Lecture 5: Decomposition and Abstraction

Decomposition

- When to decompose
- Identifying components
- Modelling components

Abstraction

- Abstraction by parameterization
- Abstraction by specification
- Pre-conditions and Post-conditions

Decomposition

Tackle large problems with "divide and conquer"

Decompose the problem so that:

- Each subproblem is at (roughly) the same level of detail
- Each subproblem can be solved independently
- The solutions to the subproblems can be combined to solve the original problem

Advantages

- Different people can work on different subproblems
- Parallelization may be possible
- Maintenance is easier

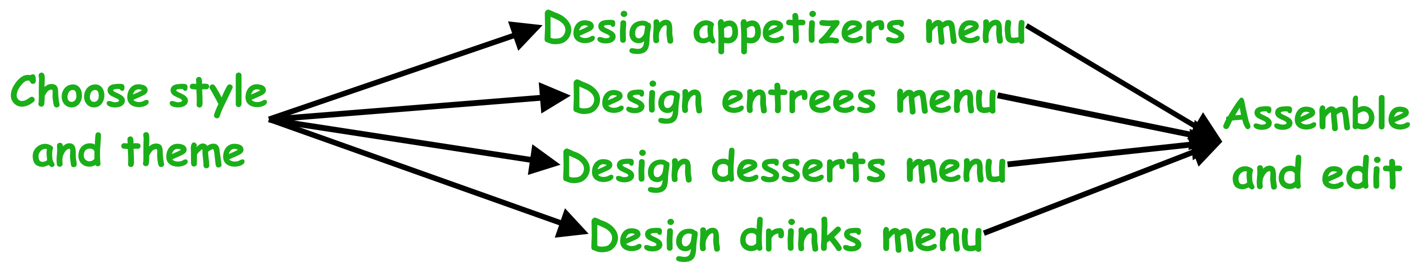
Disadvantages

- Solutions to the subproblems might not combine to solve the original problem
- Poorly understood problems are hard to decompose

Decomposition Examples

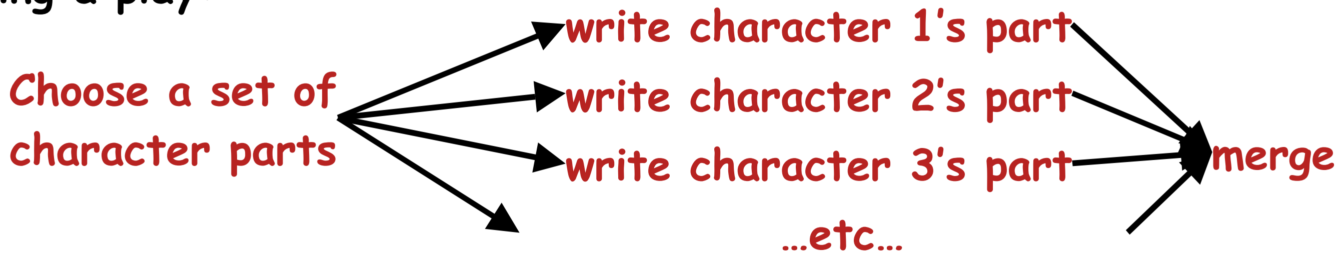
Decomposition can work well:

E.g. designing a restaurant menu



Decomposition doesn't always work

E.g. writing a play:

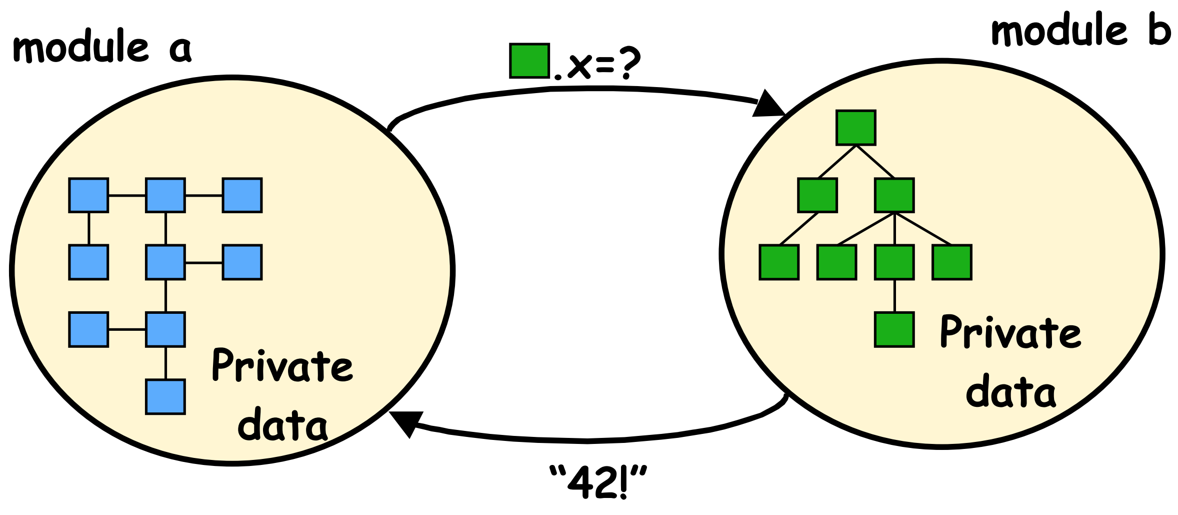


Decomposition isn't always possible

for very complex problems (e.g. Managing the economy)
for impossible problems (e.g. Turning water into wine)
for atomic problems (e.g. Adding 1 and 1)

Step 1: Identify components

- a good decomposition minimizes dependencies between components
  - coupling - a measure of inter-component connectivity
  - cohesion - a measure of how well the contents of a component go together
- information hiding
  - having modules keep their data private
  - provide limited access procedures
  - this reduces coupling

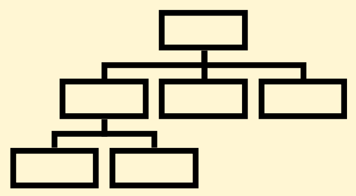


How to decompose

How to decompose (cont.)

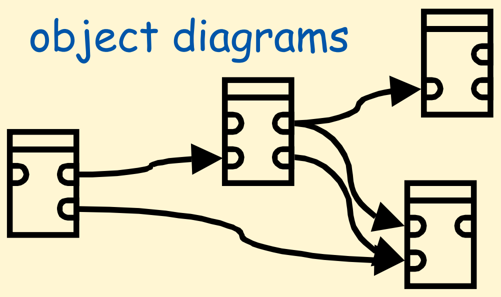
Step 2: Model the components

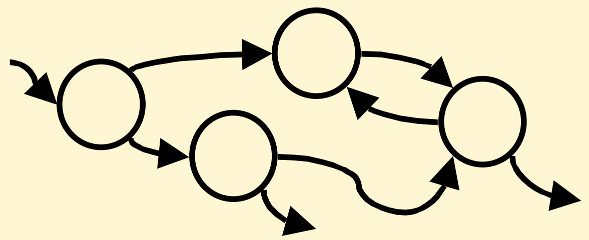
At the design level



structure charts

object diagrams





dataflow diagrams

At the coding level

procedure declarations

```
float sqrt(int);
```

procedure specifications

```
float sqrt(int x){
/* requires: x is a positive integer
   effects: returns an approximation
   of the square root of x to within
   ±10⁻⁴ */
```

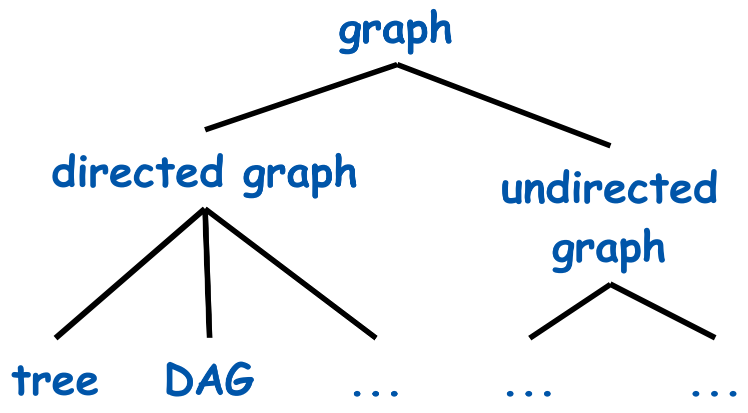
Abstraction

Abstraction is the main tool used in reasoning about software

Why? It allows you to:

- ignore inconvenient detail
- treat different entities as though they are the same
- simplify many types of analysis

Example abstractions



A file

$\updownarrow$

A sequence of bits on a disk

set membership

$\Updownarrow$

A program that takes an integer and a list returns the index of the first occurrence of the element or null if the element does not occur in the list

example

Can I replace A with B?

A

```
found = false;
i = lowbound(a);
while (i < highbound(a)+1){
  if (a[i] == e) {
    z = i;
    found = TRUE;
  }
  i = i + 1;
}
```

B

```
found = false;
i = highbound(a);
while (i > lowbound(a)-1){
  if (a[i] == e) {
    z = i;
    found = TRUE;
  }
  i = i - 1;
}
```

if we could abstract away all the detail…

Using Abstraction

Abstraction can help with Decomposition

e.g. To manage the economy, try focussing on some abstracted features such as inflation, growth, GDP, etc.

Abstraction allows us to ignore inconvenient details

In programming:

Abstraction is the process of naming compound objects and dealing with them as single entities

(i.e. ignoring their details)

Abstraction doesn't solve problems…

…but it allows us to simplify them

Abstraction by Parameterization

The program fragment:

```
x * x - y * y
```

computes the difference of the squares of two specific variables, x and y.

The abstraction:

```
int squares (int x, int y) {
  return(x * x - y * y);
}
```

describes a set of computations which act on any two (integer) variables to compute the difference of their squares

Note: locally the variables are called x and y for convenience

The specific computation:

```
result = squares(big, small);
```

uses the abstraction 'squares' on two specific variables ('big' and 'small')

Abstraction by Specification

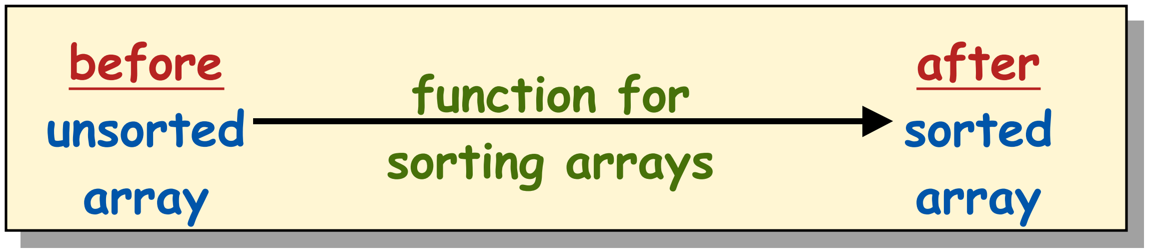
Abstraction by parameterization…

…allows us to express infinitely many computations

…but does not tell us about the *intention* of those computations

We need to capture the intention

e.g. consider what is true before and after a computation



we can abstract away from a computation (or a plan, program, function, etc) by talking about what it achieves

specification

this function can be used whenever we have an array. After it is applied, the array will be sorted into ascending order

Pre-conditions and Post-conditions

The two forms of abstraction are complementary

parameterization allows us to perform a computation on any arbitrary variables (values)

specification allows us to ignore how it is done

Unfortunately…

only abstraction by parameterization is built into our programming languages

as function (procedure) definitions

We can overcome this using comments:

```
int strlen (char s[]) {
  /* precondition: s must contain a character array,
        delimited by the null character;
     postcondition: returns the length of s as an integer;
  */
  int length = 0;
  while (s[length])
    length++;
  return(length);    }
```

Summary

Decomposition allows us to simplify difficult design tasks

A good decomposition

- minimizes coupling between components
- maximizes cohesion within components
- permits information hiding

Methods provide...

- ... techniques for decomposing problems
- ... notations for describing the components

Abstraction allows us to ignore detail

- by parameterization: allows us to describe and name sets of computations
- by specification: allows us to ignore how the computation is done

References

van Vliet, H. "Software Engineering: Principles and Practice (2nd Edition)" Wiley, 1999.

- Chapter 11 provides an introduction to the concepts in this lecture, especially section 11.1. However, van Vliet does not go into much detail about documenting procedural and data abstractions in the style I use in this and the next two lectures. For this you'll need:

Liskov, B. and Guttag, J., "Program Development in Java: Abstraction, Specification and Object-Oriented Design", 2000, Addison-Wesley.

- See especially chapters 1 and 3. I draw on Liskov's ideas extensively for advice on program design in this course. The commenting style I use ("requires", "effects", etc) is Liskov's. If you plan to do any extensive programming in Java, you should buy this book. If you don't buy it, borrow it and read the first few chapters.