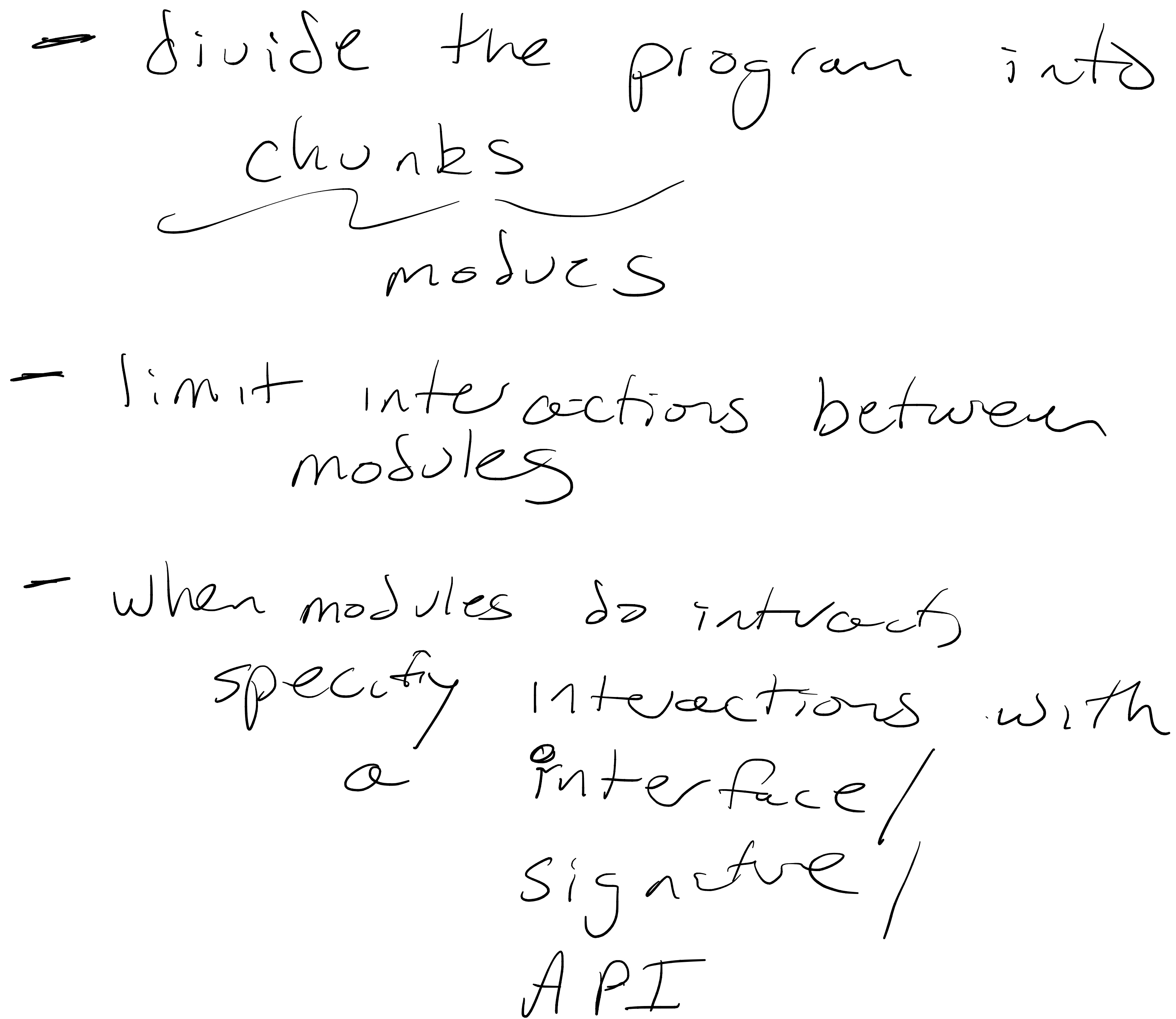
Lecture 17

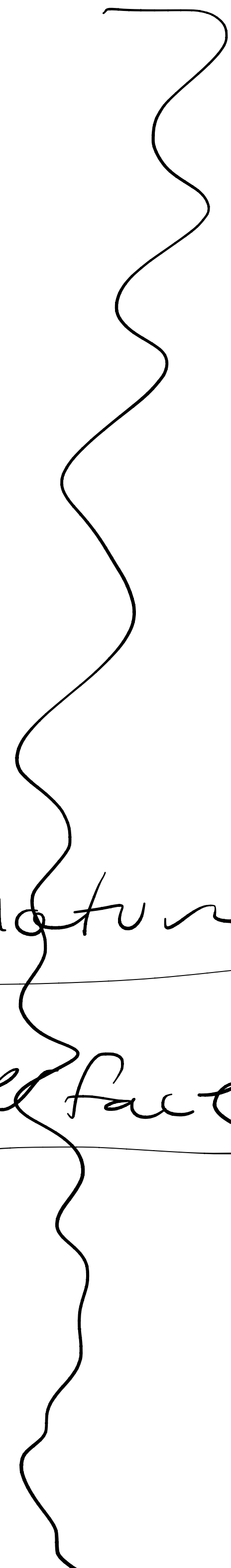
Modules

how to write bigger programs!


- divide the program into chunks
modules
- limit interactions between modules
- when modules do interact, specify interactions with a interface/ signature/ API

implementation

client



signature

interface

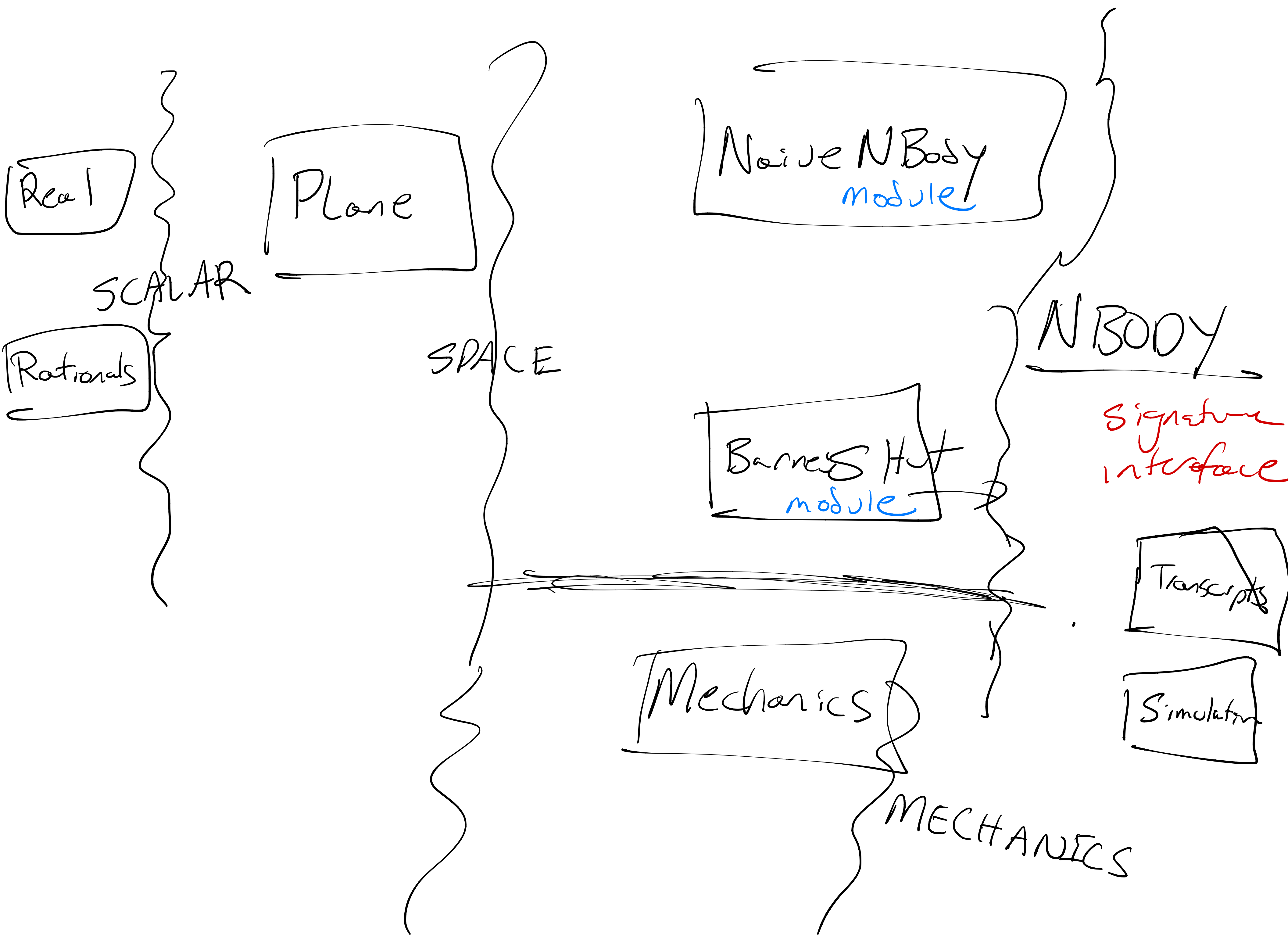
why?

1) make code easier to understand

2) allow separate client + implentation evolution

3) localize reasoning about invariants

4) catch more bugs at compile-time


Real
Rationals
SCALAR
Plane
SPACE
Naive NBody
module
Barnes Hut
module
NBODY
signature
interface
Transcripts
Simulation
Mechanics
MECHANICS

Client

```
fun countEvens(l: int list): int =
  case l of
    [] => 0
  | x::xs =>
      case evenP(x) of
        true => 1 + countEvens(xs)
      | false => countEvens(xs)

fun showEvens(l: int list): string =
  Int.toString (countEvens l)
```

Signature / Interface

```
signature COUNTER =
sig
    type counter
    val zero: counter
    val increment: counte -> counter
   val show: counter -> string
end
```

Implementation

IntCounter

BinCounter

COUNTER

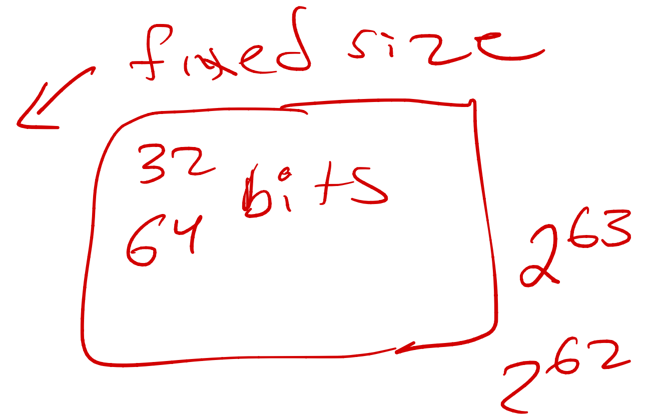
Client

structure C : COUNTER = BinCounter

```
fun countEvens(l:int list): C.counter =
  case l of
    [] => C.zero
  | x::xs =>
      case even(x) of
        true => C.increment(countEvens(xs))
      | false => countEvens(xs)

fun showEvens(l:int list): string =
  C.show(countEvens l)
```

```
structure IntCounter :> COUNTER =
struct
    type counter = int
    val zero = 0
    fun increment(x) = x+1
    fun show (x) = Int.toString (x)
end
```



1. the structure must provide a type/value for each thing in the sig
2. the values must have the indicated types
3. the implementation knows what the types <u>are</u>

counter type is

abstract

↳ client does not know counter = int

```
structure BinCounter :> COUNTER =
struct
  (* idea: [c0, c1, c2, ....]
           c0*2^0 + c1*2^1 + c2*2^2 + ..... *)

  type counter = int list
    (* invariants: no trailing 0's
                   int's always 0 or 1 *)

  val zero = []
```

```
fun increment l =
    case l of
      [] => [1]
    | 0 :: xs => 1 :: xs
    | 1 :: xs => 0 :: increment(xs)
```

```
(* i is the power of 2 for first elt of l *)

fun show_help(i:int, l:counter):String =
    case l of
      [] => "0"
    | x::xs => Int.toString(x) ^ "*2^" ^
               Int.toString(i)
               ^ "+"
               ^ show_help(i+1, xs)

fun show l = show_help(0, l)

end
```

representation		number
[ ] = [0] = [0,0] "little endian"		0
[1]	1×2^0	1
[0, 1]	$0\times2^0 + 1\times2^1$	2
[1, 1]	$1\times2^0 + 1\times2^1$	3
[0, 0, 1]	$0\times2^0 + 0\times2^1 + 1\times2^2$	4

⋮

$$\begin{array}{r} 129 \\ +\;\;\;1 \\ \hline 130 \end{array}$$

$$\begin{array}{ccc} 2^2 & 2^1 & 2^0 \\ 1 & 0 & 1 \\ + & & 1 \\ \hline 1 & 1 & 0 \end{array} \qquad \begin{array}{r} = 5 \\ +1 \\ \hline 6 \end{array}$$

$$\begin{array}{l} [1, 0, 1] = 5 \\ + \; 1 \\ \hline [0, 1, 1] = 6 \end{array}$$