Why are black boxes so hard to reuse?[1] I think our field will go through a revolution. We will fundamentally change the way we think about and use abstraction in the engineering of software.

The goal of this talk is to summarize the need for and the basic nature of this abstraction framework.

The change is not new problems or new systems, but a new way of thinking about existing problems and existing systems.

All of this work is a result of a group of us at PARC and elsewhere on these new ideas of abstraction.

I started programming in high school in Basic on a PDP-11. After a couple of years, I went off to hack theater lighting instead.

At engineering school, a crucial idea was that engineers must master complexity, and that abstraction and decomposition are the primary tools for doing so.

After a couple of years of mechanical engineering, I tried computer science again. The ideas were again abstraction and decomposition.

CLOS experience: Systems got more complex, and they got out of hand. The place where “the rubber really hit the road” was when I started working on CLOS.

The opportunity of working on a high-level programming language is like going to heaven. What you’re going to do is make a high-level system for your clients that will insulate them from all the underlying detail.

Not only was I part of the design of CLOS, but I was trying to do an implementation which ran on top of Common Lisp, which was also a very high-level abstraction.

This put me in a very interesting position of being both the customer and provider of my own story.

---

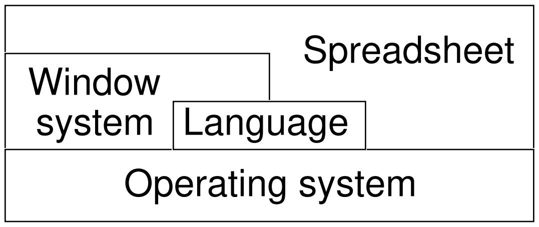
[1]This talk was given by Gregor Kiczales of Xerox PARC at OOPSLA ’94, 10/26/94. © 1994, University Video Communications. A transcript, with point-and-click retrieval of the slides, is available at `http:/www.xerox.com/PARC/spl/eca/oi/gregor-invite/gregor-transcript.html`

You can write an implementation of CLOS on top of Common Lisp in 10–20 pages of code. But mine ended up being 350 pages. In order for it to be fast, it had to be that long.

I want the kind of answers about software complexity to do justice to that kind of problem. I think the words and framework of black-box abstraction that we've been using in the past are part of the origin of that problem.

Spreadsheet example: Suppose you're going to design the display portion of a spreadsheet application.

Functional decomposition:

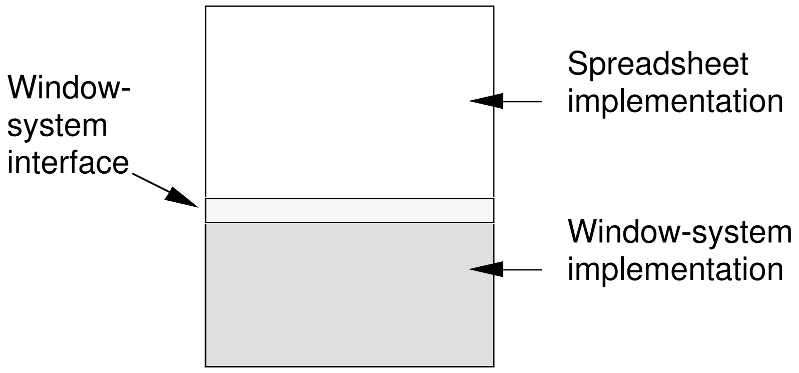


This functional decomposition is so typical that we have standardized many of these components: the window system, programming language, and operating system are standardized.

Black-box abstraction: If we look more closely, there's a clean interface that provides functionality; the implementation is hidden inside a black box.

Inside the window system, it's a horrible and complicated thing. We want to protect clients of the window system from that.

The interface provides useful functionality and "hides implementation details."



The interface is simple. That code, because it's hidden from the complexities of the window system, is very simple.

Many of us take this for granted. But it wasn't always so. In a paper from 1972, "On the criteria to be used in decomposing systems into modules," David Parnas says that every module is characterized by its knowledge of a design decision which it hides from all others. This paper introduced the idea of "hiding."

Everyone knows what a window system is. Everyone also knows what kind of functionality the window-system interface provides. It provides little boxes that you can put on the screen, click in, and type in.

So to build a spreadsheet, you should just put 100 windows on the screen and arrange them. This would be insulated from the details of the window implementation.

Spreadsheet is making massive use of the window-system functionality. That's the good news. The bad new is that it probably won't work. Even though the window-system interface hides the implementation, it comes shining through in the guise of the window system's performance.

Not all of the aspects of the implementation were really details. A number of them were really *mapping dilemmas*, strategy questions that affect performance differentially, depending on client patterns of use. (E.g., sub-window at position absolute screen coordinates.)

In this case, implementor of window system has to decide—

- whether windows should be a heavyweight data structure that memorize a lot of internal values,
- and whether the mouse tracking should be based on a general sense of geometry,

or

- whether windows should be lightweight data structures
- and mouse tracking should try to optimize for regular geometry.

Window implementors almost always choose the former. That's why you can't write the spreadsheet this way.

What happens when you implement a higher-level functionality (e.g., window-system interface) on top of a lower-level functionality is that you take the higher-level functionality and you have to “map it down” onto the lower-level functionality; that’s the “mapping.”

The sense of “dilemma” is that the implementor has a dilemma: if (s)he chooses one way, some clients will be happy, & if he chooses the other way, other clients will be happy.

Mapping dilemmas: Two more terms.

- mapping decision: a decision that an implementor makes about a mapping dilemma.
- mapping conflict: implementor’s decision vs. client preference. Implementor made a mapping decision that is not the one that client prefers, client’s code doesn’t run as desired.

Mapping dilemmas: Key point. The abstraction barrier *locks in* but doesn’t really hide mapping decisions. That can lead to mapping conflicts.

Is this example extreme?

- Maybe it is a little extreme, but the rhetoric of black-box abstraction that we teach in Computer Science 101 doesn’t account for this example.
- It sure is a shame, because a slightly different window system would have allowed itself to be used in this way.
- I have some less extreme examples.
    - Virtual memory. It is a canonical example of this sort of problem. Everyone knows what the abstraction is: Lots of memory, directly accessible. Client can (`malloc`, `read`, `write`, `free`).

      Dilemmas: Page-replacement policy. Most implementors choose LRU replacement, and this is right for most clients.

      But there are some clients, e.g., database programs, that would like more MRU, because the database makes random accesses to part of its memory, while doing sequential accesses to other parts of memory.

    Any program that spreads its working set around is in trouble (also graphics, garbage collectors).

    The graphics application walks through a bunch of data structures, displaying them on the screen.

    If the data structures happen to be laid out in memory wrong, and the prefetching policy of the VM doesn't line up with what the graphics system wanted, it is a serious problem. Most programmers just buy more memory.

  ◦ Example: Programming languages. Procedural abstraction hides a serious mapping dilemma—whether procedures should be called in line or out of line. There are performance-critical cases where this decision can really affect whether the client can use the procedural abstraction.

  ◦ Scientific computing: How to map the arrays?

    On a uniprocessor, how should it be blocked?

    On a parallel processor, how should it be implemented?

    These decisions affect client's ability to use the abstraction of an array.

I don't think this is just an abstract or theoretical problem. I think that examples just like these lead to a great deal of complexity in our current systems. Of the 300 pages of code in CLOS, I think a couple hundred of them are due to these kinds of problems.

Two categories of problems:

- Hematomas of duplication. Everyone knows how you would deal with the spreadsheet case. You would make one big window, and draw a bunch of vertical & horizontal lines. You would duplicate functionality, implement your own window system.

  People have done this often to get the right performance tradeoffs.

- "Coding between the lines": In garbage collector, compiler, graphics, people rewrite code over and over again till their objects are laid out in memory in such a way that the paging system works for them.

Causes code to be contorted. This also happens in garbage-collection systems when you do a great deal of contortion to conform to the virtual memory.

This is particularly a problem in systems that are used by multiple clients. If, e.g., the window system is being used by only one client, it's very easy for it to be properly tuned.

Stressed by multiple clients: But when another client comes along, like a spreadsheet, it wants a different mapping decision. It gets worse as the number of clients increases.

"Multiple clients" is just another word for reuse. These clients are having a hard time reusing the system because of the mapping conflicts. If we are interested in reuse, we should take this seriously.

As an indication of how serious the problem is, one of the major database vendors says that 35% of their products is code to deal with hematomas, the various mapping conflicts, as they try to live on multiple platforms.

Others have noticed these problems before. Mary Shaw and Bill Wulf noticed these problems in 1980. In their paper, "Toward relaxing assumptions in languages and their implementations," (*SIGPlan Notices* 15:3, March 1980) they say,

> "Traditionally, the designers and implementors of programming languages have made a number of decisions about the nature and representation of various language features that the authors feel are unnecessarily pre-emptive."

"Pre-emptive" is their way of expressing the effects of the mapping problem. They are saying that if the implementor makes one decision, and it's not the one that client needs, then the client is pre-empted from being able to use the service the way that they [sic] would like to.

They continue,

> "Both the designer and implementor have only notions of typical use available; they are making the decisions too soon. … Although most people now agree that the use of high-level languages is desirable, the fact remains that many major systems are still written in assembly language."

Of course, that is still true today, although the syntax is worse, and now it's an ANSI standard.

Summary of talk so far: We are engineers, and our chief task is to control complexity.

The basic tools that we use to do that are abstraction and decomposition.

In our discipline to date, we have tried a particular abstraction framework, called black-box abstraction, which attempts to expose functionality and hide implementation.

We've run into problems, because mapping decisions nonetheless show through. That leads to hematomas and coding between the lines in clients of these abstractions.

The fundamental problem is that clients need control over the mapping decisions.

So, one way or another, we've got to find some other approach than hiding the implementation.

What to do about it: Here are five possibilities.

- *Document the mapping decisions.* That won't work. Just knowing that something isn't what you want doesn't make it do what you want.
- *"Here's the sources; have fun!"* I don't think I need to talk much about that plan.
- *Give up.*
- *Avoid mapping dilemmas.* Design systems that are so low level that their implementation presents no mapping dilemmas.
- *Pretend it isn't a problem.* Design systems that have the desired functionality, whether or not they have mapping dilemmas. You might find that to be an odd contrast, but if you turn to a 1974 paper by a very famous person (Niklaus Wirth, Information Processing '74), "On the design of programming languages,"

  > "In fact, I found that a large number of programs perform poorly because of the language's tendency to hide 'what's going on,' with the misguided intention of 'not bothering the programmer with details.'"

  This is a very compelling paper, but basically what it ends up saying is that we should design languages that are so low level that they have no mapping dilemmas.

This is the origin of languages like C. I was a Lisp guy, and when I saw this paper, I was shocked because the Lisp community took the second approach, "Pretend it isn't a problem." They said, We'll put the right functionality into the language, and figure out how to implement it later on. I think the market has chosen about what choice was right for the time being.

Given a lack of understanding of the mapping dilemma, and the lack of a clear solution, the "pretend it isn't a problem" might have been the right approach. But maybe by the end of this talk, we'll have a better solution.

Previous approaches to tackling the problem: In programming languages, there are compiler switches or compiler pragmas or compiler declarations.

- In C, for example, there is an in-line switch for procedures.
- In high-performance Fortran, there is a switch that tells the compiler how to resolve the dilemma of how to lay the array out in memory.
- A number of window systems provide "lightweight windows" that would basically allow the spreadsheet program a way to work

These solutions do address the problem of giving the client program control over the mapping dilemma.

- Procedure implementation.
- Array layout.
- Windowing strategy.

This allows some hematomas and coding between the line to be avoided.

The declarative nature of these mechanisms makes them very reliable, but of limited power, because the client has to choose from among a fixed set of alternatives to control the mapping dilemma.

Let's look at a different solution to the virtual-memory problem that is based to some extent on object-oriented techniques. It started pretty much with work on the Mach external paper, which is presented in this paper by Young et al., which is presented in this work, "The duality of memory and communication in a multi-processor operating system," SOSP '87:

“An important component of the Mach design is the use of memory objects which … allows applications … to participate in decisions regarding secondary-storage management and page replacement.”

They let the client in to control the mapping decisions using some object-oriented techniques.

Virtual memory: In a traditional virtual memory, there is the interface of `malloc`, `read`, and `write`, and below that, everything’s hidden in a black-box implementation.

The implementation maps that interface onto physical memory pages and disk drives.

Inside the box, there is an amorphous hunk of code and data that implement the mapping, and then there is a page table that tells the system for each address where the memory is.

The problem that is being solved is to give the client control over page-replacement policy and mechanism.

One solution would be multiple copies of the virtual-memory kernel, so that some parts of memory would be controlled by one, and some by another.

That solves the problem in a theoretical sense, but it is a pain, because it means that a client has to duplicate the entire implementation to get a different kind of VM.

In Mach, they made each region of memory into an object.

- The protocol provides implementation.
- OOP provides inheritance and overriding.

There is a default class of region-of-memory objects that implements operations like `malloc` and determines which page to flush.

If a client wants a new implementation of virtual memory, it just subclasses that class and recode the one operation that it wants to change.

In a 1993 OOPSLA paper, Krueger et al. showed that in a very small amount of code, you can make a virtual memory that uses an MRU page-replacement policy.

These are examples of systems that provide clients with control over mapping decisions.

- Languages provide this control.
- Operating systems provide clients with control of virtual memory, etc. There has recently been an explosion of work on systems like Apertos, Scheduler Activations, all giving clients control of some mapping decisions.
- Many class libraries available in o-o community also have this same kind of property.

Mapping dilemmas are important when you have high-level functionality (lots of mapping to do) that needs to be efficient (the dilemma is serious).

An important paper is the one that introduces policy/mechanism separation ("Policy/mechanism separation in Hydra," R. Levin et al., SOSP '75). The first sentence is,

> "The extent to which resource allocation policies are entrusted to user-level software determines in large part the degree of flexibility present in an operating system."

That sentence is another way of saying that clients need control over mapping decisions.

So, what we have said is that earlier systems perhaps hid a little too much, because clients lost control over mapping decisions.

[Video of Tacoma Narrows Bridge collapse.]

Bridge was built in 1940. At the time, they used an abstraction framework which ignored the dynamic properties of structures like this. They knew that dynamic properties had been important, but they decided that they could be ignored. On Nov. 7, 1940, it got windy in the wrong way, and the bridge collapsed.

So, even a much older discipline can pick an abstraction framework that doesn't capture all of what matters.

Where are we now? We started out with the black-box abstraction, but found that we had to give clients control over mapping decision. We have seen some systems that try to do this in various ways.

However, the original motivation for hiding implementation was controlling the complexity presented to clients. If we now expose some of this implementation to allow control over mapping decisions, clients may again have trouble.

Giving clients control: the problem.

- Adding client control re-introduces complexity (which is what we were trying to hide in the first place).
- “If you can’t conquer, at least divide.“ We tried to conquer implementation, but it didn’t quite work. But, all by itself, this principle doesn’t say much more than use abstraction.
- But divide from what?

Base/meta separation: I think we can learn from the the community that works on computational reflection. This is the design of systems that have interfaces that need to be renegotiated. Like OOP, it is based on observation about the world, and claims that we can use that observation as a metaphor for designing systems.

It is based on the observation that people in the world renegotiate their contracts, sometimes very explicitly.

Day-to-day discourse is governed by contracts, e.g., about the delivery of widgets (what a widget is, how long it takes to build, etc.). “We need 1000 more widgets.”

Occasionally, contract needs to be renegotiated. On these occasions, discourse is about the contract. Reflective community calls this “going meta.”

One thing that the reflective community has given us is this notion of a reflective module. In such a module,

- the basic interface provides day-to-day functionality.
- the meta interface gives control over the base interface.

Base/meta is separation principle; separation *and* coordination implementation techniques.

All of us know that our friends who “go meta” all the time are somewhat of a pain, and it would be better if they did it only some of the time.

Basic design goal: Given this notion of reflective module, I want to give a basic design goal for these systems that present clients with control over mapping decisions.

“Hiding implementation” is replaced by “*separating* control over mapping decisions.” Goal is to allow programmer to program optimistically on top of window system, virtual memory, programming language.

Another interface, as much as possible separate, should allow them to adjust mapping decisions to meet optimism.

Allow client programmer to alternately focus attention (mostly) on one program or the other.

Let’s try to apply base/meta separation to some of the examples we’ve seen.

- The two programming-language examples (C and High-Performance Fortran) already provide good separation. Base and meta can be read and understood independently.
- The same holds for the virtual-memory system from above. The program for changing the paging policy can really very much be put off to the side from the program that uses virtual memory.

Here’s a slide borrowed from Taligent. It has base/meta separation.

- Client (base) API provides services. Looks like a class library. Instantiate and use. Your code calls it.
- Framework API. Provides customizability. Looks like an o-o design. Subclass and override. Calls your code.

Now let’s go back to window-system example. Here the base/meta separation wasn’t so good.

Before, we had a function called make-lightweight-text-window. The meta concern of how the mapping dilemma should be resolved had been confounded with the base concern that this was a text window. In addition, there is also a problem of combinatorial explosion. If you have a number of meta concerns and a number of base concerns, you get an explosion in name space.

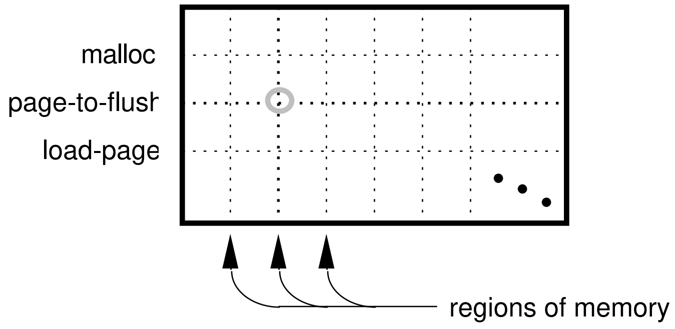
Separate base and meta namespaces. Avoids name explosion (it is easier to adjust meta independently of base).

At this year's OOPSLA, there is a paper by Lortz and Shin that gives a number of examples of achieving base/meta separation. Here the class `set` is clearly separated from client control over the mapping decision.

Let me talk now about a few additional design goals.

- Incrementality: Deltas from a good default. (Clients would like to write as little code as possible to change mapping decisions.)
- Scope control: What is affected. (If client uses meta interface to change a mapping decision, it should be able to bound the scope of the change, e.g., change how a single array is laid out, not all arrays.)

OO for Scope Control



- Interoperability: (If you change mapping decision for one array, it should be possible to use that array in a program that normally uses other arrays.)

The role of OOP: OOP is a very good way to achieve this separation.

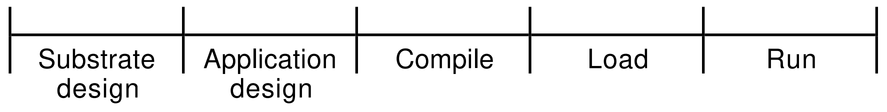
- You can use o-o to achieve incrementality. You use it to get an internal module structure that allows customization.
- O-o can also be used to achieve scope control. In the VM systems, they got scope control by using OOP to get a kind of cartesian coordinate system. In the slide, vertical axes are individual regions of memory; horizontal axes are operations on those regions. OOP allows clients to choose a different page-to-flush operation for one region of memory.

One combination of base/meta separation and OOP is the concept of meta-object protocol. It is a system that gives clients control over internal aspects of the system: Base & meta interfaces; OOP for scope, incrementality, meta objects in the meta interface.

One more design goal: efficiency. This is one reason why we said, We need to give clients more control over mapping decisions. We need to be sure that giving clients control doesn't cost us more efficiency than it gains.

Let me make the observation that the game we're playing is all about binding time.

The later you make the decision, the more control client has:



← Easier to recover decision-making overhead

More is known; → potentially better decisions

This period of time between compile- and run-time that we've traditionally thought about as being very quantized can actually be thought of as more of a spectrum.

For any critical operation, you'll run it a number of times. So maybe you can defer some of the work from the first running to later runs.

There has been some work on advanced compiler techniques to allow code to be generated

- when needed
- customized
- highly optimized.

There's a triad of techniques

- Partial evaluation
- Lazy evaluation
- Run-time code generation.

There has been a lot of work on this in Smalltalk and Self implementations.

One relevant paper is a little-known paper from SOSP ’89 by Henry Massalin and Calton Pu, “Threads and input/output in the synthesis kernel.”

> “We have introduced the principles of code synthesis … frequently executed synthesis kernel calls are “compiled” and optimized at run time using ideas similar to currying and constant folding … when we open a file for input, a custom-made (thus short and fast) read routine is returned for later read calls.”

File-system implementation is constantly being tuned to suit a particular implementation and a particular file.

While there’s no meta-interface per se, this technology can be used to get efficient meta-interfaces, because it allows very late binding of mapping decisions.

Another summary: We are engineers, and our chief task is to control complexity.

The basic tools that we use to do that are abstraction and decomposition.

In our discipline to date, we have tried a particular abstraction framework, called black-box abstraction, which attempts to expose functionality and hide implementation.

We’ve run into problems, because mapping decisions nonetheless show through. That leads to hematomas and coding between the lines in clients of these abstractions.

The fundamental problem is that clients need control over the mapping decisions.

Many systems exposed various kinds of mapping decisions.

If we are going to give clients control, you still need to control complexity. If you can’t conquer, you should at least divide. I talked about principles like scope control and incrementality that are important to getting that right.

We can look at work on computational reflection, OOP, and compiler techniques. We can use case studies to learn how to design systems that are going to give clients the control that they need.

Value of this new abstraction framework:

- Allows us to talk about code bloat, “This hematoma comes from a mapping decision …”
- We can request features from the module designer: “I need control over this mapping decision.”
- Helps us to evaluate module designs:
    - “How good is the base/meta separation?”
    - “How fine-grained is the scope control?”
    - “How good is the incrementality?”
    - “What technology does this meta-interface use?”

I have tried to lay out the framework of a new abstraction structure. There is a lot of work left to do.

Other issues:

- How to specify mapping dilemmas. I think we will need to learn new methodologies. I think they will be highly iterative.
- Specification. How will we deal with giving clients control over implementation decisions in a way they will understand?
- Verification. If clients are to be able to change something, we have to be able to test the system.
- Functionality dilemmas. This talk has motivated giving clients control based entirely on a performance argument. But one of things we learned on CLOS meta-object protocol is that sometimes clients want to change the behavior or semantics of the system, in addition to changing the mapping decision. We have to provide ways to do that.

Summary: I have been talking about a new set of words and way of understanding an old problem that will help us build systems that are important today.

Let’s look a little farther in the future and see what is really going on.

- We tried to take an abstraction mechanism and hide implementation. That didn’t work, because implementation showed through.

  We knew from the beginning that those interfaces were only partial descriptions of what was going on.

We designed them that way because what is really going on is much too complicated for us to cope with.

- The same is true in any engineering discipline. Suppose I want to build a simple “brick” out of blocks. This is really a complex system; several forces are acting on blocks.

  Other engineering disciplines deal with the problem that no one description can capture the complexity of the system. So they have multiple descriptions to capture more or less detail.

  Other descriptions capture entirely different issues.

- In software, we have a very similar situation. We have abstract descriptions like high-level pseudo-code.

  Many things aren’t said.

  Can be “run” to automatically produce complete behavior. (That is unique to software.)

  The substrate implementation fills in the missing pieces.

What would it mean to have the best of our side of engineering and their side of engineering?

Long-term goal: The best of both worlds.

- Abstract descriptions to manage complexity.
- Multiple descriptions to cover all of what matters.
- Automatic combination and execution, because that’s what’s special about software.

One description has to work hard not to say things, so that they can be said in another description.

We need multiple, automatically combined and executed descriptions.

This is a long-term goal, but today we have seen some steps in this direction.