
THIS COMPUTER TAX PROGRAM SURE SAVES A LOT OF TIME
CHARLES BOLLINE '91/5
HOW'S IT LOOK?
WE EARNED $35B IN 1990 AND WE OWE $68,576 ?!

modular design

- decomposing the problem into smaller "pieces"
  divide and conquer strategy
  a smaller problem is easier to solve
- test/debug (smaller, independent code segments)
- reuse of code
  "off the shelf" components
  reliable since it is widely used
- easier to read & understand quickly (time is $$$$)
  abstraction, information hiding
- organization of the program (lots of code)
- modify/ maintenance is easier and faster (why?)
  (independence – loosely coupled)
- more independence of parts allows work in teams

"common sense"

- Understand the problem from the execution perspective.
  (Eg. What should the screen look like while under execution?)
- Develop sample inputs (files) of reduced size using sample lines from the testing files provided.
  What exactly would the output be?
- One could begin by coding the user input part. One could also begin by initializing small data sets until you get some functionality then add the user input later. (We don't want a test program that needs to be tested – keep it simple and grow it.)
- Prepare a list of test inputs and respective outputs (test oracle). (test boundaries of the data range—test some out of bounds).
- Develop easy functions that are fairly generic that you're sure you will need by just reading over the problem.
- Isolate parts of problem that accomplish an intermediate step. (Eg. Just output all the tokens to the screen to see if you're identifying them properly before you bother with loading arrays and duplicate handling.) You should always have the program in a state that will execute albeit perhaps not totally functional. "G – R – O – W" the program.
- Save intermediate step versions for future testing in case you need to go back to that part and make revisions.

Decomposition

How do we determine what our modules will be?

- several techniques exist for designing program parts
- we talked about “common sense” where we can easily identify some modules
- multiple decomposition techniques are often used together
- the rest of this document discusses top-down design as the exclusive decomposition strategy

Top-down design

Focus is on the tasks or procedures that need to be performed to solve a problem that has been specified. From the specified problem or task, chunk the task into smaller chunks of tasks. Then take each "smaller" chunk and make even smaller chunks. Continue to refine the chunks of tasks until the task is simple enough to do without further chunking.

A programming task can be defined as a function or method. So, we begin with the specifications for the "larger" problem and code the entire problem in one function (summary). To simplify the coding of any function , specify and call (nonexistent) functions to handle parts of the job. This is coding at a high level of abstraction (more abstraction) since we are not concerned at all about the code for the called functions (procedural abstraction). We are concerned only about the one function we are currently developing.

Then repeat this process (stepwise refinement) for coding each function specified working down to lower levels of abstraction until the function code can be completed without the need to specify and call any more functions. This results in a layering of abstraction levels. A structure chart is a view of the hierarchical layering of the function calls. At each successive lower level in the chart, there is less abstraction.

tasks FIT together

As we develop a function's code we need:

1. **algorithm** logic
   flow of control (sequencing) using coded statements(if, loops, etc)
   using function calls and noting data flow

2. **specification** for each nonexistent function call:
   what the function's tasks are
   (**not** **how** it accomplishes the tasks– no code.)

   Make **pre** and **post** conditions clear within:

   - description of the task(s)
   - parameter description (data flow)
   - possible return statement (data flow)
   - possible throws list (eg. indicating a precondition has failed)
     describes fault tolerance …

Crazy 8's : top-down design

What functions would be needed and exactly how would they fit together?

Need flow of control since sequencing is part of "fitting" together to accomplish the tasks specified.

Need to determine data flow between methods too.

Problem statement (and directions) for Crazy 8's was provided in class. This is a statement of the specifications. Remember that programmers must be vigilant about clarity and completeness of specifications. This is a large part of any programming job! Ask a lot of questions. Often, specification issues arise at implementation time.

Crazy Eights function: Main

setup post: deck, dealer, computerscore, userscore, maxptspergame

do {

shuffleCards pre: deck
post: deck

dealCards pre: deck, dealer
post: deck, computerhand, userhand, discard, namedsuit, currentplayer

playOneSet pre: deck, computerhand, userhand, discard, namedsuit, currentplayer
post: computerhand, userhand

scoreOneSet pre: computerhand, userhand, computerscore, userscore, dealer
post: computerscore, userscore, dealer

}while (! gameOver) pre: computerscore, userscore, maxptspergame
post: return boolean

shutdown pre: computerscore, userscore, maxptspergame

Detailed Specifications for each function called at this level.
(4 are shown here as examples)

```
/*************************************************************
setup  post:  deck, dealer, computerscore, userscore, maxptspergame
Initializes a new deck of cards, establishes the dealer by randomly choosing
between the computer or the user, initializes both player scores to 0, and
prompts the user to establish the maximum points for the overall game.
*/
void setup ( )
{
}
```

```
/*************************************************************
shuffleCards  pre: deck (has been created)
              post: deck
Randomly arranges the cards in the deck.
*/
void shuffleCards()
{
}
```

```
/*****************************************************************
```

dealCards pre: deck, dealer

post: deck, computerhand, userhand, discard, namedsuit, currentplayer

Deals 7 cards to each hand and one for the discard card. If the discard happens to be an 8, then the first player (not the dealer) establishes the namedsuit. The currentplayer is set to be the player that is *not* the dealer.

```
*/
void dealCards()
{
}
```

```
/*****************************************************************
```

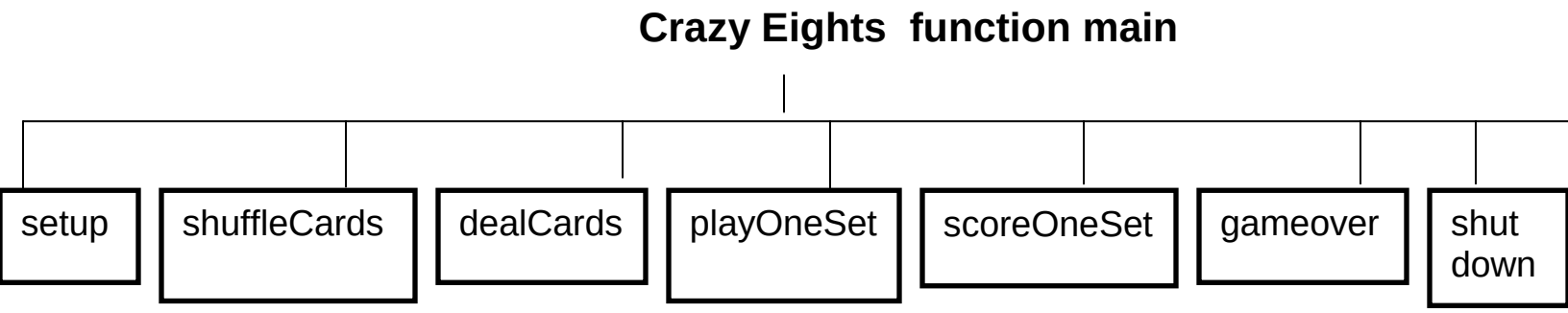
playOneSet pre: deck, computerhand, userhand, discard, namedsuit, currentplayer

post: computerhand, userhand

Players take turns until a player runs out of cards, or until the deck empties AND neither player can discard a card. Both hands are returned once the set is completed.

```
*/
void PlayOneSet()
{
}
```

Structure Chart **(hierarchy of function calls)**



Repeat the same process with every function.
Begin with its specification, then write its code.
If additional function calls are required, then specifiy them carefully.

Pick: playOneSet

```
/*************************************************************
```

playOneSet pre: deck, computerhand, playerhand, discard, namedsuit, currentplayer

post: computerhand, playerhand

Players take turns until a player runs out of cards, or until the deck empties AND neither player can discard a card. Both hands are returned once the set is completed.

```
*/
void playOneSet(…parameters needed for pre-conditions ?)
{   boolean computerpass = false,  playerpass = false;
    do{
        switch(currentplayer) {
            case user:  doUserTurn  pre: userhand, deck,
                                         discard, namedsuit, userpass
                                post: userhand, deck,discard,
                                      namedsuit, userpass
            case computer: doComputerTurn
                                pre: computerhand, deck, discard,
                                     namedsuit, computerpass
                                post: computerhand, deck discard,
                                      namedsuit, computerpass
        }//switch
        getNextPlayer       pre:currentplayer
                            post:currentplayer
    }while ( ! isSetOver )  pre: computerhand, playerhand,
                                 computerpass, playerpass
                            post:  returns boolean
}//playOneSet
```

Detailed Specifications for each function called in playOneSet.
(1 example is shown here)

/**

isSetOver pre: computerhand, userhand,
computerpass, userpass

post: returns boolean

Returns true if either the computerhand or the userhand is empty.
If both hands still have cards, then true is returned only if both the computerpass and the userpass are true, false otherwise.

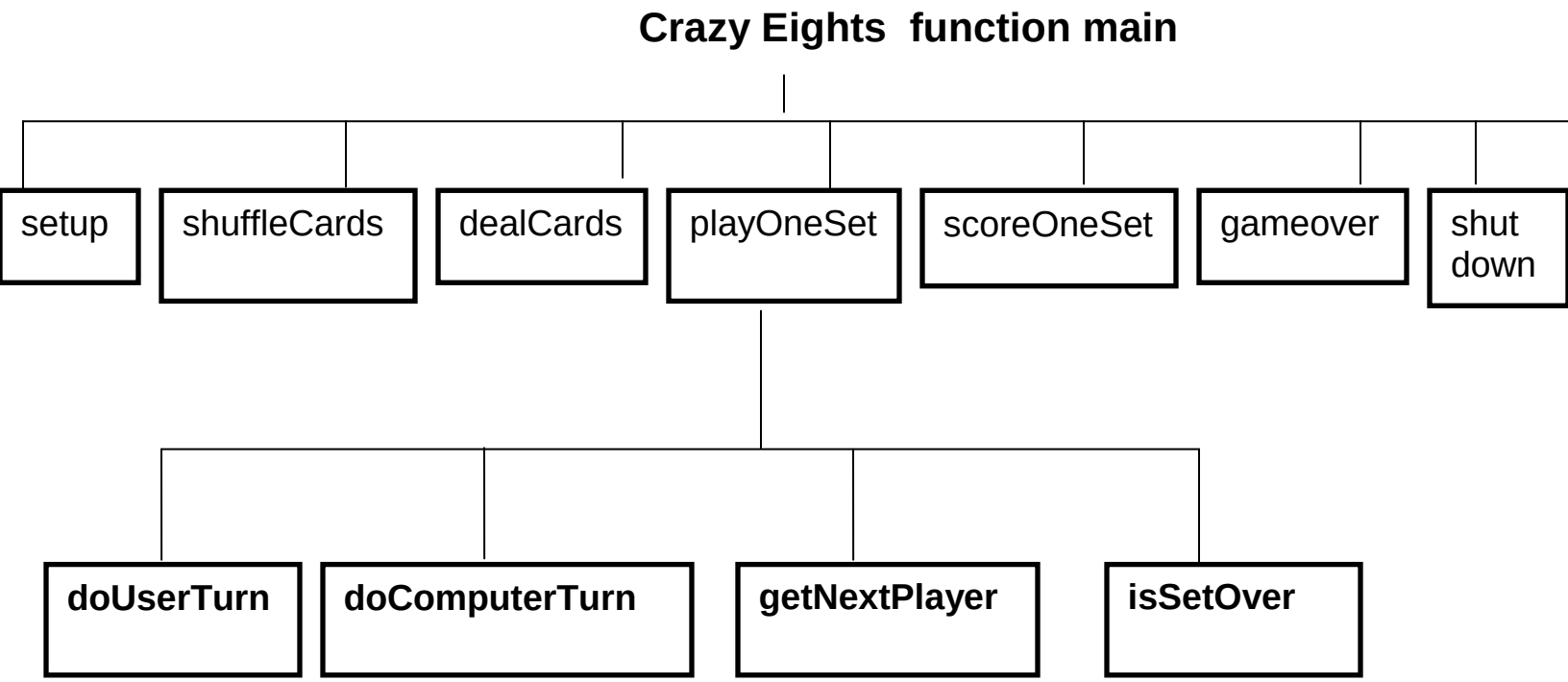
*/

boolean isSetOver(parameters needed for preconditions?)

(Once all four called functions in playOneSet have been specified,

then the structure chart is updated.)

Structure Chart (hierarchy of function calls)



What is the next step in this design process?

Craps is a dice rolling game of chance. The program will show the rules of the game and ask if you want to play. Here are the rules:

First roll of 7 or 11, you **win** outright.

First roll of 2,3, or 12, you **lose** outright.

First roll of 4..10, that number becomes your **point number** and you keep rolling until you either roll a 7 or you roll your point number again. If you roll your point number first, you win; if you roll a 7 first, you lose.

main:

```
while ( wantsToPlay() )   //prompts user, returns boolean….
    playCraps();          //plays one complete game
```

You design playCraps()

playCraps: What is wrong with this design?? Critique:

rolldice()
checkdice()
rolldice()
checkdice()

playCraps:

```
firstRoll = rollDice(2);
String message = "first roll = " + firstRoll + "\n";
if ( isWinner(firstRoll))
   showMessage(message + "You WIN!!");
else if ( isLoser(firstRoll))
    showMessage(message + "You LOSE!!");
else
    finishGame(firstRoll);
```

Not done yet!!! Complete specifications for each function:

description of what the method does
preconditions:
postconditions:
parameters:
returns:
throws:
sample call:

What is the specification/documentation for these method calls?

"Separation of concerns"

Specification (Interface)

"What" is the task???
Describe the task.
This is the "contract".

Details: pre/post conditions etc.
Info needed to *use* the module.
Documentation…

Implementation

"How" to accomplish the task?
Make the computer function as described in the "contract".

Details: coded algorithms, data types, expressions, data structures, etc.
Documentation…

- information hiding
  The programmer **using** the module need not see the code.
  Provide abstraction for the programmer for ease (hide some details)
- protection for the code (from unintended or malicious intents)
  Some languages provide features to enforce info hiding/access
  eg. "private" variables in classes, local variables in functions
- can change the implementation without changing the programs that call it (typically extend, not replace -- why?)

How do we physically accomplish this abstraction barrier?
We separate the specification from the implementation into separate files.

javadoc tool:
Javadoc is a special program that creates an .html file from special comments in a .java file.
.html file is the specification
.java file is the implementation

C++:
header file .h is the specification
.cpp file is the implementation

some "javadoc"-like tools (DOC++)