Abstraction and design

What is design, and why do you need one?

The ideas in this lecture

- Design is
  - high-level abstraction
  - what is left when you remove the implementation details
  - more art than science
- It is possible—and important—to think about design and implementation independently

Summary of last week

- We started with a problem
- We designed a solution to that problem
- We implemented that solution in
  - two different ways
  - two different languages
- The implementations share key abstractions

What are the key abstractions?

- We view the *input* as a stream of *tokens*.
- Each time we read a token, we check whether it will fit on the *line;* if not, we write the line on the *output* and clear it first.
- The above work is done in a function called *reformat*.

How might we describe a design?

- Describe each component, including
  - what it is supposed to do
  - what are its inputs and outputs
  - what state information the component maintains, and what it does

Machine-readable descriptions

- In some languages, you can say quite a bit about a component by writing incomplete code
  - In C++, you can write class definitions and not implement the member functions
  - In ML, you can write signatures and not write corresponding structures
- We will look at each language in turn

The Token class

- We will ignore the private data

```
enum Toktype {
    WORD, BREAK, END
};
class Token {
public:
    Token(istream*);
    Toktype type() const;
    string word() const;
};
```

The Line class

- Again, we will ignore the private data

```
class Line {
public:
    Line(int);
    void reset();
    bool canfit(string) const;
    void append(string);
    void print(ostream*) const;
};
```

The reformat function

- Here, we will ignore what the function actually does, and just declare it

```
void reformat
        (istream*, ostream*, int);
```

Is such a description a design?

- Not exactly—it says more about the implementation than we'd like—but
  - It provides valuable context for design
  - It gives information in a form that can be checked
    - We can compile these three declarations as shown, either independently or together
    - We can also compile (but not run) the full definition of the `reformat` function, given only these incomplete class definitions

Machine-checkable descriptions in ML

- We can write signatures without the corresponding structures
- We can even compile code that uses only the signatures

The TOKEN signature

```
signature TOKEN =
  sig
    datatype Toktype =
        WORD of string | BREAK | END
    val construct: TextIO.instream -> Toktype
  end
```

The LINE signature

```
signature LINE =
  sig
    type T;
    val construct: int -> T
    val reset: T -> T
    val canfit: T * string -> bool
    val append: T * string -> T
    val print: T * TextIO.outstream -> unit
  end
```

Using the design-level description

```
functor Reformat(structure L : LINE
                 structure T : TOKEN) =
  struct
    fun reformat(istrm, ostrm, n) =
       (* definition of reformat *)
  end
```

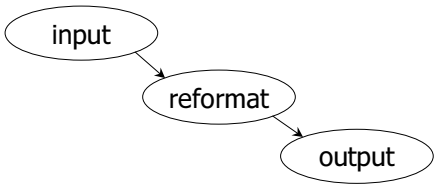
- The idea is that a functor lets us write programs that know only about the signatures of the data they use

The Reformat functor in full detail

```
functor Reformat(structure L : LINE  structure T: TOKEN) =
  struct
    fun reformat(istrm, ostrm, n) =
      let fun f(l) =
        case T.construct(istrm) of
          T.BREAK =>
            (L.print(l, ostrm); TextIO.output(ostrm, "\n");
             f(L.reset(l)))
        | T.WORD(w) =>
            if L.canfit(l, w)
            then f(L.append(l, w))
            else (L.print(l, ostrm); f(L.append(L.reset(l), w)))
        | T.END =>
            L.print(l, ostrm)
      in f(L.construct(n))
    end
  end
```

How else might we describe our design?

Use a sketch to describe the major components and the information flow between them



Even such a minimal design is useful

- From our little diagram, we can see that
  - All input goes into `reformat` and nowhere else
  - All output comes from `reformat` and nowhere else
- What else can we see?

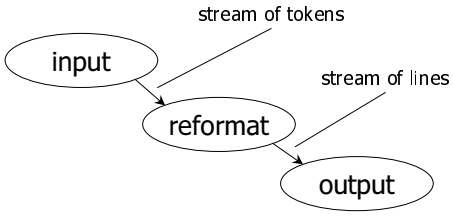
What else might we say about the design?

- The ovals in our diagram represent parts of the program
- The arrows represent data flow
- It is easy to think about the properties of the ovals
- What about properties of the arrows?

What does an arrow mean?

- Data moves from one part of a system to another …
- … in one direction or both …
- … according to a particular format
  - or structure
  - or protocol

So we might label the arrows also



Complementary tools

- Class definitions (signatures) make concrete some of the key abstractions in the program
- Simple diagrams show how these abstractions relate to each other
- Both can be made more concrete as the design progresses

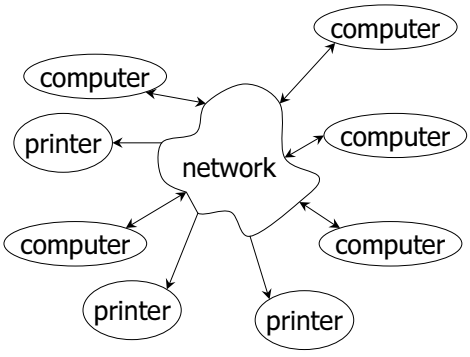
A larger design example

- I work in a building with hundreds of people, hence
  - hundreds of computers
  - dozens of printers
- How might we organize a printing system for such a building?

Desirable properties for a printing system

- Easy to use, for example:
  - A user of any computer should be able to print on any printer that the computer is equipped to handle
  - The system should be reliable
- Easy to administer, for example:
  - Adding or removing printers and computers should be easy

A very high level design



Is this a good design?

- Yes, as far as it goes
- But it doesn't go very far

Why is it a good design?

- It captures important aspects of how we would like printers to behave
  - We don't connect printers to computers; instead we connect them to "the network"
  - Neither computers nor printers care about each other's details, or the details of how the network works

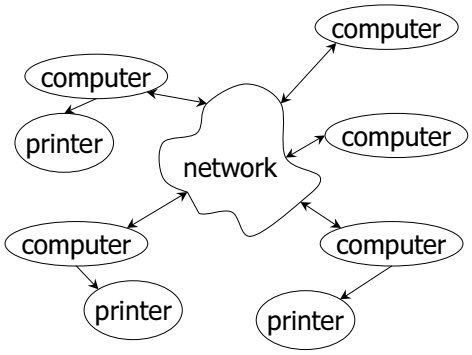
What details might we fill in?

- How to handle contention for printers
  - arbitration?
  - spooling?
- What protocol we use to talk to printers
  - PostScript?
  - TCP/IP?
- We will look only at the contention problem

How might we handle printer contention?

- Do nothing
  - Printers designed for network environments typically allow only one computer at a time to talk to them
- Have a spooling system (or many!)
  - Each printer has a computer designated to receive input for that printer
  - The computer resolves contention for that printer

Our revised design



Properties of this design

- It can solve the contention problem
  - as long as we do something about a spooling system
- It is less reliable than the previous design
  - if a spooling computer fails, it isolates the corresponding printer
- It has become harder to administer

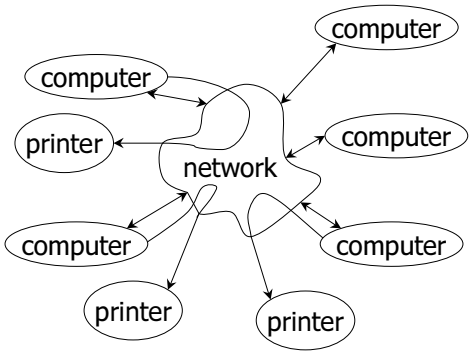
Whence the administrative difficulties?

- Printers no longer connect to the network; instead, they connect to computers
  - There might be distance limitations
  - Changing a spooler requires moving cables
  - How do computers talk to printers if not through a network?

Making the design more practical

- We can still associate a spooling computer with every printer …
- … but we can let the computer talk to the printer over the network anyway

Our revised revised design



What might we do next?

- Figure out how a spooler works
- Think about how computers find printers
- How might we go about solving these problems?

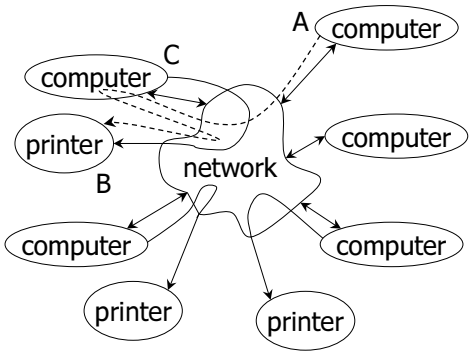
Use cases

- An increasingly popular idea in software design is the notion of *use cases:*
  - Think about ways in which your design might be used
  - Track the information flow through the design, noting what happens at each stage
  - Use what you learn to refine the design

Example of a use case

- User of computer A wants to print on printer B, spooled on computer C
- What happens when we try this?

Use case diagram



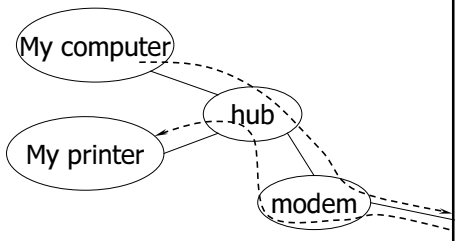
What can we learn from this example?

- Whatever we want to print makes two trips through the network:
  - one from computer A to computer C
  - one from computer C to printer B
- Is this a problem?
  - Not if the network is fast enough…
  - …and a network should be much faster than a printer, right?

Finding a performance problem

- When is a network slower than a printer?
- When it is connected by a telephone line! We have failed to account for the fact that parts of the network are in people's homes.
- Suppose, for example, that computer A and printer B are both in my house…

A user-centric view of the system



How might we solve this problem?

- Make my computer spool for my printer
  - This might fail if other aspects of the design require spoolers to be always up
- Have two spoolers for my printer: One for me, and one for everyone else
  - This will work most of the time, but the printer might have to resolve contention
- Either choice affects the whole design

The moral of the story

- Design matters!
- You can learn a lot about design, even with simple tools, by experimenting
- You can get the computer to help you by writing high-level parts of the program and compiling it—even if you can't run it

Homework (due Monday)

- Design a system to support cashiers in a convenience store
- The design should be detailed enough to make it clear how to handle several usual and unusual use cases…
- …but it should not be more detailed than necessary (no more than two pages or so)

Examples of usual use cases

- A customer comes in, buys some items, and pays with cash or a credit card
- A cashier goes on duty, and says how much cash is in the register.
- A cashier goes off duty; how much cash should be in the register?

Examples of unusual use cases

- A customer buys some items, and
  - pays for part of the order with a credit card and the rest with cash, or
  - discovers a cash shortage and leaves some items behind after they've been rung up
- A customer wants to return an item
- A cashier wants to transfer some cash to another cashier

What to hand in

- An overall description of the system
- For each component,
  - What information is stored there
  - How it connects with other components
  - What information goes over the connections
- Examples of use cases it handles, and how it handles them