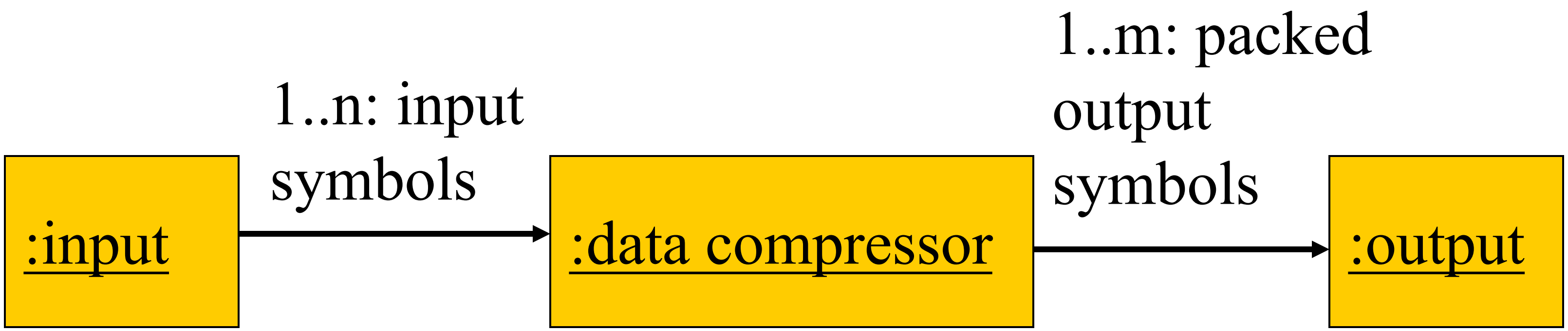
CPUs

- Example: data compressor.

Goals

- Compress data transmitted over serial line.
  - Receives byte-size input symbols.
  - Produces output symbols packed into bytes.
- Will build software module only here.

Collaboration diagram for compressor

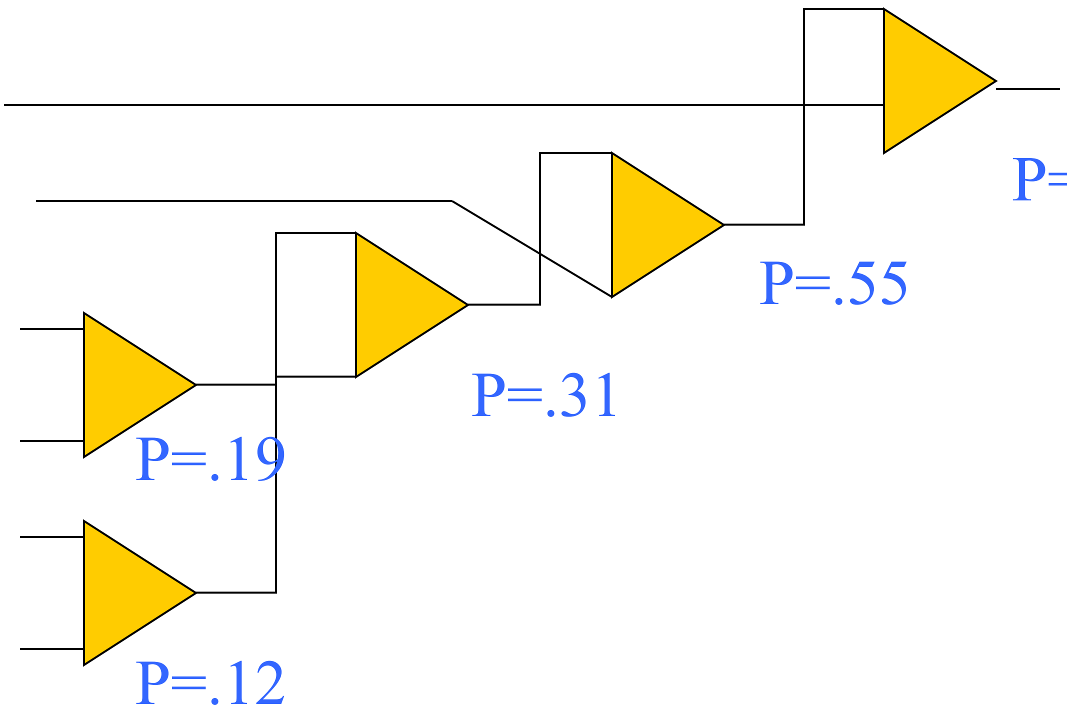


Huffman coding

- Early statistical text compression algorithm.
- Select non-uniform size codes.
  - Use shorter codes for more common symbols.
  - Use longer codes for less common symbols.
- To allow decoding, codes must have unique prefixes.
  - No code can be a prefix of a longer valid code.

Huffman example

character	P
a	.45
b	.24
c	.11
d	.08
e	.07
f	.05



Example Huffman code

- Read code from root to leaves:

a	1
b	01
c	0000
d	0001
e	0010
f	0011

Huffman coder requirements table

name	data compression module
purpose	code module for Huffman compression
inputs	encoding table, uncoded byte-size inputs
outputs	packed compression output symbols
functions	Huffman coding
performance	fast
manufacturing cost	N/A
power	N/A
physical size/weight	N/A

Building a specification

- Collaboration diagram shows only steady-state input/output.
- A real system must:
  - Accept an encoding table.
  - Allow a system reset that flushes the compression buffer.

data-compressor class

data-compressor

buffer: data-buffer
table: symbol-table
current-bit: integer

encode(): boolean,
data-buffer
flush()
new-symbol-table()

data-compressor behaviors

- encode: Takes one-byte input, generates packed encoded symbols and a Boolean indicating whether the buffer is full.
- new-symbol-table: installs new symbol table in object, throws away old table.
- flush: returns current state of buffer, including number of valid bits in buffer.

Auxiliary classes

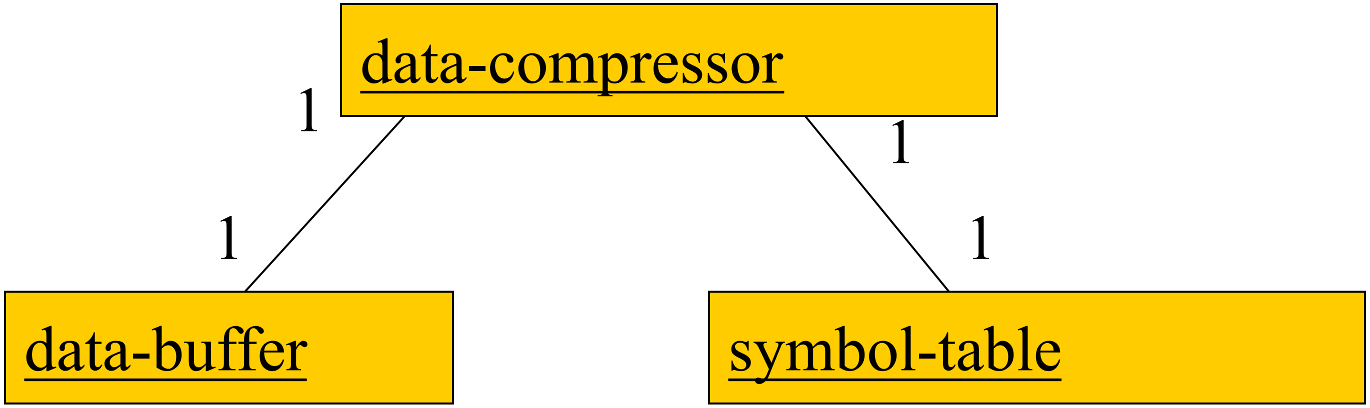
data-buffer
databuf[databuflen] : character len : integer
insert() length() : integer

symbol-table
symbols[nsymbols] : data-buffer len : integer
value() : symbol load()

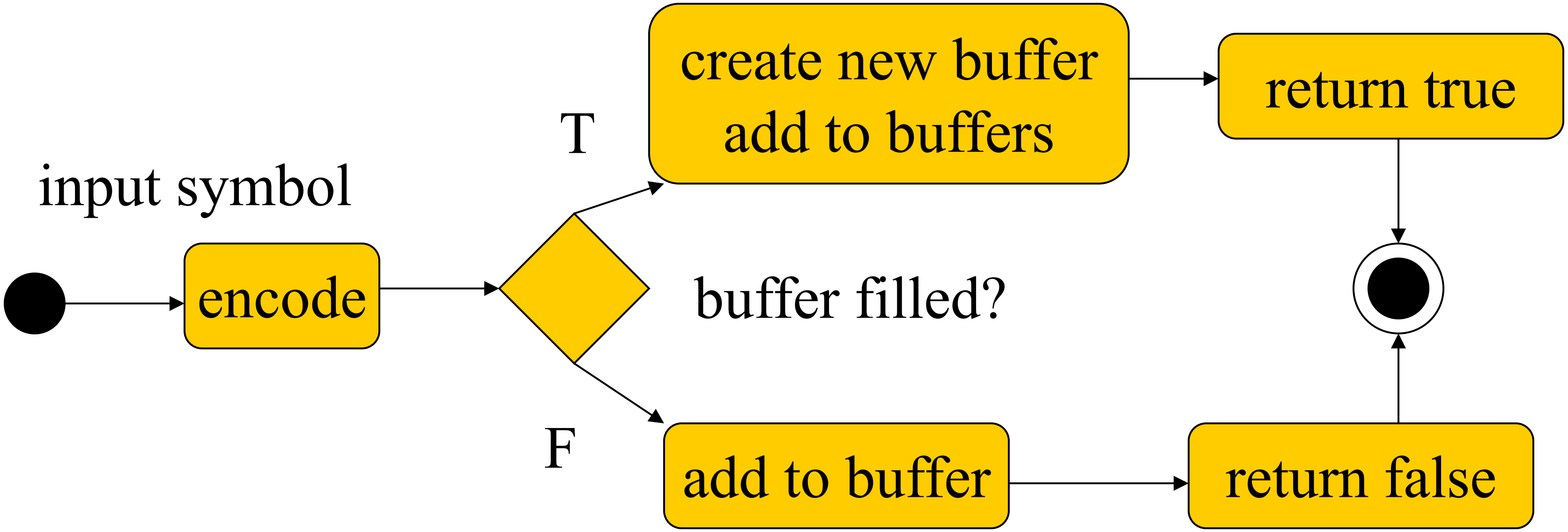
Auxiliary class roles

- data-buffer holds both packed and unpacked symbols.
  - Longest Huffman code for 8-bit inputs is 256 bits.
- symbol-table indexes encoded verison of each symbol.
  - load() puts data in a new symbol table.

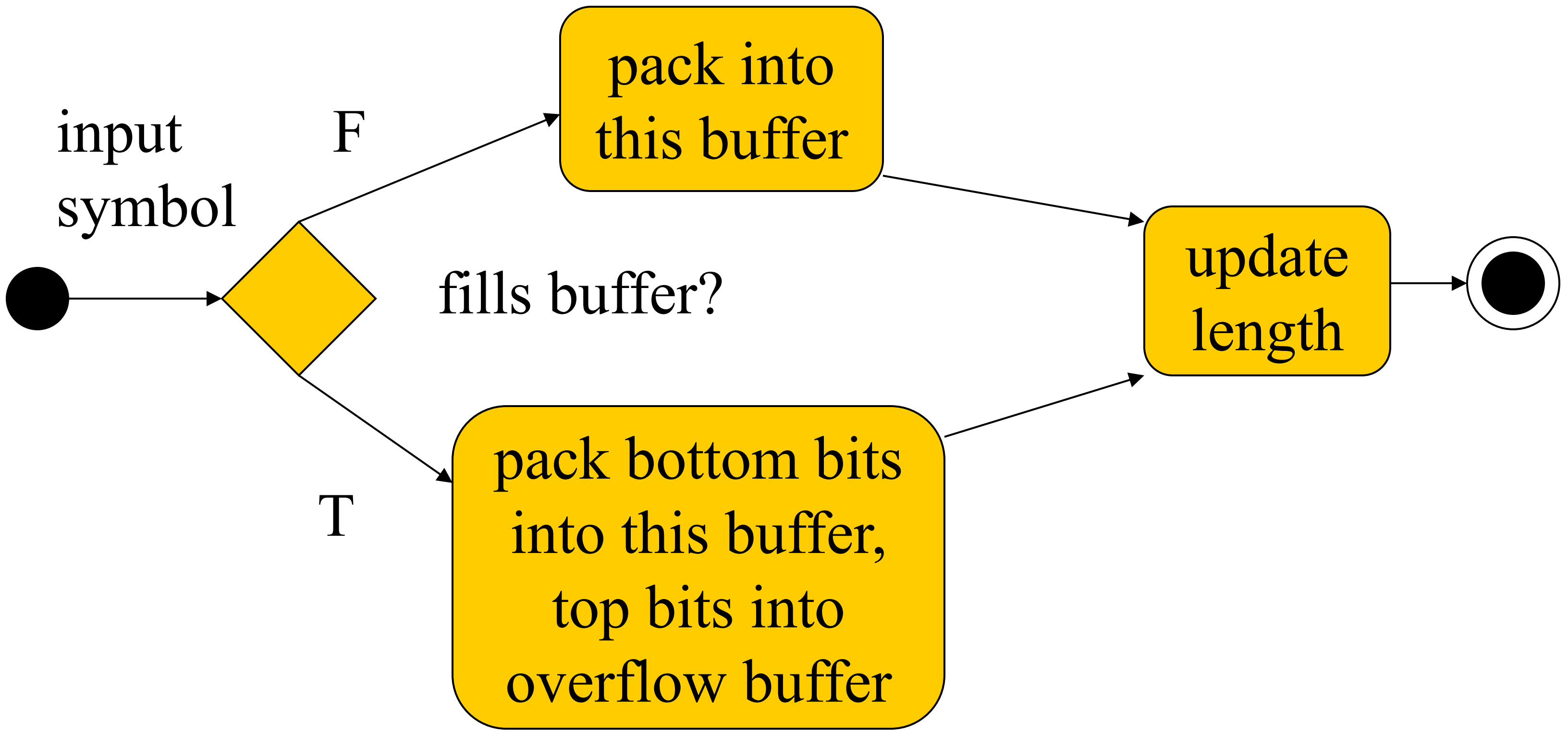
Class relationships



Encode behavior



Insert behavior



Program design

- In an object-oriented language, we can reflect the UML specification in the code more directly.
- In a non-object-oriented language, we must either:
  - add code to provide object-oriented features;
  - diverge from the specification structure.

C++ classes

```
Class data_buffer {
    char databuf[databuflen];
    int len;
    int length_in_chars() { return len/bitsperbyte; }
public:
    void insert(data_buffer,data_buffer&);
    int length() { return len; }
    int length_in_bytes() { return (int)ceil(len/8.0); }
    int initialize();
    ...
```

C++ classes, cont'd.

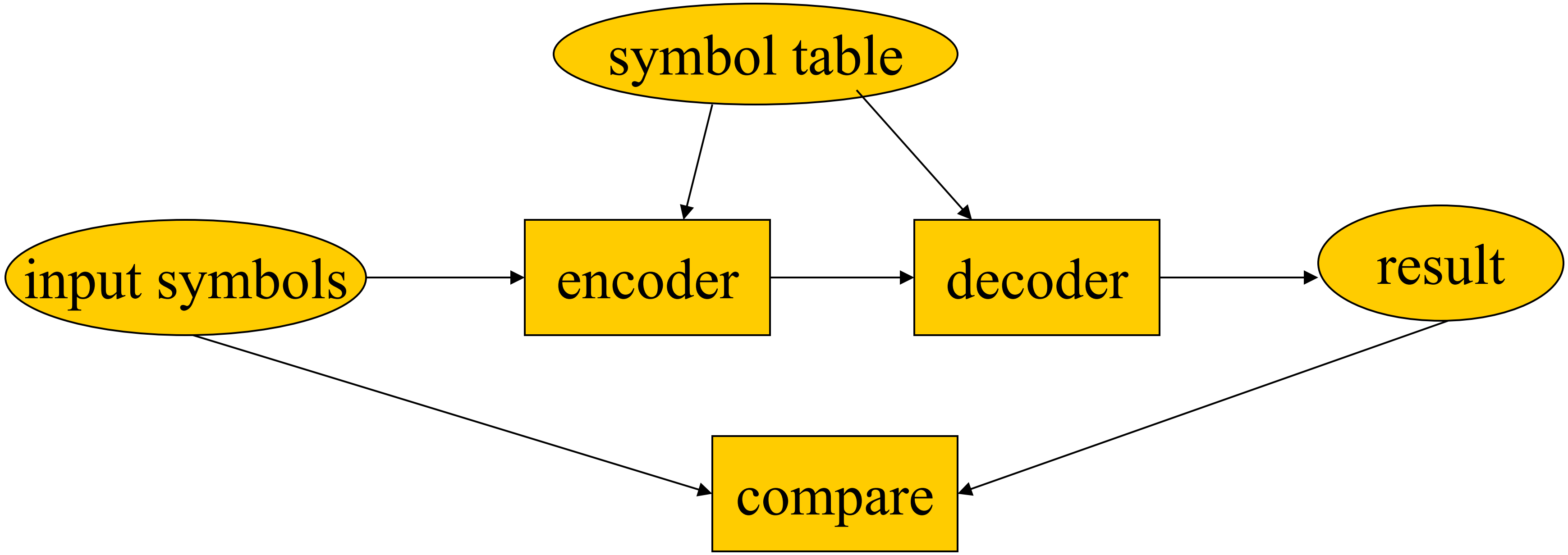
```
class data_compressor {
    data_buffer buffer;
    int current_bit;
    symbol_table table;
public:
    boolean encode(char,data_buffer&);
    void new_symbol_table(symbol_table);
    int flush(data_buffer&);
    data_compressor();
    ~data_compressor();
}
```

C code

```
struct data_compressor_struct {
    data_buffer buffer;
    int current_bit;
    sym_table table;
}
typedef struct data_compressor_struct data_compressor,
    *data_compressor_ptr;
boolean data_compressor_encode(data_compressor_ptr
    mycmptrs, char isymbol, data_buffer *fullbuf) ...
```

Testing

- Test by encoding, then decoding:



Code inspection tests

- Look at the code for potential problems:
  - Can we run past end of symbol table?
  - What happens when the next symbol does not fill the buffer? Does fill it?
  - Do very long encoded symbols work properly? Very short symbols?
  - Does flush() work properly?