CS 3110

Modular Specification

Prof. Clarkson

Fall 2015

Today's music: In C by Terry Riley

Review

Previously in 3110:

- architecture and design of large programs

Today:

- more on design principles
- specification
  - for clients
  - for implementers

DESIGN, CONTINUED

Criteria for modular design

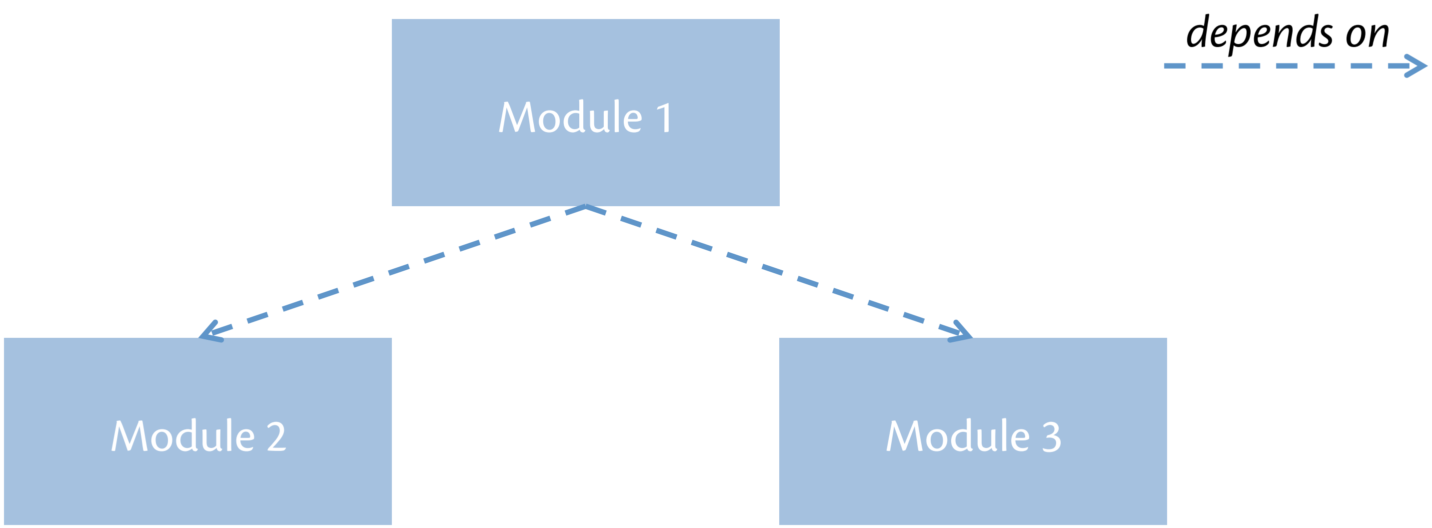
- **Coupling:** strength of relationship between modules
  - *highly coupled* modules have strong relationships with other modules
  - *loosely coupled* modules have weak relationships with other modules [good]
- **Cohesion:** strength of relationship within module
  - *highly cohesive* modules have strong relationships within module [good]
  - *loosely cohesive* modules have weak relationships within module

To reduce coupling...

- Keep external interfaces *narrow:*
  - hide representation types
  - hide helper functions
  - keep the number of functions small
- Keep external interfaces *simple:*
  - keep functions arguments few and their types small
  - don't let return values contain too much or too little information
- Pass *data* through interfaces but not *control:*
  - Passing control means telling the module what to do or how it should behave in the future
  - Passing data means just providing inputs that will be transformed into outputs

Dependence

- A module *depends on* another if it uses a value, function, or type from it
- Module dependency diagram (MDD) depicts that relationship



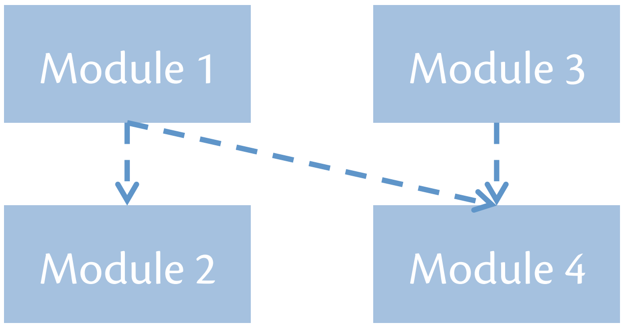
Dependence

- **Fan out** of *M*: number of modules *M* depends on
- **Fan in** of *M*: number of modules that depend on *M*
- both increase coupling
- cycles increase coupling

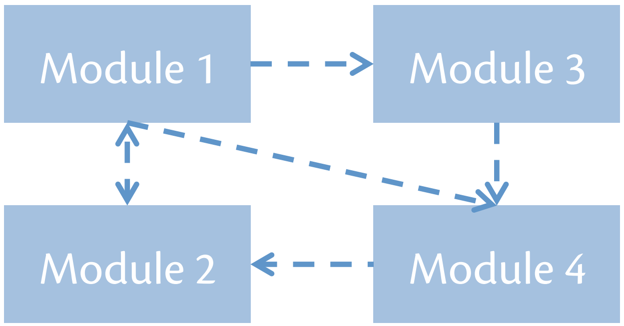
Question

Which of these MDDs exhibits weaker coupling?

A:



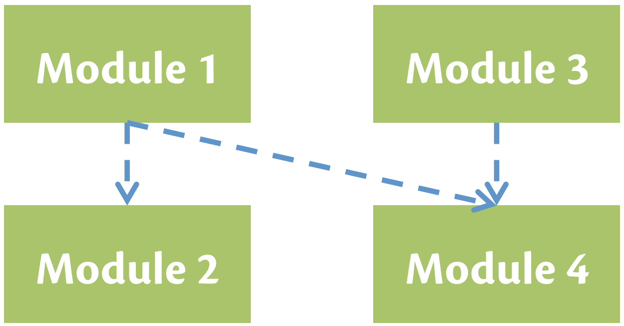
B:



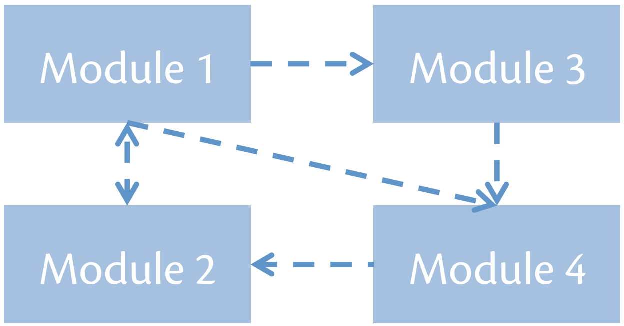
Question

Which of these MDDs exhibits weaker coupling?

A:



B:



To increase cohesion...

- Reduce coupling
  – Strong coupling can be a sign that code is in the wrong place
  – Redesign to move it into a more cohesive module
- Make sure all parts of interface are at least logically related
- Better yet, make sure all parts of module contribute toward performing a single purpose
- Try writing a single sentence that fully and accurately describes purpose of module
  – conjunctions, commas, and multiple verbs all suggest lower cohesion
  – words related to time ("first", "next", "after") suggest lower cohesion

Recap

- Architecture: highest-level design
  - components, connectors, constraints
  - some patterns: pipe and filter, shared data, client server
- System design
  - Simplicity is the main criterion
  - Principles: modularity = partitioning + abstraction
  - Strategies: top down vs. bottom up
  - Coupling and cohesion
- Next...detailed design through specification

Abstraction by specification

- Document behavior of function
  - Primarily, with pre- and postconditions
  - Use documentation to reason about behavior
    - instead of having to read implementation

- We've been teaching you this for three semesters now, I hope...but...
  - the language syntax doesn't demand it
  - the compiler doesn't check it
  - ...so writing good specs is a skill that takes longer to mature

Specifications

A **specification** is a contract between an implementer of an abstraction and a client of an abstraction

- Describes behavior of abstraction
- Clarifies responsibilities
- Makes it clear who to blame

An implementation **satisfies** a specification if it provides the described behavior

Many implementations can satisfy the same specification

- Client has to assume it could be any of them
- Implementer gets to pick one

Benefits of abstraction by specification

- **Locality:** abstraction can be understood without needing to examine implementation
  - critical in implementing large programs
  - also important in implementing smaller programs in teams
- **Modifiability:** abstraction can be reimplemented without changing implementation of other abstractions
  - update standard libraries without requiring world to rewrite code
  - performance enhancements: write the simple slow thing first, then improve bottlenecks as necessary

Good specifications

- **Sufficiently restrictive:** rule out implementations that wouldn't be useful to clients
  - common mistakes: not stating enough in preconditions, failing to identify when exceptions will be thrown, failing to specify behavior at boundary cases
- **Sufficiently general:** do not rule out implementations that would be useful to clients
  - common mistakes: writing operational specifications instead of definitional (saying how, not what), stating too much in a postcondition
- **Sufficiently clear:** easy for clients to understand behavior
  - common mistakes: verbosity, omission of details and examples, lack of structure
  - best case: client reads spec and comes away confused
  - worst case: client read spec, thinks they understand it, but they don't hence can't use abstraction correctly

Goal is to write specifications that are restrictive AND general AND clear

When to write specifications

- During design:
  - posing and answering questions about behavior clarifies what to implement
  - as soon as a design decision is made, document it in a specification
- During implementation:
  - update specification during code revisions
  - a specification becomes obsolete only when the abstraction becomes obsolete

Audience of specification

- Clients
  - Spec informs what they must guarantee (preconditions)
  - Spec informs what they can assume (postconditions)
- Implementers
  - Spec informs what they can assume (preconditions)
  - Spec informs what they must guarantee (postconditions)

But the spec isn't enough for implementers…

Example: sets

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val mem : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val size : 'a set -> int
end
```

Sets without duplicates

```
module ListSetNoDup : SET = struct
  (* the list may never have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l =
    if mem x l then l else x :: l
  let size = List.length
end
```

Sets with duplicates

```
module ListSetDup : SET = struct
  (* the list may have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = x :: l
  let rec size = function
  | [] -> 0
  | h::t -> size t +
              (if mem h t then 0 else 1 )
end
```

Compare set implementations

- Both have the same representation type, `'a list`
- But they interpret values of that type differently
  - `[1;1;2]` is {1,2} in `ListSetDup`
  - `[1;1;2]` is not meaningful in `ListSetNoDup`
  - In both, `[1;2]` and `[2;1]` are {1,2}
- Interpretation differs because they make different assumptions about what values of that type can be:
  - passed into operations
  - returned from operations
- e.g.,
  - `[1;1;2]` can be passed into and returned from `ListSetDup`
  - `[1;1;2]` should not be passed into or returned from `ListSetNoDup`

Question

Consider this implementation of *set union* with representation type **`'a list`**:

```
let union l1 l2 = l1 @ l2
```

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists

B. There could be duplicates in lists

C. Both A and B

D. Neither A nor B

Question

Consider this implementation of *set union* with representation type **`'a list`**:

```
let union l1 l2 = l1 @ l2
```

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists

B. There could be duplicates in lists

C. Both A and B

D. Neither A nor B

Representation type questions

- How to interpret the representation type as the data abstraction?

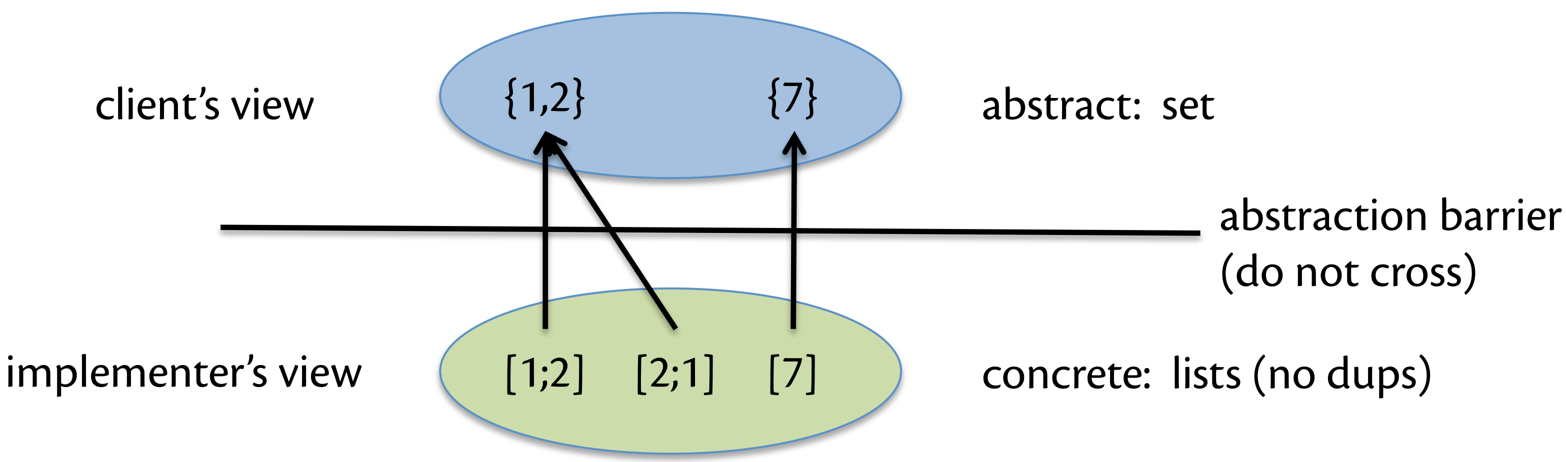
  ...abstraction function

- How to determine which values of representation type are meaningful?

  ...representation invariant

Abstraction function

- **Abstraction function** (AF) captures designer's intent in choosing a particular representation of a data abstraction
- Not actually an OCaml function, but a mathematical function
- Maps *concrete values* to *abstract values*



Documenting AFs

```
module ListSetNoDup : SET = struct
  (* AF: the list [a1; ...; an] represents
   *   the set {a1,...,an}.  [] represents
   *   the empty set. *)
  type 'a set = 'a list
  ...
end
module ListSetDup : SET = struct
  (* AF: the list [a1; ...; an] represents
   *   the smallest set containing the
   *   elements a1, ..., an.  [] represents
   *   the empty set. *)
  type 'a set = 'a list
  ...
end
```

Implementing AFs

- Mostly you don't
  - Would need to have an OCaml type for abstract values
  - If you had that type, you'd already be done...
- But sometimes you do
  - `string_of_X` or `toString()`
  - quite useful for debugging

Representation invariant

- Recall: AF may be partial
  - [1;1;2] is not a valid **`ListSetNoDup`**
- **Representation invariant** characterizes which concrete values are *valid* and which are *invalid*
  - “Rep invariant” or RI or just "invariant" for short
  - Valid concrete values will be mapped by AF to abstract values
  - Invalid concrete value will not be mapped by AF to abstract values
    - CANNOT meaningfully apply AF to values that don’t satisfy RI

Documenting RI

```
module ListSetNoDup : SET = struct
  (* AF: the list [a1; ...; an] represents
   *   the set {a1,...,an}.  [] represents
   *   the empty set. *)
  (* RI: the list contains no duplicates *)
  type 'a set = 'a list
end
module ListSetDup : SET = struct
  (* AF: the list [a1; ...; an] represents
   *   the smallest set containing the
   *   elements a1, ..., an.  [] represents
   *   the empty set. *)
  type 'a set = 'a list
end
```

Implementing the RI

- Great habit to cultivate
- Implement it EARLY, before any operations are implemented
- Common **idiom**: if RI fails then raise exception, otherwise return concrete value

```
let repOK (x:'a list) : 'a list =
  if has_dups x then failwith "RI"
  else x
```

- When debugging, check **`repOK`** on every input to an operation and on every output

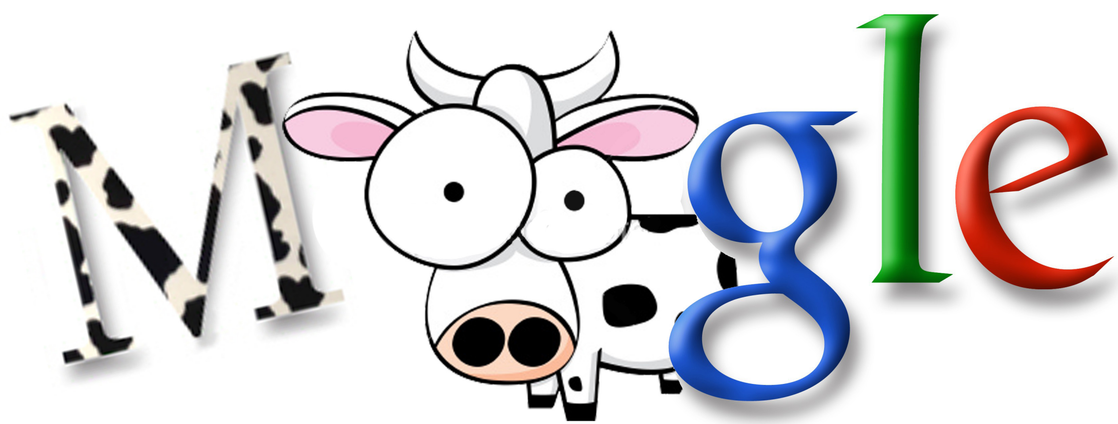
Checking the RI

```
module ListSetNoDup : SET = struct
  (* AF: ... *)
  (* RI: ... *)
  type 'a set = 'a list
  let repOK = ...
  let empty = repOK []
  let mem x l = List.mem x (repOK l)
  let add x l =
    if mem x (repOK l) then (repOK l)
    else repOK(x :: l)
  let size l = List.length (repOK l)
end
```

Funny story…this saved the CS 3110 tournament one year

A3

- Out now, due in about 9 days
- Implement a web search engine (crawler, indexer, efficient data structures)
- We've covered everything you need to get started



Upcoming events

- [today] A3 released

This is invariant.

THIS IS 3110