Stepwise Refinement

Steven Zeil

September 29, 2013

Contents

1 Stepwise Refinement **2**
1.1 Pseudo-Code 2
1.2 Stepwise Refinement 3

2 Top-Down Design **14**
2.1 Example: Top-Down Auction 14

1 Stepwise Refinement

Stepwise Refinement

Stepwise refinement is a basic technique for *low-level design.*

Invented by Niklaus Wirth in 1971, it may be the oldest systematic approach to software design still in use. But it remains quite relevant to modern programming.

Stepwise refinement is a discipline of taking small, easily defended steps from a very generic view of an algorithm, adding a few details at each step, until the path to an actual program becomes clear. It's a great answer to that awful, hollow feeling that you may get when staring at a blank sheet of paper or editor window and wondering, "how in the world am I supposed come up with a program to do ...?"

. . . . . .

1.1 Pseudo-Code

Fundamental to the process of stepwise refinement is *pseudocode*, a low-level design tool or notation hat allows us the flexibility to write about algorithms that we don't entirely understand yet.

Pseudo-Code

Also known as "structured English"(at least, in English-speaking countries), pseudocode uses

- Programming Language-like control structures
  - Sequence – one thing after another
  - Selection – `if`
  - Iteration – `while`, `for`
- natural language (English) descriptions of functions and data
- program-style indentation and `{ }`for grouping and readability

. . . . . .

Pseudo-Code Example

```
void primaryElection() {
  read all candidates;
  read # of states conducting primaries;
  for each such state {
      read votes cast in this state;
      assign delegates to candidates;
    }
  for each candidate c {
      print report for candidate(c);
    }
}
```

. . . . . .

1.2 Stepwise Refinement

Stepwise Refinement is Stepwise

The heart of stepwise refinement is the idea that

- pseudo code statements are repeatedly expanded
  - until we reach a level of detail where the translation into true programming language code is “obvious”.

. . . . . .

Procedure

1. Start with a general statement of the problem.
2. Pick any statement whose translation into code is non-obvious.
3. Expand that statement into 2 or more steps.

4. Repeat steps 2-4 until all statements can be easily coded.

The essential elements here are that you

- Expand *one* statement at a time
  - replacing it by *two or more*, more detailed, statements
- and the replacements must, together, account for the *whole* statement being replaced

. . . . . .

Example: The Auction Program

Start with a general statement:

```
//Determine winners of daily auction;
```

Pick a statement that needs expansion. (OK, there's only one choice.)

```
// Determine winners of daily auction;
//* initialize auction data;
//* determine Winners;
//* clean up;
```

The first expansion is kind of easy. Almost any algoithm can safely be said to start with initialization, end with some kind of cleanup, with all the hard work happening in between.

But there's nothing wrong with easy steps. In fact, small easy steps are preferred because you can usually convince yourself more easily that you have not made a mis-step and forced yourself into a direction where you will never be able to complete a working design.

- Although not absolutely necessary, I recommend writing your pseudocode as comments.
  - Why? Because this way, when you are done, you can insert the actual code in between lines of pseducode, resulting ina very well-documented algorithm that will explain your thought process along with the actual code itself.
- The *'s are a convention to help indicate the *depth* of the expansion.

- Our original statement "Determine winners of daily auction" was our level-0 description of the program.
- When you expand a level-0 statement, you get two or more level-1 statements. When you expand a level-1 statement, you get two or more level-2 statements.
- The number of '*' in the comment prefix indicates the level.
  So these new statements are all level-1, and we can see at a glance which statement they must be an expansion of.

.....

determining the winners

Again, pick a statement that needs expansion

```
// Determine winners of daily auction;
//* initialize auction data;
//* determine Winners;
//** for each item {
//**   resolve the auction for item;
//**}
//* clean up;
```

We could have picked any of the 3 level-1 statements. But I had a pretty good idea that determining the winners means that we needed to determine a winner for each item.

- Pick a statement that needs expansion

.....

Initializing the auction data

```
// Determine winners of daily auction;
//* initialize auction data;
```


```
//** read items;
//** read bidders;
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve the auction for item;
//** }
//* clean up;
```

- This is a pretty safe bet, given the info provided in the auction program description.
- It should be a bit more apparent, now, how the * convention aids us in keeping track of which statements are expansions of which other statements.

.....

Resolving the auction for one item

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//** read bidders;
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//***   get winning bid for this item;
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
```


```
//***   } else {
//***      print notice of no winner;
//***   }
//** }
//* clean up;
```

.....

Getting the winning bid

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//** read bidders;
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   for each bid for this item {
//****      if (bid is the best so far) {
//****         remember this bid as highest so far;
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
```


```
//***      print notice of no winner;
//***   }
//** }
//* clean up;
```

.....

Best bid so far

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//** read bidders;
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   for each bid for this item {
//****      if (bid is the best so far) {
 //*****      if (bid is in time && //*****          bid is higher than highestBid && //*****          bid
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
//***      print notice of no winner;
//***   }
//** }
```


```
//* clean up;
```

.....

Reading items

```
// Determine winners of daily auction;
//* initialize auction data;
 //** read items;
 //*** read #items;
 //*** for (int i = 0; i < #items; ++i)
 //***   read item[i];
  //** read bidders;
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   for each bid for this item {
//****      if (bid is the best so far) {
//*****      if (bid is in time &&
//*****          bid is higher than highestBid &&
//*****          bidder has enough money) {
//****         remember this bid as highest so far;
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
```


```
//***      print notice of no winner;
//***   }
//** }
//* clean up;
```

.....

read bidders

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//*** read #items;
//*** for (int i = 0; i < #items; ++i)
//***   read item[i];
//** read bidders;
//*** read #bidders;
//*** for (int i = 0; i < #bidders; ++i)
//***  read bidder[i];
//** read bids;
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   for each bid for this item {
//****      if (bid is the best so far) {
//*****      if (bid is in time &&
//*****          bid is higher than highestBid &&
//*****          bidder has enough money) {
//****         remember this bid as highest so far;
```


```
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
//***      print notice of no winner;
//***   }
//** }
//* clean up;
```

.....

Read bids

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//*** read #items;
//*** for (int i = 0; i < #items; ++i)
//***   read item[i];
//** read bidders;
//*** read #bidders;
//*** for (int i = 0; i < #bidders; ++i)
//***  read bidder[i];
//** read bids;
//*** read #bids;
//*** for (int i = 0; i < #bids; ++i)
//***  read bid[i];
//* determine Winners;
//** for each item {
```


```
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   for each bid for this item {
//****      if (bid is the best so far) {
//*****      if (bid is in time &&
//*****          bid is higher than highestBid &&
//*****          bidder has enough money) {
//****         remember this bid as highest so far;
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
//***      print notice of no winner;
//***   }
//** }
//* clean up;
```

.....

Remembering Bids

```
// Determine winners of daily auction;
//* initialize auction data;
//** read items;
//*** read #items;
//*** for (int i = 0; i < #items; ++i)
//***   read item[i];
//** read bidders;
```


```
//*** read #bidders;
//*** for (int i = 0; i < #bidders; ++i)
//***  read bidder[i];
//** read bids;
//*** read #bids;
//*** for (int i = 0; i < #bids; ++i)
//***  read bid[i];
//* determine Winners;
//** for each item {
//**   resolve auction for item;
//*** get winning bid for this item;
//****   highestBid = 0;
//****   winnerSoFar = "";
//****   for each bid for this item {
//****      if (bid is the best so far) {
//*****      if (bid is in time &&
//*****          bid is higher than highestBid &&
//*****          bidder has enough money) {
//****         remember this bid as highest so far;
//*****         highestBid = amount of this bid;
//*****         winnerSoFar = person who made this bid;
//****         remember this bid as highest so far;
//****      }
//***    }
//***   if any bid is a winner {
//***      print winning bid;
//***      subtract bid amount from winners account;
//***   } else {
//***      print notice of no winner;
//***   }
```


```
//** }
//* clean up;
```

. . . . . .

Are We Done Yet?

- Depends partly on the individual
- Are there any questions about how you would code the rest of this?

. . . . . .

2 Top-Down Design

Top-Down Design

- *Top-down design* is the extension of the idea of Stepwise Refinement to high-level design (breaking the system into modules)
  - Treat "highest level" pseudo code statements as functions
- Works well in medium-size programs
  - but there are better alternatives for very large systems

. . . . . .

2.1 Example: Top-Down Auction

Example: Top-Down Auction

Top-down starts like ordinary stepwise refinement:
Start with a general statement:

```
// Determine winners of daily auction;
```

Pick a statement that needs expansion

```
// Determine winners of daily auction;
//* initialize auction data;
//* determine Winners;
//* clean up;
```

But now we treat this as an indication that we will want 3 functions:

.....

Pseudocode statements ⇒ Function calls

```
int main (int argc, char** argv) {
  // Determine winners of daily auction;
  initializeAuctionData();
  determineWinners();
  cleanUp();
  return 0;
}
```

Pick one and design it, top-down.

.....

Determining Winners (stepwise)

In stepwise refinement, we would have gone from

```
// Determine winners of daily auction;
//* initialize auction data;
//* determine Winners;
//* clean up;
```

to

```
// Determine winners of daily auction;
//* initialize auction data;
//* determine Winners;
//** for each item {
//**     resolveAuction(item);
//** }
//* clean up;
```

.....

Determining Winners (top-down)

In top-down, we instead go from

```
int main (int argc, char** argv) {
  // Determine winners of daily auction;
  initializeAuctionData();
  determineWinners();
  cleanUp();
  return 0;
}
```

to

```
int main (int argc, char** argv) {
  // Determine winners of daily auction;
  initializeAuctionData();
  determineWinners();
  cleanUp();
  return 0;
}

void determineWinners() {
  for each item {
     resolveAuction(item);
  }
```


```
}
```

Now we might further infer that we will want a `resolveAuction` function.

. . . . . .

Initializing Auction Data

Pick one of the unexpanded functions (initializeAuctionData, cleanUp, or resolveAuction) to design.

```
void initializeAuctionData() {
//   read items;
//   read bidders;
//   read bids;
}
```

Do we make each of these 3 lines into 3 more functions?

- A judgment call.
  - I probably would because three data sets are so different
- If not, we continue designing this function body via stepwise refinement

  which should proceed pretty much as it did in the stepwise example

. . . . . .

Continuing Top-Down

```
int main (int argc, char** argv) {
  // Determine winners of daily auction;
  initializeAuctionData();
  determineWinners();
  cleanUp();
  return 0;
}
```


```
void determineWinners() {
  for each item {
     resolveAuction(item);
  }
}

void initializeAuctionData() {
//  read items;
//  read bidders;
//  read bids;
}
```

Let's look at resolving the auction.

.....

Resolving the auction

```
void resolveAuction(item) {
//    get winning bid for this item;
//    if any bid is a winner {
//       print winning bid;
//       subtract bid amount from winners account;
//    } else {
//       print notice of no winner;
//    }
}
```

Do we make any of these lines into a separate function?

- A judgment call.
  - Maybe getWinningBid

.....

getWinningBid

```
void getWinningBid (item) {
//   highestBid = 0;
//   for each bid for this item {
//      if (bid is the best so far) {
//         remember this bid as highest so far;
//      }
//   }
}
```

This calls for some internal stepwise refinement to add missing detail.

. . . . . .

Refining getWinningBid

```
void getWinningBid (item) {
//   highestBid = 0;
//   for each bid for this item {
//       if (bid is the best so far) {
//*       if (bid is in time &&
//*           bid is higher than highestBid &&
//*           bidder has enough money) {
//          remember this bid as highest so far;
//       }
//   }
}
```

- Note that, because the earliest levels of refinement were split into functions, no single function tends to have nearly as many levels of * refinement as when we are doing "pure" stepwise refinement.

. . . . . .

Further refining getWinningBid

```
void getWinningBid (item) {
//   highestBid = 0;
//*    winnerSoFar = "";
//   for each bid for this item {
//       if (bid is the best so far) {
//*       if (bid is in time &&
//*           bid is higher than highestBid &&
//*           bidder has enough money) {
//          remember this bid as highest so far;
//*           highestBid = amount of this bid;
//*           winnerSoFar = person who made this bid;
//       }
//   }
```

Any more separate functions?

- time comparison (if bid is “no later than” end of auction)

.....

noLaterThan

```
void noLaterThan (time1, time2) {
   if time1 hours < time2 hours
      return true;
   if time1 hours > time2 hours
      return false;

   if time1 minutes < time2 minutes
      return true;
```


```
    if time1 minutes > time2 minutes
       return false;

    return (time1 seconds <= time2 seconds);
}
```

.....

Recap

Functions introduced so far:

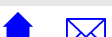
- `main`, `initializeAuctionData`, `determineWinners`, `cleanUp`, `resolveAuction`, `readItems`, `readBidders`, `readBids`, `getWinningBid`, `noLaterThan`
  - Overall:

```
void noLaterThan (time1, time2) {
//   if time1 hours < time2 hours
//      return true;
//   if time1 hours > time2 hours
//      return false;
//
//   if time1 minutes < time2 minutes
//      return true;
//   if time1 minutes > time2 minutes
//      return false;
//
//   return (time1 seconds <= time2 seconds);
}
```


```
void getWinningBid (item) {
   highestBid = 0;
   winnerSoFar = "";
//   for each bid for this item {
      // if (bid is the best so far) {
//*      if (bid is in time &&
//*         bid is higher than highestBid &&
//*          bidder has enough money) {
          // remember this bid as highest so far;
//*          highestBid = amount of this bid;
//*          winnerSoFar = person who made this bid;
      }
   }
}


void resolveAuction(item) {
//   get winning bid for this item;
//   if any bid is a winner {
//      print winning bid;
//      subtract bid amount from winners account;
//   } else {
//      print notice of no winner;
//   }
}

void initializeAuctionData() {
//  read items;
//  read bidders;
//  read bids;
}
```

```
void determineWinners() {
//  for each item {
//     resolveAuction(item);
//  }
}


int main (int argc, char** argv) {
  // Determine winners of daily auction;
  initializeAuctionData();
  determineWinners();
  cleanUp();
  return 0;
}
```

- Eventually we will want to divide these into modules
  - But we've already looked at how that would happen