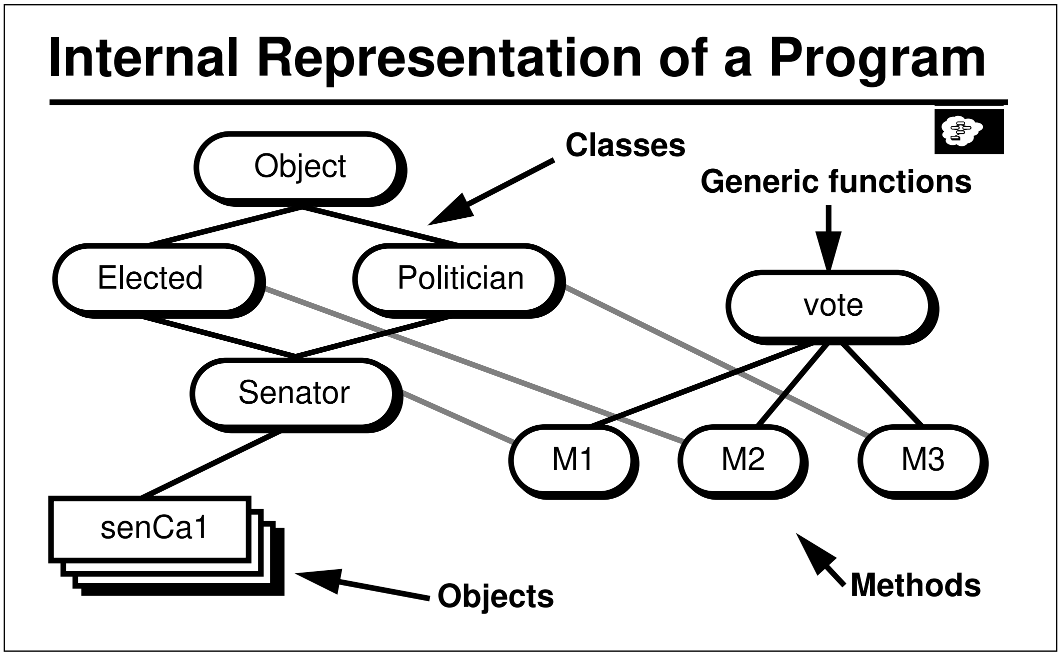
Slides that show information inherently maintained within a substrate implementation's state will be marked by a black box that is partially made transparent by a cloud with a diagram inside. The cloud is to remind the reader that the slides with this marker do not show the actual implementation of a substrate. They merely show what all implementations of that substrate have to be maintaining in some form or other. In the previous figures, this included entities representing classes, generic functions and methods, and their interconnection.



Here is a more detailed look at some of the information in the previous figure. Boxes with rounded corners show an elaboration of the kinds of boxes on page 57. This one shows information about a class, which any implementation would have to maintain in some form. Some implementations might use an object to do this, others might use some other kind of device.

Whatever form it takes, inside each implementation there will be information describing each class, including information taken from the class definitions. Examples are the class names, their superclasses and the slots specified explicitly for each class (rather than the ones inherited from superclasses).

The notation "#xxx" (sharp followed by anything) is a compact visual way of indicating a structure passed around by an implementation.



This figure shows one of those clouds. It "speculates with conviction" about which data relevant to the problem at hand would inherently have to be maintained in any implementation of TinyObjects. Again, this slide is not showing the details of any particular implementation, instead it shows a representation of programs that any TinyObjects implementation—interpreted or compiled— inherently has to maintain in some form[1].

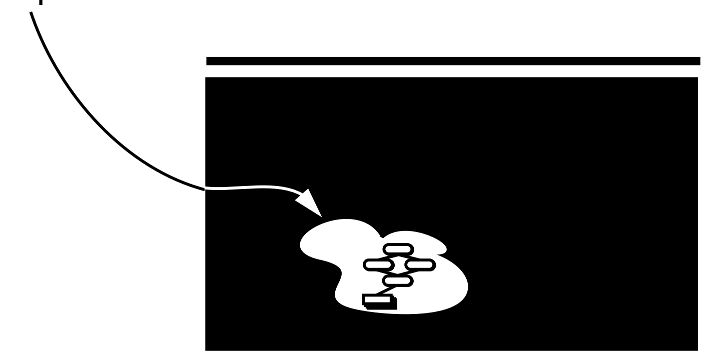
For people who are not object system designers or implementors, this slide may be a bit surprising. But object systems are not magic. In the same way that a window system has windows, or an accounting system has accounts, a TinyObjects implementation will have representations of classes, methods and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

This is the sort of data the programmer is asking to access, and it is easy to imagine other programmers wanting this information as well.

1. `Object` is a built-in TinyObjects class which by default is at the root of every inheritance tree. It is therefore the root of the inheritance tree of all TinyObjects programs.

Inherent In Every Implementation

Looking for information that is inherently present in all implementations



The way to proceed in this situation is to think about which of the information needed by the programmer would have to be present in all implementations of TinyObjects. This is very different from opening the black box of any particular implementation! The cloud with information in it is intended to convey that no actual data structures of any one implementation is being exposed or examined.

In the current example, information about the structure of programs is needed. Here is a visualization of this information as it would be maintained somehow in every implementation of TinyObjects.

What is this Programmer Asking For?

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

Essentially, a representation of the inheritance structure in their program.

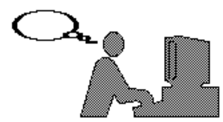
The second step in working with this programmer request is to back away a bit from the details to develop a more general sense of what is being asked for. In this case, it is relatively clear, the request is about access to the basic inheritance structure of programs.

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 about essentials is sometimes murky. It can take quite some time to get to a clear understanding of what a programmer needs.

Dear TinyObjects Developers,

A TinyObjects programmer writes:

> For my new QA program I need to write programs that have access to 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?



Remember the stance of giving the programmer access to enough power that he can solve his own problems. Object system implementors should not answer:

- "Ask your vendor for a browser"
- "You don't really need that…"
- "Use awk or grep or…".

Rather, it is time to enter the second step in the loop...

ing the newly accessible functionality can be addressed by the programmer, without further involvement of the substrate designer or implementors.

Flow of the Following Material

Successive examples of the programmer encountering problems working with TinyObjects' object model; the response to which will be a progressive opening of the model's implementation.

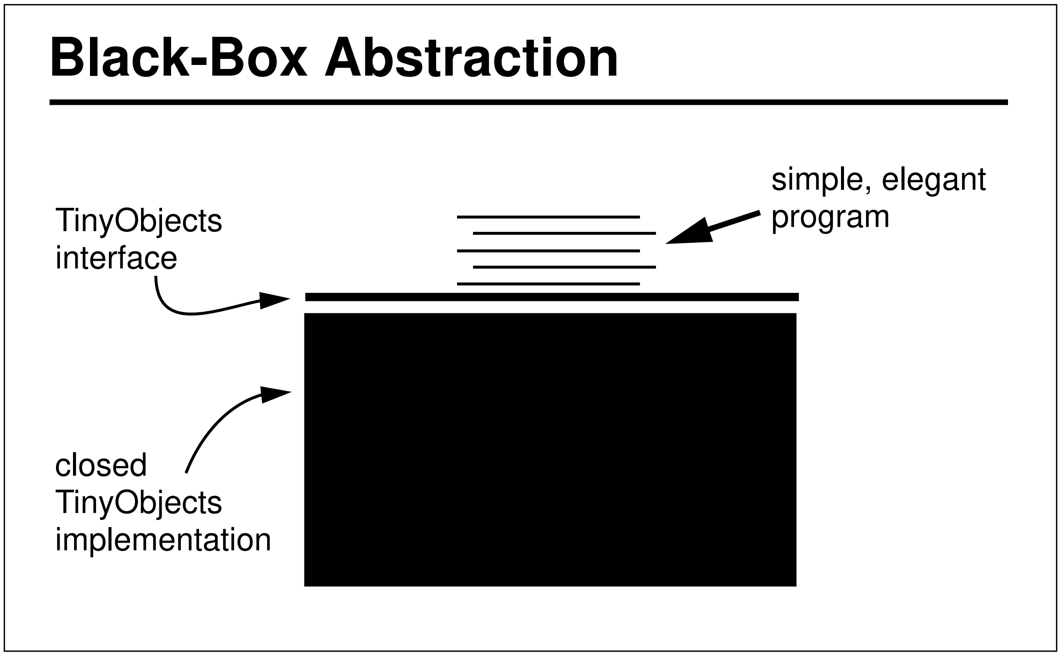
The actual problems used are made up, but they are representative of real problems seen with CLOS and other object-oriented systems.

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

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

- The programmer complains to the substrate designer about the inability to accomplish a task using the substrate.
- That designer analyzes what the programmer is really asking for in more general terms.
- The designer identifies the information and mechanisms inherently present in all implementations of the substrate that, if accessible to the programmer, would allow him to extend the substrate by himself to fill his needs.
- The designer develops a clean abstraction to such information and mechanisms, thereby opening the implementation as needed—in a disciplined fashion. She ensures coherence of the resulting extended system, and the existence of efficient implementation algorithms.
- The designer tests the abstraction and, finally,
- Substrate implementors ensure that their implementations support it.

If done right, at the end of this cycle, a whole class of shortcomings requir-



TinyObjects is, like most of its cousins, a 'black-box' design. It presents the programmer with an 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. The programmers writes his code on top of this interface, with no knowledge of the implementation below him.

This chapter will show that the totality of information hiding enforced by black boxes is not always an advantage. It will also show how the notion of black-box abstraction can be expanded to allow some useful 'looking inside', without compromising the safety and portability goals of the approach.

Where the Programmer Stands

TinyObjects is a traditional, closed system. The application programmer can use it as a 'black box,' building on top of it in the traditional way.

But if he wants something more, or something different…

… Too Bad

TinyObjects is simple and traditional. But the programmer is going to have problems with it.

Chapter 3 But I Wish I Knew…

This chapter will show how one class of common problems can be solved by giving programmers *introspective* access to the system they are building on top of. Introspective access makes it possible to examine the state of the system in principled ways, but without being allowed to make any changes.

This will be a first complete cut into open implementation design, establishing many of the principles of this approach.

Voting (3/3)

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

"It is crucial for the country's competitiveness
that the crane industry be strong. So high-rise
restrictions are bad for the country."
```

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 the TinyObjects model and associated language. This is all the preparation needed to start showing how a metaobject protocol can be used to open the implementation of a substrate like this, thereby making it more flexible and useful for the programmer.

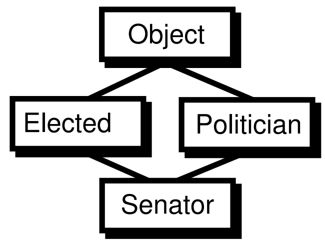
Voting (2/3)

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

"All men are created equal.  Now is the time for
all good men to come to the aid of their party.  A
penny saved..."
```

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 43 just calls **vote** recursively, causing a fillibuster.

Voting (1/3)

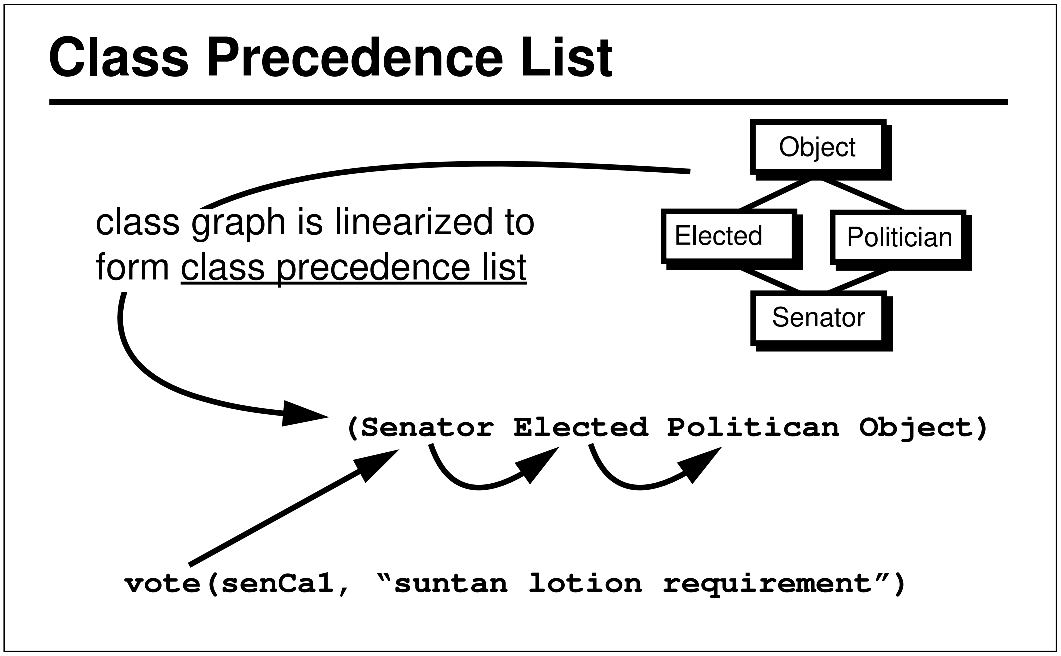
```
vote(senCa1, "suntan lotion requirement")

  --> "NO"
```



`(Senator Elected Politican Object)`

When the generic function call shown above is executed, the first method that runs is the one specialized to **`Senator`** (bottom of page 43). 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 43). When that method invokes **`callNextMethod,`** it runs the method specialized to **`Politician`** (page 34).

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.



This figure introduces the concept of *class precedence list* and shows how it is used by method dispatch and **callNextMethod**. The class precedence list (CPL) of a class is a linearization of its superclasses. (The details of the linearization are not important now, suffice it to say that it is a topological sort based on subclass/superclass relationships and the order that superclasses appear in class definitions.)

The workings of method dispatch and **callNextMethod** can now be explained precisely. 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 example on the next page.

Making Senators

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

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

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

Some Additional `vote` Methods

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

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

Here are two more methods on the generic function **`vote`**, that define special behavior for the classes **`Elected`** and **`Senator`**. These methods are in addition to the two methods previously defined, which were specialized to the classes **`Politician`** (page 34) and **`Manager`** (page 36).

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 which case they enter a fillibuster (the inifinitely recursive call to **`vote`**).

Additional Initialization

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

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

initializer Senator (vip, concerns,
                     margin, state){
  initialize Elected (vip, margin);
  initialize Politician (vip, concerns);
  vip.state = state;}
```

This slide shows the initializers for all three classes defined on the previous page. From a programmer's perspective, these initializers mandate the format of the initargs list when each of the classes is created. So, for example, when creating senators, three initargs need to be passed. The concerns, the margin of victory and the state.

Note that it is the responsibility of the initalizer on 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.

Senatorial Voting Algorithms

```
class Politician () (concerns);

class Elected () (margin);

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

Assume the following readers and writer:

```
concerns       setConcerns
margin
state
```

The politician example will now be elaborated to show a few more details of how method overriding works and how initializers operate together.

First, some additional classes are defined. The class **`Elected`** is a mixin class, not designed to be instantiated on its own. The class **`Senator`** is an instantiable class and, as shown, **`Senator`**s are elected **`Politician`**s.

At the bottom of the figure are four simple state accessing generic functions. 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 not written afterwards.

Using Readers and Writers

```
concerns(mayor) --> ("IRS")

setConcerns(mayor, list("ethics committee"))
```

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

Encapsulation by Style

Reader and Writer Generic Functions and Methods

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


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

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

When programming in TinyObjects, a style is used which does not have clients 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 direcly 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 simple state accessing generic functions and methods will often be omitted and will simply be 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?

Initialization of Objects

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


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


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

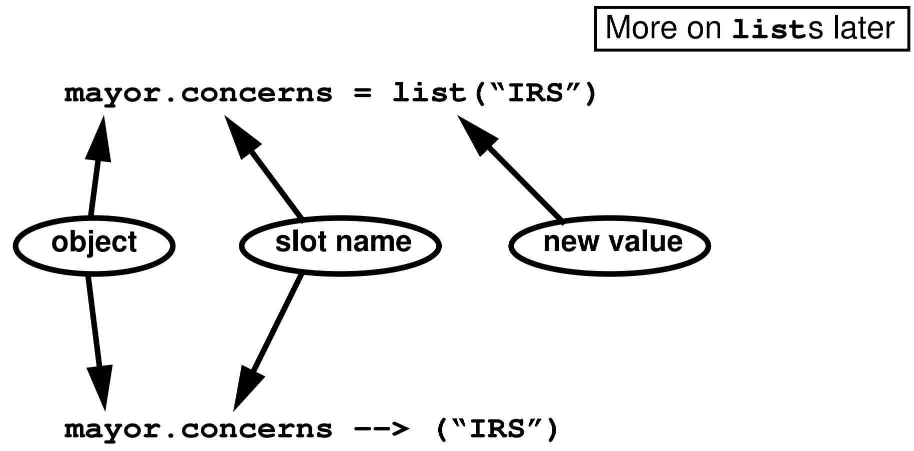
Object initialization is accomplished through a built-in mechanism that allows each class to have its own *initializer*. These are special pieces of code that are called automatically as part of object creation.

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.)

From the perspective of instantiators of the **`Politician`** class, the effect of this initalizer 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 that call to the initializer for **`Object`**.

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, **`Politican`** objects end up with a single slot, **`concerns`**.

Slots are read and written using a dot notation as shown on the slide.

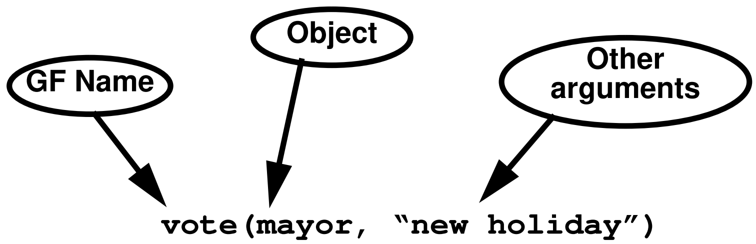
> Question: You've said that TinyObjects is a multiple inheritance model. What happens if two different superclasses define a slot with the same name?
>
> Answer: In this situation, TinyObjects works like CLOS and Smalltalk, not like C++. That is, the resulting objects end up with only a single slot by that name. While this is a simple solution, and often satisfactory, there are cases where it is innappropriate (i.e. the diamond example mentioned on page 74 and page 76). This is clearly something to add to your list of criticisms that an open implementation of TinyObjects should allow programmers to address for themselves.

Polymorphism

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

ceo = new Manager()

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

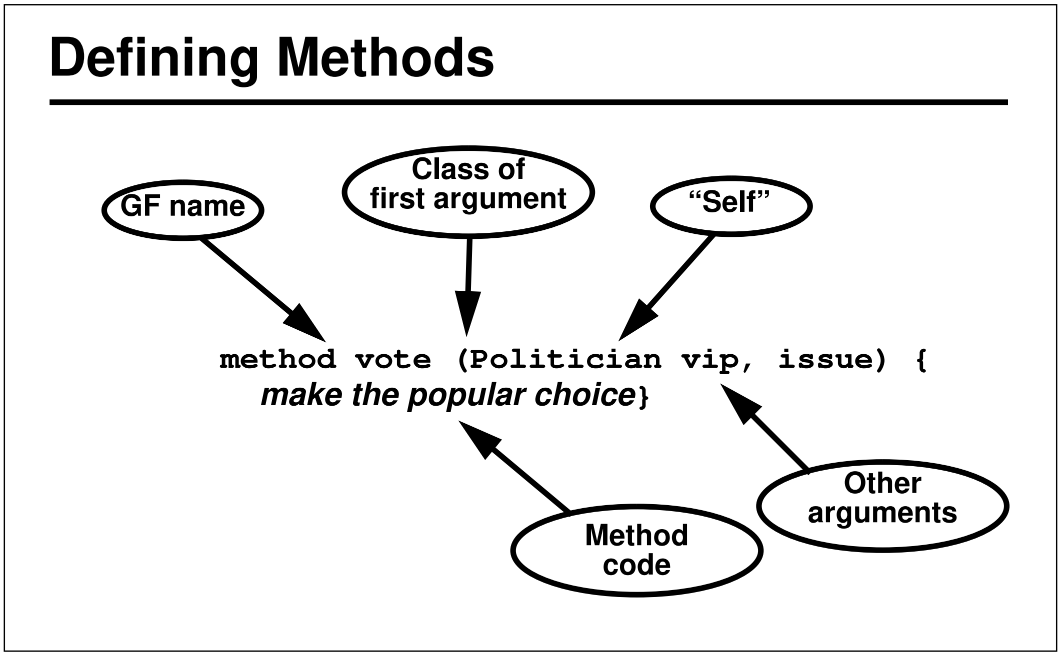
This is an example where more than one method is defined on a single generic function. First, a new class of potential voter, **`Manager`**, is defined. A method on **`vote`**, specialized to **`Manager`** is defined as well. The two calls to **`vote`** show the effect of having the two different methods—politicians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

Calling a Generic Function



Generic functions are called just like regular functions. Method dispatch is driven by the first argument only (i.e. **`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. An example of subclasses overriding methods will come up shortly.



A method definition contains four components: the keyword **`method`**, the name of the generic function the method is to be added to, an argument list and a body of code. The argument list must contain as many arguments as were specified when the generic function was defined. The first argument is special in that it contains the name of 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 first argument appears explicitly in the method parameters, rather than being an implicit "self" argument as it is in some languages. In this case, it allows the programmer to give politicians their own notion of self!

Defining Generic Functions

TinyObjects uses the concept of generic functions, which are similar to C++ virtual 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.

The figure shows a generic function that takes two arguments. Once methods are added, the class of the first argument (the **`person`**) will determine the method that is run.

Making Objects

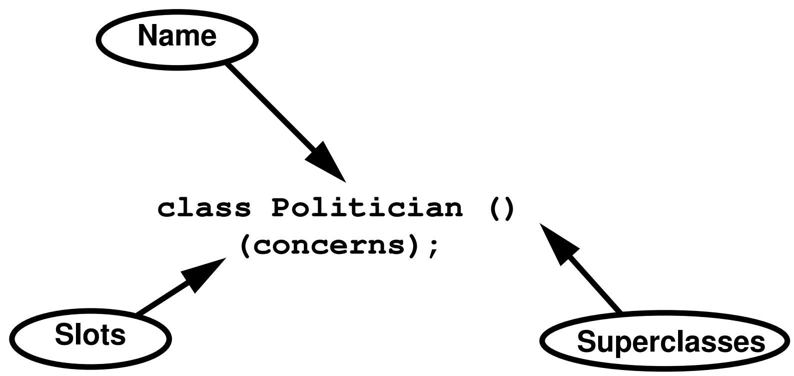

```
mayor = new Politician(...)

  -->  #<Politician 1>
```

Objects are created using the **`new`** primitive. The syntax has 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.

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

As a convention in this book, an arrow "**`-->`**" indicates the return value of a procedure, function or language primitive. TinyObjects itself, when printing out an object, will use a syntax beginning with "**`#<`**" and ending with "**`>`**". So, the second line on the slide above simply says that the result of executing the assignment statement, including the object creation, is to return a **`Politician`** object.

Defining Classes



Now, some examples will introduce the mechanisms and syntax of TinyObjects.

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 does support multiple inheritance—and other examples of defining a class with multiple superclasses will come up shortly.

Throughout this book a naming convention will be used 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.

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, TinyObject's object system is a replacement for C's type system. All variables are bound to objects, and even simple types 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, Tiny Objects has a general scheme for automatically initializing objects as they are created.

Differences From CLOS

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

For those who know CLOS, 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. But, it does have **`callNextMethod`**. So, it has primary methods only, with **`callNextMethod`**, and specialization to the first argument only.

TinyObjects only has instance slots, and does not have any slot initializers, slot initargs or automatically generated slot accessors.

Also, at least initially, TinyObjects is more C-like than CLOS, in that classes, generic functions and methods are not first class.

Differences from Smalltalk

- Multiple inheritance
- Object initialization mechanism is different
- No metaclasses (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. The biggest 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.

Differences from C++

- Much simpler
- Dynamic type checking
- Everything is a pointer
- Dynamic dispatch exclusively (virtuals)
- No public/private
- Name conflicts in inheritance lattice are automatically disambiguated

All values manipulated by TinyObjects programs are dynamically typed, type-safe pointers to objects; this is similar to languages like Lisp and Smalltalk. C++ programs, in contrast, manipulate both objects and non-object values, and rely primarily on static typing.

While C++ distinguishes between virtual member functions (methods) and non-virtuals, TinyObjects only operates with the runtime dispatch of virtuals. Also, TinyObjects uses ordinary function syntax to invoke member functions, while C++ uses a dot notation for member function calls.

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 them only through accessor (client) functions.

Another difference lies in how TinyObjects and C++ resolve conflicts among function and data member names in classes that inherit from each other. While C++ expects programmers to specify explicitly which data or function member they wish to refer to, TinyObjects uses a class graph linearization algorithm to combine them together, or to order their precedence. We will have more to say about this.

TinyObjects

A very simple object model

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

Plus:
a list data structure

This slide presents the features of TinyObjects, in the terminology to be used throughout the book. The next four slides compare TinyObjects with the object models of other principle object-oriented languages. The remainder of this chapter presents the syntax and behavior of specific object model elements.

TinyObjects is 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 objects have (instance variables in Smalltalk, data members in C++).

Generic functions are TinyObject's equivalent of "messages." Methods on generic functions are specialized to classes. Methods specialized to subclasses override those for superclasses, but a method can use **`callNextMethod`** to run the superclass method.

For simplicity, "everything is a pointer" in TinyObjects. This means there is no need to deal with dereferencing pointers or finding object addresses. All parameter passing is by reference, all return values are pointers. Extensive use will be made of a built-in list datastructure similar to that found in some languages and data structure libraries.

TinyObjects.frm

Chapter 2

TinyObjects: An Example Object System

This chapter briefly introduces a working object system, TinyObjects. It is not intended to be a full-scale programming tool, but was instead designed to be a simple "everyman" of object-oriented programming facility, 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.

As TinyObjects is being presented, be a critic. Think about how useful it would be. Is it missing features you think you might need? Do the features it has work the way you would like? Make notes of the problems you notice. These will be useful later in designing, evaluating and improving the open implementation to be developed.

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.

Roadmap

- Start with a simple object system: TinyObjects
- Making it visible
- Making it manipulable
- Making it modifiable
- Recovering performance

Iterative, Example-driven approach

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.

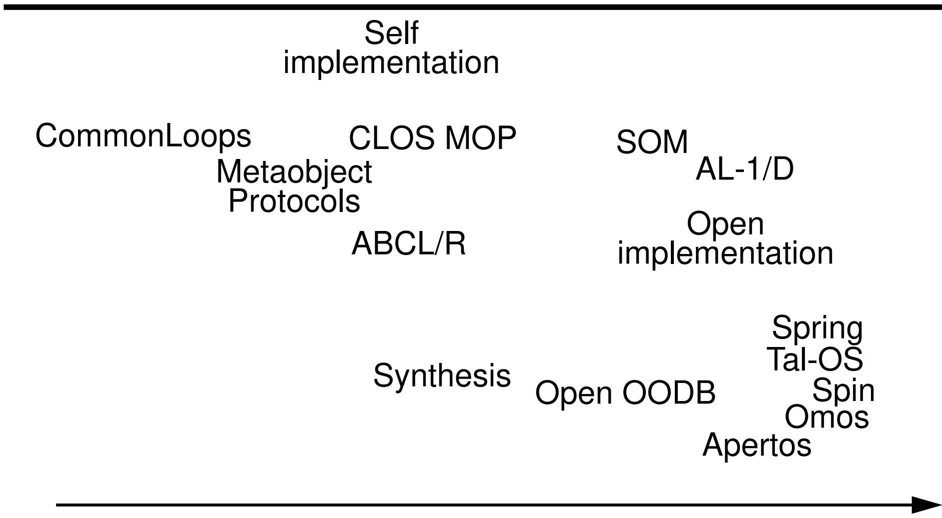
The following chapter presents the design of a traditional (black-box) object system called TinyObjects. It will be the example used throughout the book. Chapter 3 through Chapter 7 work through a series of programmer complaints, gradually opening the implementation to address them.

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, TinyObject 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++.

Some Related Work (2/2)

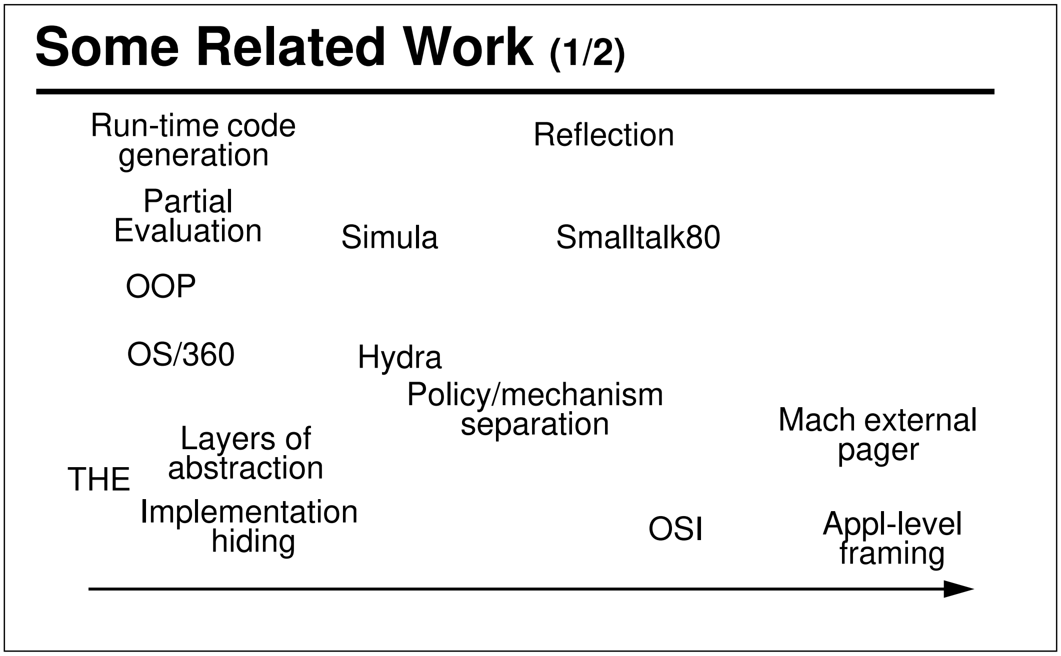


Continuing with the development of languages, metaobject protocols began to be used in *CLOS* to manage the complexities introduced by reflective architectures. The use of reflection, meanwhile, was broadened in *ABCL/R* to cover problems of distribution.

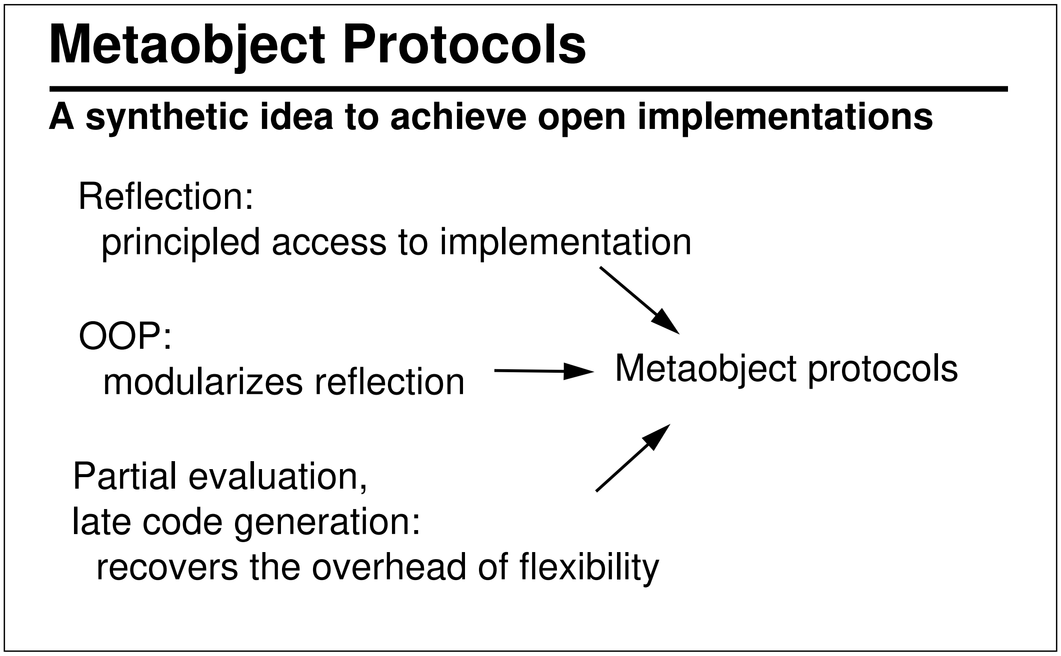
Similarly to the networking community, it became clear that abstraction brought inefficiencies which needed to be offset by advanced implementation techniques that reached back to the ideas of partial evaluation and run-time code generation. The implementation of the *Self* language is notable in this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



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.



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.

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 10. 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 11, or the callback of page 12 are examples for this.

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

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 overconstrained?

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.

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.

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.

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.

Client Programmer Frustration

Functionality
- missing
- wrong

Performance
- space
- time

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.

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.

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 (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 (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.

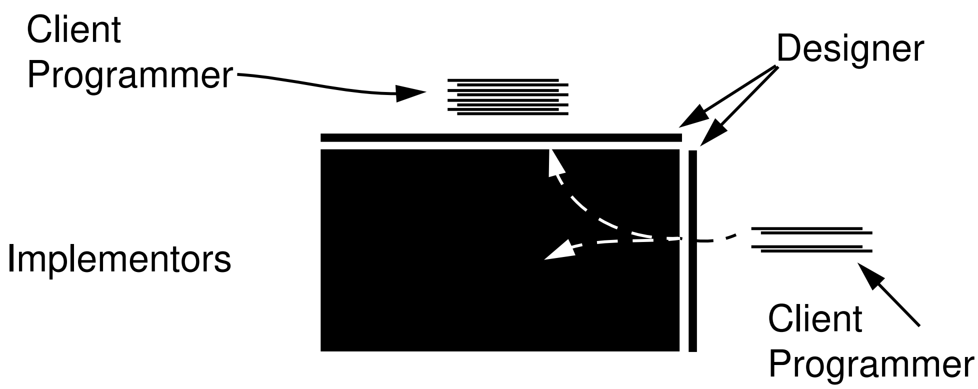
Getting Down To Work

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.

Increased Programmer Access



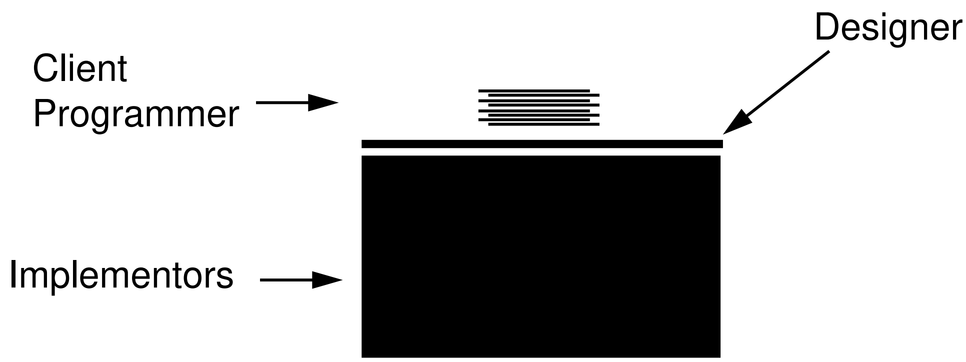
A new kind of interface allows the client programmer to access and modify implementation and design.

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?

Roles and Work Products



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.

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 co-incidental. 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.

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 indispensible 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 (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 worth while is that each substrate tends to support many applications. In addition, the techiques introduced here will be applicable to a wide variety of substrate systems.

-

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?”

The 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.

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.

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*.

1. In this book, the term behavior is used rather than semantics.

Chapter 1 Introduction

Table of Contents

Open Implementations

and Metaobject Protocols

Gregor Kiczales

Andreas Paepcke