Modular design

- Hard to develop medium-to-large programs (>10k LOC)
  - Too complex to understand
  - May require domain-specific knowledge
- Must break into subsystems (modules) that can be designed separately
  - Must be possible to think about module behavior abstractly
  - Modules can be correct in isolation
  - Can identify source of failure
- Modules compose to make whole system
 Interfaces

Good modular design reduces cost
- Modules fit together
- Modules can usually be changed without affecting other modules (loose coupling)
- Design using local reasoning
- Test modules in isolation

• For each module $M$ provide interface $I$ describing what it provides
  - A contract between modules

 Language mechanisms

• ML: module = structure, interface = signature
• C: module = .c file, interface = .h file
• Java: module = class, interface = interface/javadoc
  - Deriving interface from code is dangerous
Example

• Module interface: usually collection of function declarations, plus some abstract data types (ADTs)
• Example: simple Java module for performing rational arithmetic

```java
interface Rational {
    Rational add(Rational);
    Rational multiply(Rational);
    ... implementation unspecified!
}
```

Interface and implementation(s)

```java
class Rational {
    int p; int q;
    Rational add(x, y) {
        return new Rational(
            x.p*y.q + x.q*y.p, x.q*y.q);
    }
}
```

2 possible implementations

```java
class Rational {
    int p; int q;
    Rational add(x, y) {
        return new Rational(
            (x.p*y.q + x.q*y.p)/gcd(x.q,y.q),
            x.q*y.q/gcd(x.q,y.q));
        int gcd(int q1, int q2) {...
    }
}
```
**Clients vs. Implementers**

- **Interface creates two roles:**
  - Client writes code that uses the module via interface
  - Implementer writes code that implements module

- **Obligations:**
  - Clients should not rely on anything not in interface
  - Implementer should provide everything in interface

- **Benefits**
  - Lower cost: client doesn’t have to know about module internals
  - Loose coupling: Can freely change module if interface unchanged
  - Can assign blame!

**What makes an interface good?**

- **Tension between client and implementer**
- **Narrow (small, simple)**
  - few exposed operations, types
  - easy to learn and understand
  - easy to implement & test
- **Complete - supplies all needed functionality**
  - But avoid unnecessary complexity!
  - Limit of short-term memory: ~7 things

<table>
<thead>
<tr>
<th>Narrow</th>
<th>Wide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited</td>
<td>Powerful</td>
</tr>
<tr>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>Loose coupling</td>
<td>Tight coupling</td>
</tr>
<tr>
<td>Easy to change</td>
<td>Hard to change</td>
</tr>
</tbody>
</table>
Loose coupling

Module

Interface/exports
(module as implementation)

"Outerface"/imports
(module as client)

• Keep both interactions simple!

Specifications

• Type declarations aren’t an adequate behavioral model; need specifications

  val sqr: float -> float
  - what does this do?

• Specification should include
  - Preconditions: requirements on the client to use operation/resource; implementer can rely on them
  - Postconditions: requirements on what the implementation does; client can rely on them

• Meaning: if the precondition is satisfied, then the postcondition will be satisfied

  (* Requires: x>=0 (precondition)
   Results: sqr(x) is the positive square root of x (postcondition) *)

  val sqr: float -> float
Using comments

• Goal: help people understand code
  - Client wants to understand what module does from interface
  - Implementer/maintainer wants to understand how module satisfies interface
  - Different kinds of comments!

• Interface (ML signature) contains:
  - function specifications
  - ADT overview

• Implementation (ML struct) contains:
  - algorithm explanations
  - data structure (representation) invariants
  - how to interpret data structure as abstract data type

Function specifications

• Useful to have a standard form: clauses
• Requires clause: preconditions on calling a function
• Results clause: postconditions
• Checks clause: preconditions with a description of failure behavior (default: throw exception to terminate program)
  (* sqr(x) is the positive square root of x
   Checks: x\geq 0 *)
• Modifies/effects clause: specifies all side effects of call (not needed in functional style)
  (* Effects: copy(a,b) copies all elements of a into b
   Requires: a and b are different hash tables *)

val copy: hashTable*hashTable -> unit
Avoiding overspecification

- Definitional spec describes external behavior
  
  ```
  fun find(a: int vector, y: int): int = ...
  (* Operational: loop j from 0 up to
   a.length-1 and return j if a[j] = y
   Definitional: return j such that a[j] = y *)
  ```

- Allows new implementations, easier for clients to use, easier to spot omissions

  - Definitional: Loose coupling, Easy to change
  - Operational: Tight coupling, Hard to change

- Nondeterministic specification: can have more than one result

Formal specifications

  ```
  (* let j = find(a,y)
   pre:  exists(i) Vector.sub(a,i) = y
   post: 0<=j & j<=Vector.length(a) &
         Vector.sub(a,j) = y
   modifies: none
  *)
  val find: int vector * int -> int;
  ```

- Automatic theorem prover can show that implementation satisfies spec
  - Java ESC
  - Larch C
Refinement

- Specification is **stronger** if it constrains behavior more, e.g.
  1. \( f(y) \) is an integer if \( y < 0 \)
  2. \( f(y) \) is an integer less than \( y \)
  3. \( f(y) \) is a prime factor of \( y \)

- A stronger specification **refines** a weaker specification (reduces set of poss. behaviors)
  - Stronger is not necessarily better...
- 1 refines 2 if: \( \text{pre}_2 \Rightarrow \text{pre}_1 \land \text{post}_1 \Rightarrow \text{post}_2 \)
- Refining specification will not break clients
- An implementation is a specification too!
- Correct implementation is a **refinement** of its specification: its behaviors are allowed

Hierarchical decomposition

- In well-designed code:
  - Bottom level code units are methods/functions (~1-100 LOC)
  - Modules have up to a couple of dozen ops
  - At most a couple of dozen modules to implement related functionality
- Top-level modules scale to ~10k LOC progs
- Modularity alone isn’t enough for large systems—need **hierarchy** of modules
Hierarchical decomposition

- Divide and conquer: must break large modules into smaller modules
- Multiple levels of hierarchy
- Good design if: only need to think about one module, one level at a time
- How to manage large-scale design?

Module dependency diagram

- When design of one module changes, other modules may need corresponding changes - how to identify?
- Module dependency diagram (MDD) gives high-level overview of system structure

```
client
  |  «kind»
  ↓
implementation
```

- Different kinds of dependency: «uses», «accesses», «references», «derives»
Keeping dependencies simple

• Too many dependencies or cycles: harder to debug, maintain, extend software

Design/implementation strategy

• Step 1: define interface that partitions problem
• Step 2: implement module

In hierarchy, what order?
Bottom-up implementation

- **Bottom-up**: develop modules before the modules that depend on them
- **Advantage**: catch key technology/performance issues early
- **Advantage**: always working code, easy testing
- **Disadvantage**: catch large-scale design flaws late

Top-down implementation

- **Top-down**: develop using modules before modules they depend on
- **Advantage**: get high-level design right from start, early prototype
  **Advantage**: easier to design interfaces well, quickly spec out system
- **Disadvantage**: harder to test until program complete