The Design of a Class Mechanism for Moby

Kathleen Fisher
AT&T Labs — Research

John Reppy
Bell Labs, Lucent Technologies

May 1999

What is Moby?

- Moby is an ML-like language with support for class-based, object-oriented programming.
- Moby also includes other ML-style features, such as inductive datatypes, pattern matching, exceptions, and parametric polymorphism.
- We designed Moby in the context of the ML2000 effort; that group has influenced its design substantially.

Design Philosophy

Swiss-Army Knife approach:	vs.	Lego approach:
Cool gizmo that does (almost) everything.		Simple pieces that fit together well.

Challenge: Design a class mechanism that

- provides rich object-oriented programming idioms,
- has minimal overlap with module system.

Moby Bricks

Modules and signatures

- Modules group definitions and may be parameterized over other modules. Signatures control external visibility and may be specified independently.

Classes and class interfaces

- Classes define objects and support implementation inheritance. Class interfaces are the types we give to classes in signatures.

Objects and object types

- Objects are generated by classes. Object subtyping is *structural*.

Visibility Control in Moby

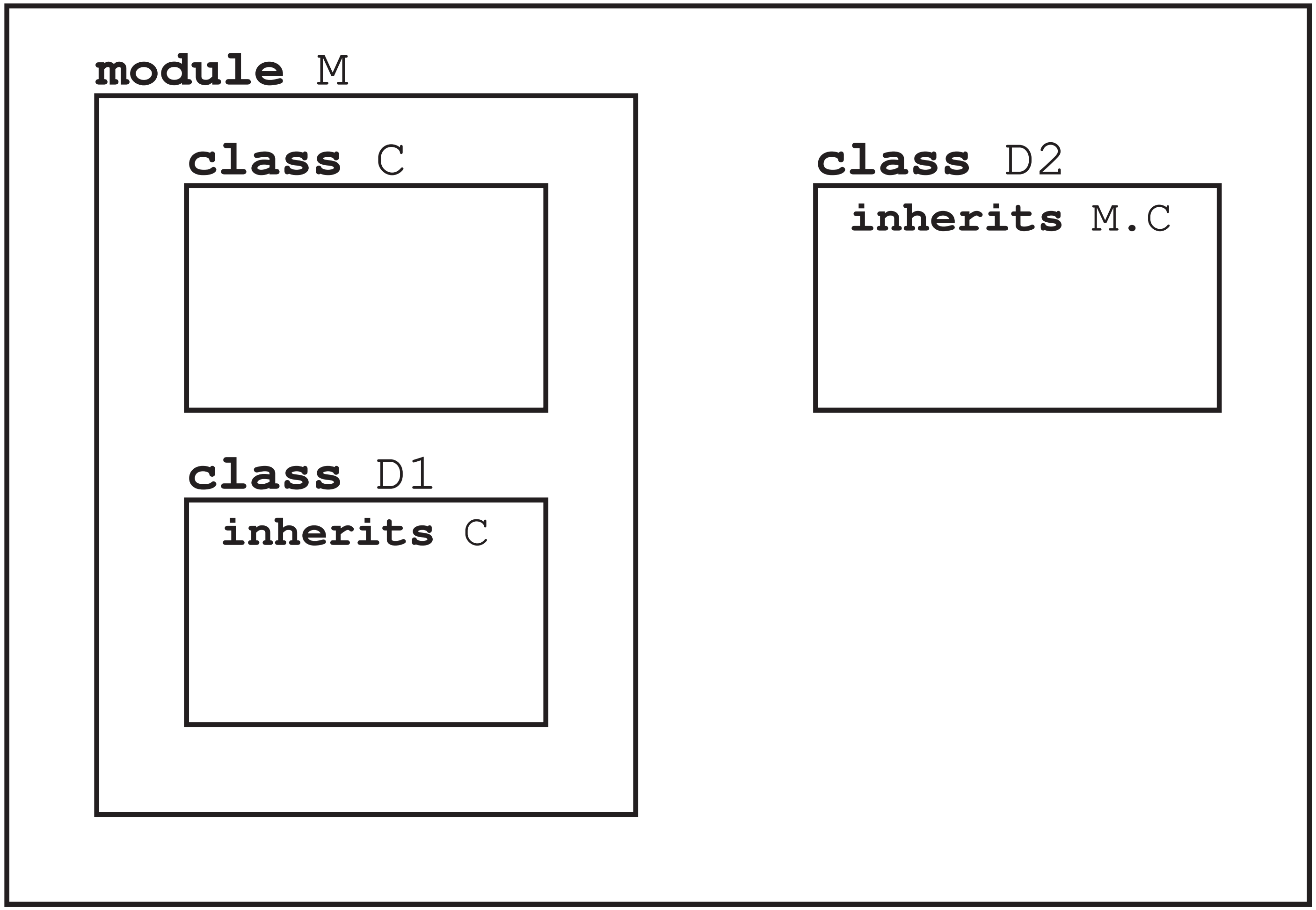
In Moby, modules and classes both divide code into two privilege classes.

- Modules: inside *vs.* outside
- Classes: object *vs.* class clients

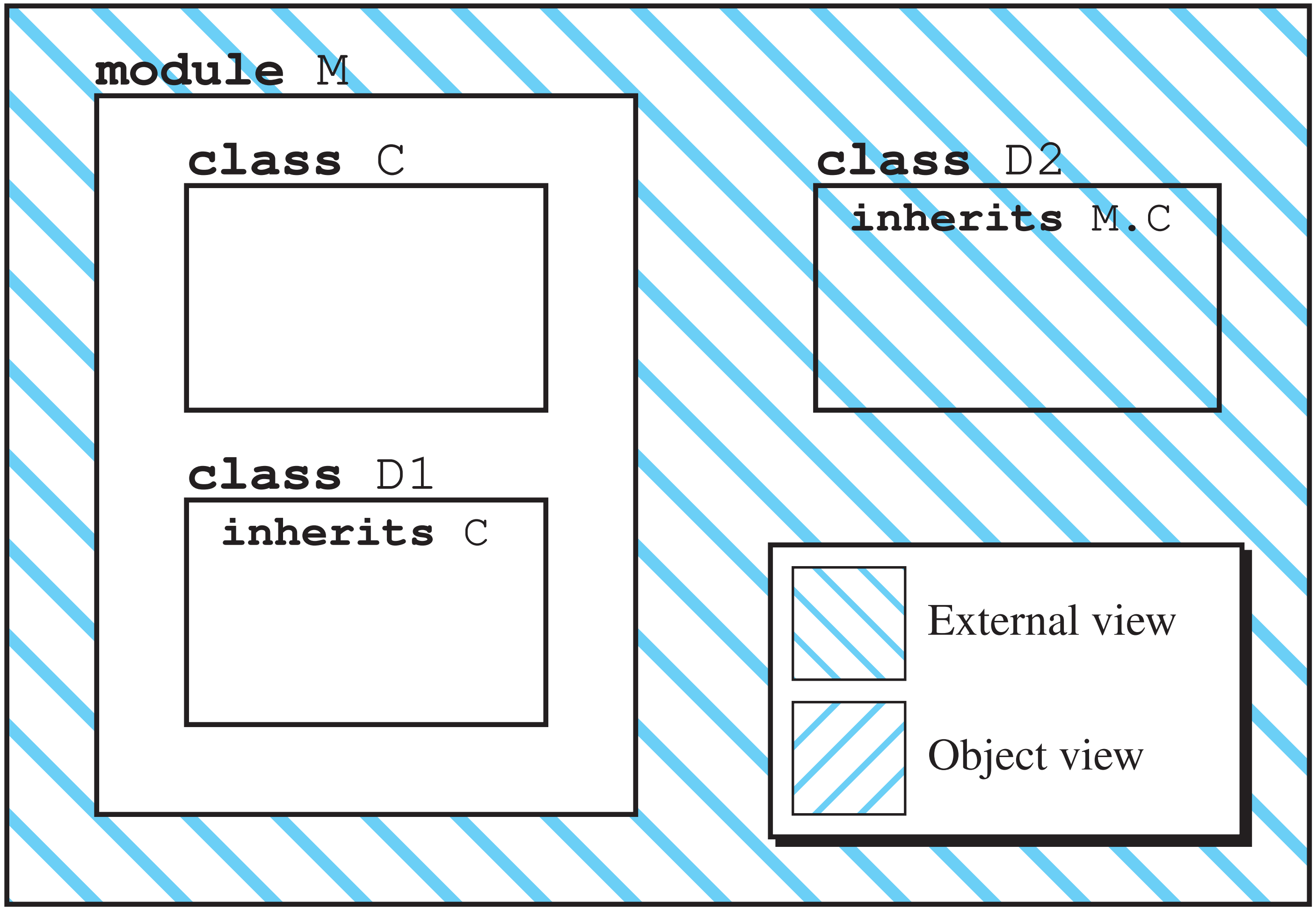
These divisions can be combined to emulate the standard kinds of access control:

	Class view	Object view
Internal	“private”	“package scope”
External	“protected”	“public”

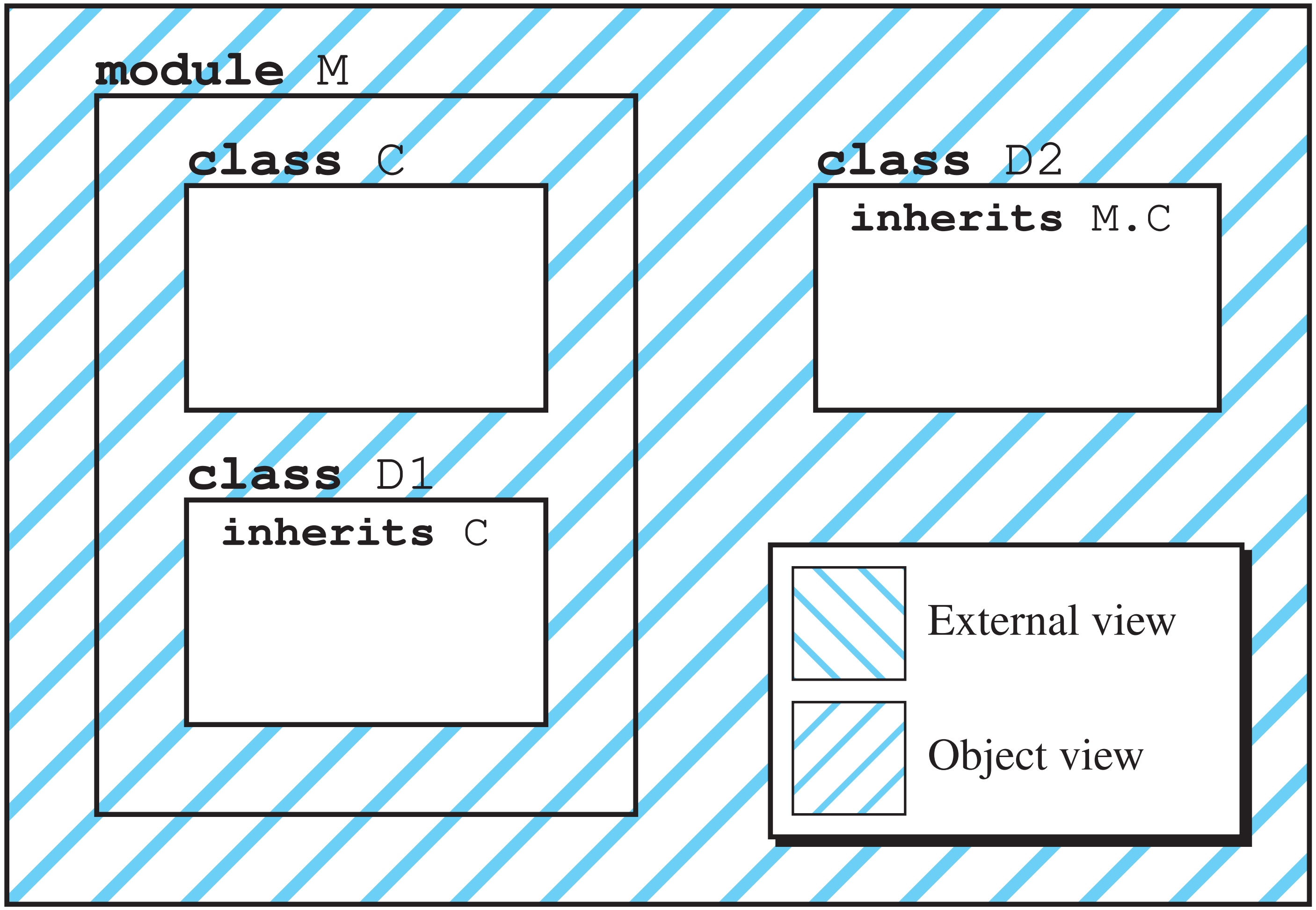
Visibility Control in Moby: An Example



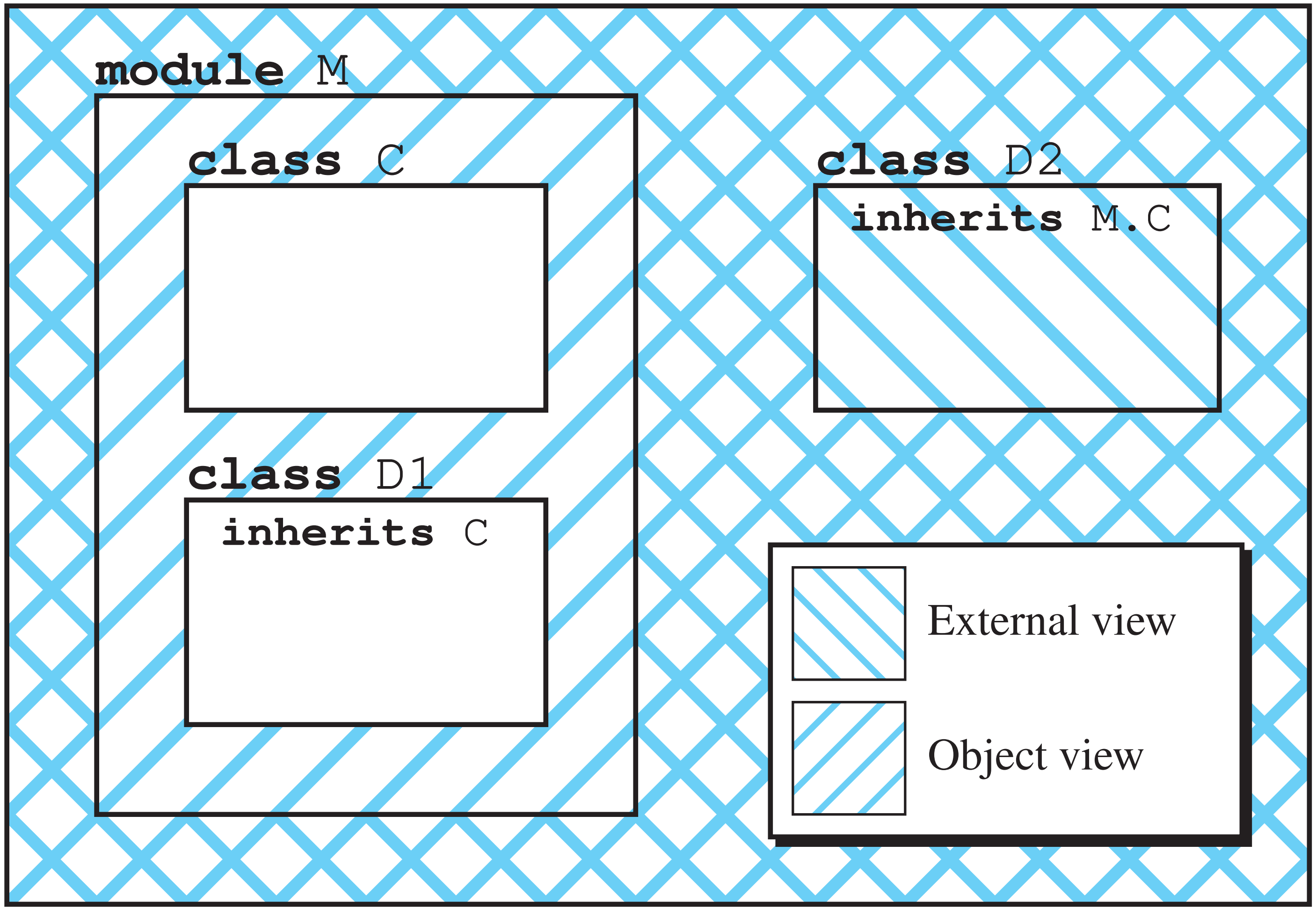
Visibility Control in Moby: External View



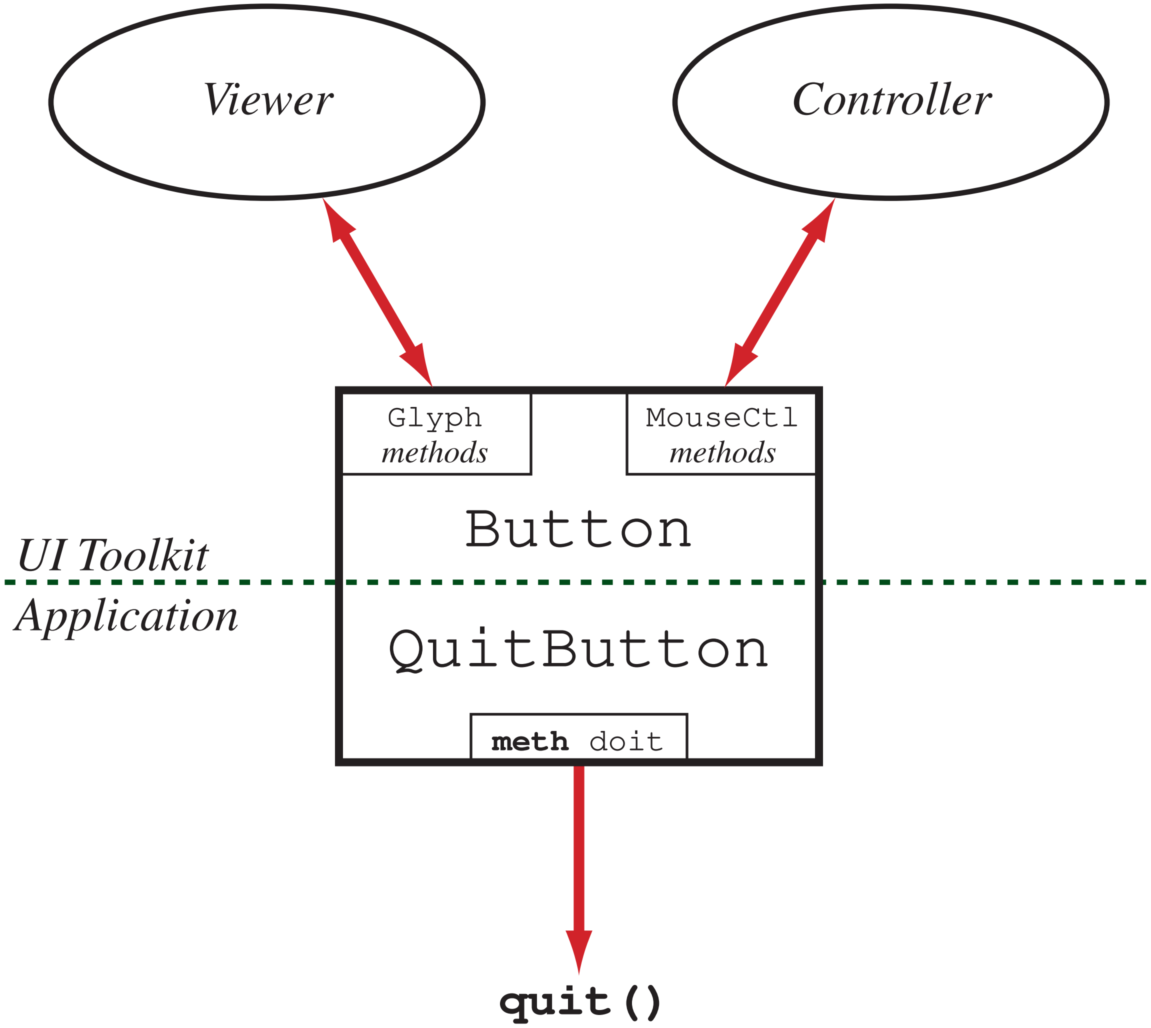
Visibility Control in Moby: Object View



Visibility Control in Moby



Extended Example: Quit Button Widget



The Viewer

- Signatures describe module contents.
- Dot notation is used to access components.
- Object types contain method and field descriptions.
- Object types are recursive.

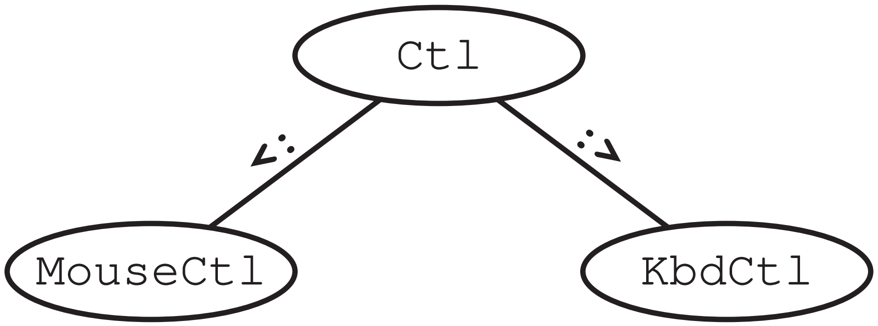
```
signature VIEWER {
  type View
  objtype Glyph {
     field view : View
     meth draw : Geom.Rect -> Unit
     meth pick : Geom.Pt -> Option(Glyph)
  }
}
```

The Controller

```
signature CONTROLLER {
  objtype Ctl {
     meth glyph : Unit -> Viewer.Glyph
  }

  objtype MouseCtl {
     meth glyph : Unit -> Viewer.Glyph
     meth mouseDn : Geom.Pt -> Unit
  }

  objtype KbdCtl extends Ctl {
     meth keyPress : String -> Unit
  }

  val addKbdCtl : KbdCtl -> Unit
  val addMouseCtl : MouseCtl -> Unit
}
```

Controller Notes

- Signature `CONTROLLER` describes interface to controller module.
- Structural subtyping:



- Object types support both *width* and *depth* subtyping.
- Keyword **`extends`** syntactic sugar for copying components.

The Button Class

```
class Button {
  field label : String

  public field view : Viewer.View
  public meth draw (r : Geom.Rect) : Unit { ... }
  public meth pick (p : Geom.Pt) : Option(Viewer.Glyph) { ... }

  public meth glyph () : Viewer.Glyph { self }
  public meth mouseDn (p : Geom.Pt) : Unit {
    self.doit()
  }
  ...
  abstract meth doit : Unit -> Unit
  ...
  maker mkButton(v : Viewer.View, s : String) {...}
  ...
}
```

Button Class Implementation Notes

- Class contains a sequence of method, field, and maker declarations.
- Keyword `self` used to refer to other components of host object.
- Keyword `public` indicates that component may be used by *object clients* of this class.
- Unmarked components may only be used by deriving classes (*class clients*).
- Methods marked `abstract` must be overridden by deriving classes before an object can be instantiated.

The Text Button Module

The signature exposes `Button` class but hides the `Controller` methods.

```
module TextButton : {
  class Button {
     public field view : Viewer.View
     public meth draw : Geom.Rect -> Unit
     public meth pick : Geom.Pt -> Option(Glyph)
     abstract meth doit : Unit -> Unit
     maker mkButton of (Viewer.View * String)
  }
} {
  ... implementation of Button class ...
}
```

The Application

The application specializes the `Button` class to implement `QuitButton`.

```
module Application {
  fun quit() : Unit { ... }
  ...
  class QuitButton {
     inherits TextButton.Button
     meth doit () : Unit { quit() }
     public maker mkQuitButton of (v : Viewer.View) { ... }
    }
  }
  ...
}
```

Application Notes

- `QuitButton` inherits from `TextButton.Button` through the signature on the previous slide. Consequently, `QuitButton` objects are known to be subtypes of `Viewer.Glyph`, but not `Controller.Ctl`.
- Method `doit` cannot be called by object clients.
- Maker `mkQuitButton` is **`public`**; it may be used to instantiate objects.

Object Creation and Initialization Code

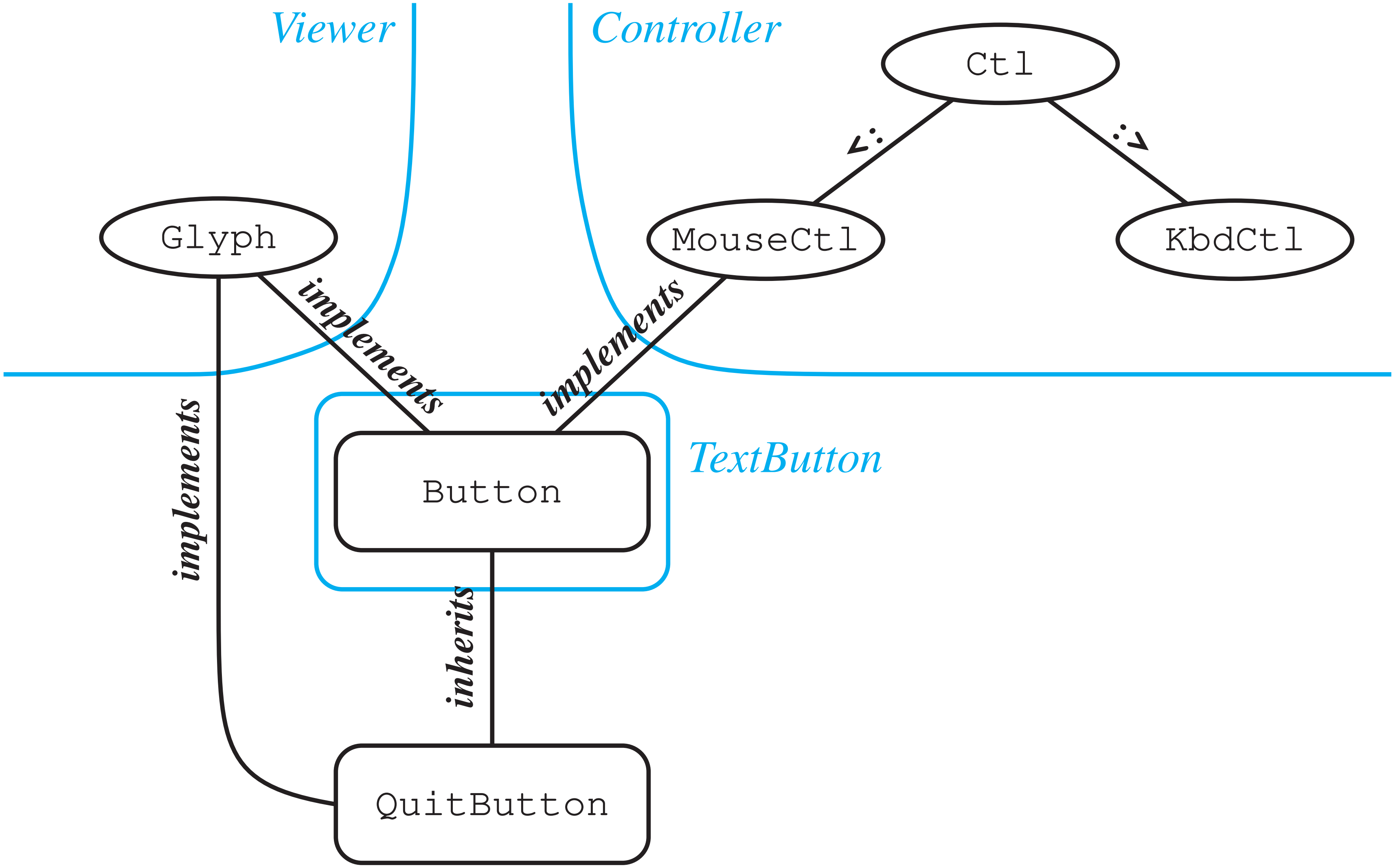
```
class Button {
  ...
  maker mkButton(v : Viewer.View, s : String) {
    field view = v;
    field label = s
  }

  initially Controller.addMouseCtl(self)
}



class QuitButton {
   inherits TextButton.Button
   ...
   public maker mkQuitButton of (v : Viewer.View) {
      super.mkButton(v, "Quit")
   }
}
```

Object Creation and Initialization

The expression `new mkQuitButton(v)` executes as follows:

1. The maker `mkQuitButton` in the `QuitButton` class is invoked.
2. It invokes the `mkButton` maker from its super-class.
3. The `mkButton` maker initializes the `view` and `label` fields, and then returns.
4. The `mkQuitButton` returns and the object is *created*.
5. The new object is passed as **`self`** to the **`initially`** clause in the `Button` class.
6. There is no **`initially`** clause in the `QuitButton` class, so the new object is returned to the creating context.

Class and Type Relations



Other OOP Idioms

- Static members: Moby module system.
- C++ friends, Java package scope: Moby module system and partial type revelations.
- C++ private inheritance: Moby inheritance plus signature ascription.
- Java interfaces: Moby object types.
- Final classes: hide class in module, export creation functions.

Summary

Moby provides ...

- a rich module system.
- a simple class-based object-oriented programming mechanism.
- support for common OOP idioms.
- minimal overlap between class and module mechanisms.

Modules *vs.* Classes in Moby

Feature	Modules	Classes
Name spaces	√	
Private scope	√	
Protected scope		√
Friends via package scope	√	
Signature matching	√	
Data abstraction	√	
Type abstraction	√	
Partial type abstraction	√	
Parameterization	√	
Import	√	
Implementation inheritance		√
Value/type definitions	√	
Method/field definitions		√
Object construction		√

Related Work

Languages with strong class mechanisms and weak module mechanisms:

- C++
- Java
- Eiffel

Languages with strong module mechanisms and weak class mechanisms:

- Modula-3
- OCaml

Classes via modules:

- OML (PLDI'96, FOOL 3; 1996)
- Riecke-Stone (unpublished; 1997)
- Vouillon (FOOL 5; 1998)

Comparison with OCaml

Major differences: OCaml ...

- extends Hindley-Milner type inference with row polymorphism.
- supports *mytype* and binary methods.
- does not support private inheritance.
- does not support package scope.

Comparison with Java

Major differences: Moby ...

- provides structural subtyping.
- object types can have fields.
- supports private inheritance.
- supports “abstract” base classes (via functors).
- provides value-oriented object creation.
- uses signature matching to control visibility.
- support parameterized classes (via functors and polymorphism).

Status

- Language design is 90% complete.
- Parser is implemented; type-checker under construction.
- Foundations of object and class mechanisms described in a technical report.
- More information available from
  `http://www.cs.bell-labs.com/~jhr/moby/index.html`

Status (continued)

- Logo has been designed.

`http://www.cs.bell-labs.com/~jhr/moby/index.html`