Modules and Abstract Data Types

CSI 3120

Amy Felty

University of Ottawa

The Reality of Development

We rarely know the *right* algorithms or the *right* data structures when we start a design project.

- When implementing a search engine, what data structures and algorithms should you use to build the index? To build the query evaluator?

Reality is that *we often have to go back and change our code*, once we've built a prototype.

- Often, we don't even know what the *user wants* (requirements) until they see a prototype.
- Often, we don't know where the *performance problems* are until we can run the software on realistic test cases.
- Sometimes we just want to change the design -- come up with *simpler* algorithms, architecture later in the design process

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

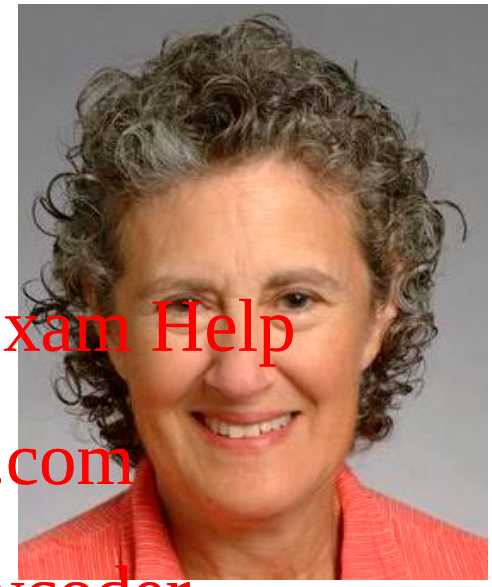
The primary trick: use *data and algorithm abstraction*.

- *Don't* code in terms of concrete features that the language provides.
- *Do* code with *high-level abstractions* in mind that fit the problem domain.
- Implement the abstractions using a *well-defined interface*.
- Swap in *different implementations* for the abstractions.
- *Parallelize* the development process.

Abstract Data Types

Barbara Liskov
Assistant Professor, MIT
1973

Invented CLU language
that enforced data abstraction



Barbara Liskov
Professor, MIT
Turing Award 2008

"For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

Language-enforced Abstraction

Rule of thumb*: *Use the language to enforce an abstraction.

- **Murphy's law for unenforced data abstraction**:
  - What is not enforced, will be broken at some point, by a client

- **This is what module systems are for!**

  - reveal little information about *how* something is implemented
  - provide maximum flexibility for change moving forward.

  - pays off down the line

- **Like all design rules, break it when necessary**
  - recognize when a barrier is causing more trouble than it's worth

- **ML has a particularly great module system**

Building Abstract Types in OCaml

Use OCaml modules to build new abstract data types!

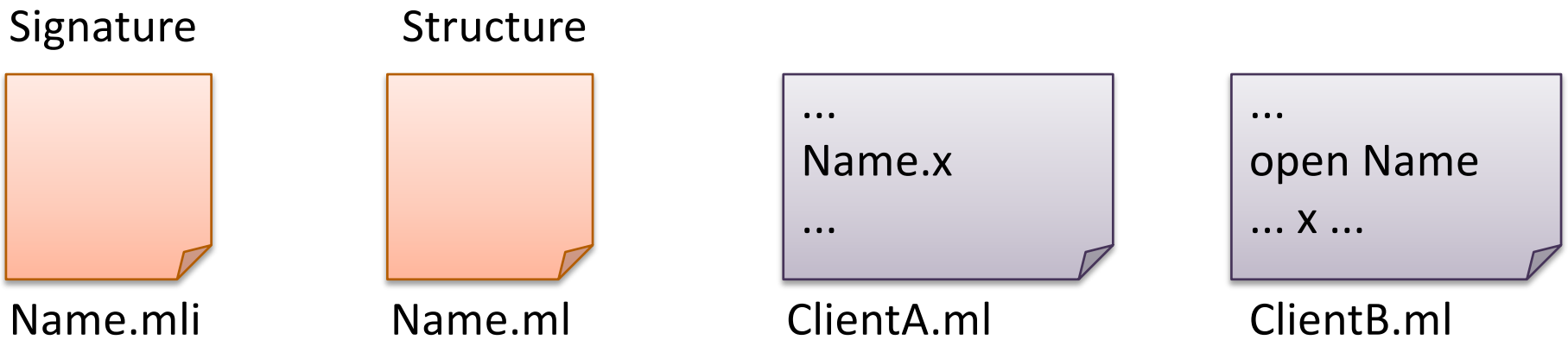
- ***signature***: an interface.
  - specifies the abstract type(s) without specifying their implementation
  - specifies the set of operations on the abstract types
- ***structure***: an implementation.
  - a collection of type and value definitions
  - notion of an implementation matching or satisfying an interface
    - gives rise to a notion of sub-typing
- ***functor***: a parameterized module
  - really, a function from modules to modules
  - allows us to factor out and re-use modules

Simple Modules

OCaml Convention:

- file Name.ml is a *structure* implementing a module named Name
- file Name.mli is a *signature* for the module named Name
  - if there is no file Name.mli, OCaml infers the default signature
- Other modules, like ClientA or ClientB can:
  - use *dot notation* to refer to contents of Name. eg: Name.val
  - open Name: get access to all elements of Name
    - opening a module puts lots of names in your namespace

Open modules with discretion!



At first glance: OCaml modules = C modules?

C has:

- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:

- tighter control over type abstraction
  - define abstract, transparent or translucent types in signatures
    - *i.e.,* give none, all or some of the type information to clients
- more structure
  - modules can be defined within modules
  - *i.e.,* signatures and structures can be defined inside files
- more reuse
  - multiple modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules!

At first glance: OCaml modules = C modules?

C has:

- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:

- tighter control over type abstraction
  - define abstract types
    - *i.e.*, give only some definitions to clients
- more
  - modules
  - *i.e.*, sig
- more reuse
  - multiple modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules!

ML = Winning!

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```

convention: when the module is about 1 data type, use **t** as the name of the type.

clients refer to **Stack.t**

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```

empty and push are abstract ***constructors***: functions that build our abstract type.

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int
  end
```


is_empty is an ***observer*** – useful for determining properties of the ADT.

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```


pop is sometimes called a ***mutator*** (though it doesn't really change the input)

Example Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```


top is also an *observer*, in this functional setting since it doesn't change the stack.

Put comments in your signature!

```
module type INT_STACK =
  sig
    type t
    (* create an empty stack *)
    val empty : unit -> t

    (* push an element on the top of the stack *)
    val push  : int -> t -> t

    (* returns true iff the stack is empty *)
    val is_empty : t -> bool

    (* pops top element off the stack;
       returns empty stack if the stack is empty *)
    val pop : t -> t

    (* returns the top element of the stack; returns
        None if the stack is empty *)
    val top : t -> int option
  end
```

Signature Comments

- Signature comments are for clients of the module
  - explain what each function should do
    - how it manipulates abstract values (stacks)
  - ***not*** how it manipulates concrete values
  - don't reveal implementation details that should be hidden behind the abstraction

- Don't copy signature comments into your structures
  - your comments will get out of date in one place or the other
  - an extension of the general rule: don't copy code

- Place implementation comments inside your structure
  - comments about implementation invariants hidden from client
  - comments about helper functions

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type t = int list
    let empty () : t = []
    let push (i:int) (s:t) : t = i::s
    let is_empty (s:t) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:t) : t =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:t) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type t = int list
    let empty () : t = []
    let push (i:int) (s:t) : t = i::
    let is_empty (s:t) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:t) : t =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:t) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

Inside the module, we know the ***concrete type*** used to implement the abstract type.

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type t = int list
    let empty () : t = []
    let push (i:int) (s:t) : t =
    let is_empty (s:t) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:t) : t =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:t) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

But by giving the module the INT_STACK interface, which does not reveal how stacks are being represented, we prevent code outside the module from knowing stacks are lists.

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
```



```
s0 : ListIntStack.t
s1 : ListIntStack.t
s2 : ListIntStack.t
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
         (* i : int option = Some 4 *)
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
         (* i : int option = Some 4 *)
let  j = ListIntStack.top (ListIntStack.pop s2)
         (* j : int option = Some 3 *)
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
         (* i : int option = Some 4 *)
let  j = ListIntStack.top (ListIntStack.pop s2)
         (* j : int option = Some 3 *)
open ListIntStack
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let  i = ListIntStack.top s2
         (* i : int option = Some 4 *)
let  j = ListIntStack.top (ListIntStack.pop s2)
         (* j : int option = Some 3 *)
open ListIntStack
let  k = top (pop (pop s2))
         (* k : int option = None *)
```

An Example Client

```
module type INT_STACK =
  sig
    type t
    val push  : int -> t -> t
    …

module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4 s1
…
let l  = List.rev s2
```

```
Error: This expression has type ListIntStack.t but
an expression was expected of type 'a list.
```


Notice that the client is not allowed to know that the stack is a list.

Example Structure

```
module ListIntStack (* : INT_STACK *) =
  struct
    type t = int list
    let empty () : t = []
    let push (i:int) (s:t) = i::s
    let is_empty (s:t) =
      match s with
        | [ ] -> true
        | _::_ -> false
    exception EmptyStack
    let pop (s:t) =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:t) =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```


Note that when you are debugging, you may want to comment out the signature ascription so that you can access the contents of the module.

The Client without the Signature

```
module ListIntStack (* : INT_STACK *) =
  struct
    …
  end

let s = ListIntStack.empty()
let s1 = ListIntStack.push 3 s
let s2 = ListIntStack.push 4 s1

…
let  l = List.rev s2
         (* l : int list = [3; 4] *)
```


If we don't seal the module with a signature, the client can know that stacks are lists.

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type t = int list
    let empty () : t = []
    let push (i:int) (s:t) = i::s
    let is_empty (s:t) =
      match s with
        | [ ] -> true
        | _::_ -> false
    exception EmptyStack
    let pop (s:t) =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:t) =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

When you put the signature on here, you are restricting client access to the information in the signature (which does *not* reveal that stack = int list.) So clients can *only* use the stack operations on a stack value (not list operations.)

Example Structure

```
module type INT_STACK =
  sig
    type stack
    ...

    val inspect : stack -> int list
    val run_unit_tests : unit -> unit

  end

module ListIntStack : INT_STACK =
  struct
    type stack = int list
    ...

    let inspect (s:stack) : int list = s
    let run_unit_tests () : unit = ...
  end
```


Another technique:

Add testing components to your signature.

Another option: we have 2 signatures, one for testing and one for the rest of the code)

Recall the Integer Stack Signature

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end
```

A Signature for Polymorphic Stacks

```
module type INT_STACK =
  sig
    type t
    val empty : unit -> t
    val push  : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top
  end
```



```
module type STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push  : 'a -> 'a stack -> 'a stack
    val is_empty : 'a stack -> bool
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a option
  end
```

One Implementation

```
module ListStack : STACK =
  struct
    type 'a stack = 'a list
    let empty() : 'a stack = []
    let push (x:'a)(s:'a stack) : 'a stack = x::s
    let is_empty(s:'a stack) =
      match s with
      | [] -> true
      | _::_ -> false
    let pop (s:'a stack) : 'a stack =
      match s with
        | [] -> []
        | _::tl -> tl
    let top (s:'a stack) :'a option =
      match s with
        | [] -> None
        | h::_ -> Some h
end
```

Wrap up and Summary

- It is often tempting to break the abstraction barrier.
  - e.g., during development, you want to print out a set, so you just call a convenient function you have lying around for iterating over lists and printing them out.
- But the whole point of the barrier is to support future change in implementation.
  - e.g., moving from unsorted invariant to sorted invariant.
  - or from lists to balanced trees.
- Many languages provide ways to leak information through the abstraction barrier.
  - "good" clients should not take advantage of this.
  - but they always end up doing it.
  - so you end up having to support these leaks when you upgrade, else you'll break the clients.

Key Points

OCaml's linguistic mechanisms include:

- *signatures* (interfaces)
- *structures* (implementations)
- *functors* (functions from modules to modules)

We can use the module system

- provides support for *name-spaces*
- *hiding information* (types, local value definitions)
- *code reuse* (via functors, reuseable interfaces, reuseable modules)

Information hiding allows design in terms of *abstract* types and algorithms.

- think "sets" not "lists" or "arrays" or "trees"
- think "document" not "strings"
- the less you reveal, the easier it is to replace an implementation
- use linguistic mechanisms to implement information hiding
  - invariants written down as comments are easy to violate
  - use the type checker to guarantee you have strong protections in place