Programming Correctly by Stepwise Refinement

Brought to you by:
Shiri Dori
Shai Erera

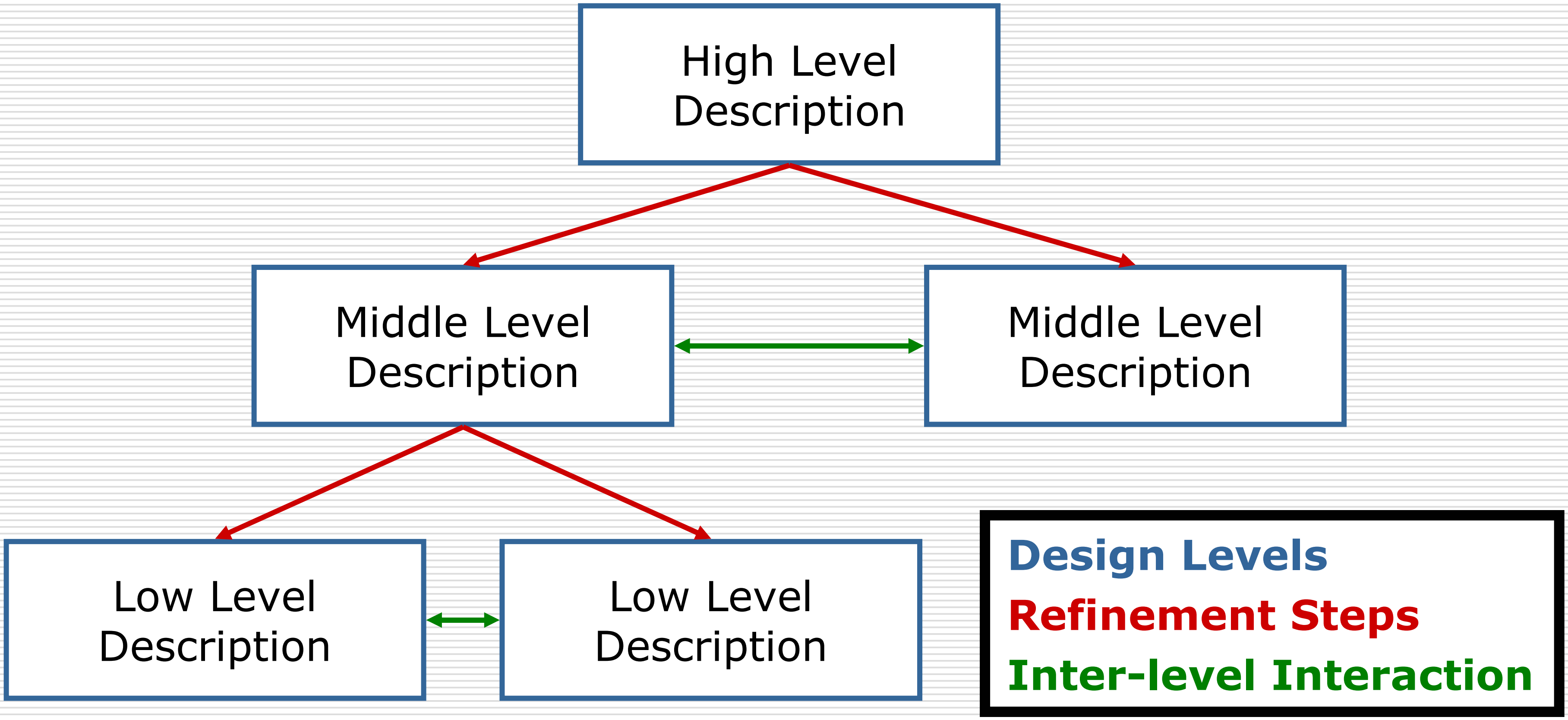
Ad-hoc programming

- Today, as in the past, some programmers just start writing, and "cross their fingers" in hope
- Debugging is long and disappointing
- Rewrites or patchwork code are needed to fix errors
- Trial and error – with many errors!

What we will see today

- Last week, Yaniv and Hagai showed us top-down design, and explained why correctness proof is infeasible
- We’ll explore both topics in further depth:
  - Stepwise Refinement – how to construct a top-down design successfully
  - Correctness – how to convince yourself of program correctness

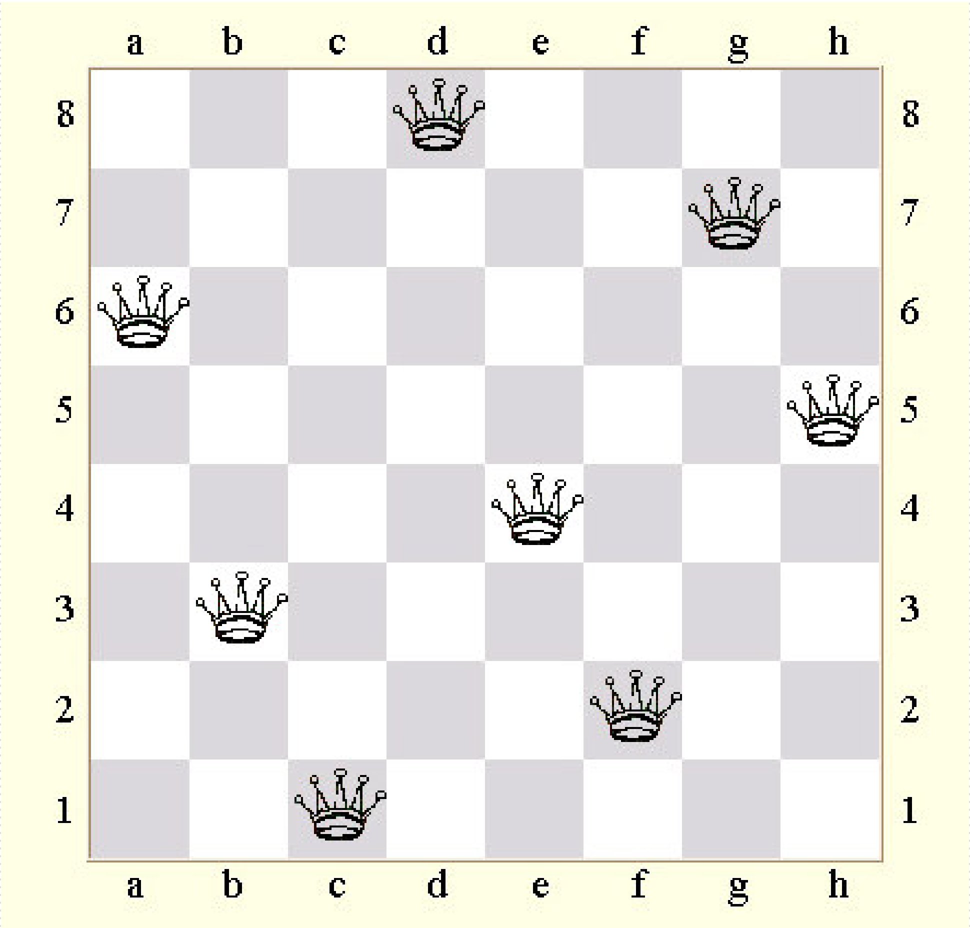
Stepwise Refinement and Top-Down Design



Papers

- N. Wirth, Program Development by Stepwise Refinement, *Communications of the ACM*, 14, 4 (1971) 221-227.
- H. D. Mills, How to write correct programs and know it, *Proceedings of the international conference on Reliable software* (1975) 363-370

Program Development by Stepwise Refinement



Motivation

- Programming is taught usually by examples
- Students learn finished products
- Therefore, students focus on the Programming Language’s syntax rather than its logic
- However, programming often consists of the design of new products, rather than the maintenance of old ones

Stepwise Refinement

- Program development can be expressed as a sequence of refinement steps
- In each step several instructions are decomposed into more detailed ones
- This process terminates when the instructions are expressed in terms of the underlying PL
- The idea is demonstrated by analyzing the 8-Queens problem

8-Queens Problem

- Objective: place 8 queens on a chess board such that no queen may be taken by another
- We can view the problem as if we have a set *A* of all configurations on the board and we need to select one that matches the above criteria
- The size of *A* is enormous ($\sim 2^{32}$)

8-Queens Problem (cont)

- In the 70s, a strong computer would take ~7 hours to complete the task
- Reduce the size of *A* by defining that each queen is placed in a different column
- Change the criterion into 2 different criteria such that both form the original criterion

8-Queens Problem (cont)

- Now a strong computer would take ~100 seconds to complete the task
- However a very slow computer would take ~280 hours to complete the task
- Need to break the problem further by solving partial configurations

8-Queens Problem (cont)

- Breaking the problem further:
  - Place 1 queen and check *q*
  - Place the second queen in a square where *q* holds
  - Continue until all the queens are placed
- Based on the assumption that checking *q* for fewer queens is easier
- A partial solution cannot be extended to a full solution if it does not match the criterion

8-Queens – Pseudo Code

```
j := 1
repeat trystep j;
    if successful then advance else regress
until (j < 1) V (j > n)
```

8-Queens – First Draft

```
considerFirstColumn
repeat tryColumn
    if safe then
        setQueen;
        considerNextColumn
    else regress
until lastColDone V regressOutOfFirstCol
```

8-Queens – Refinement

```
procedure tryColumn
    repeat advancePointer; testSquare
    until safe V lastSquare

procedure regress
    reconsiderPriorColumn
    if not regressOutOfFirstCol then
        removeQueen
        if lastSquare then
                reconsiderPriorColumn;
                if not regressOutOfFirstCol then
                        removeQueen
```

8-Queens – Outline

```
considerFirstColumn
repeat tryColumn
    if safe then
        setQueen;
        considerNextColumn
    else regress
until lastColDone V regressOutOfFirstCol
```

8-Queens – Data Representation

- 8x8 boolean matrix
- Need to consider data representation in terms of **efficiency** and **ease** of performing the various operations
- What about a vector of size 8?

8-Queens – Further Refinement

integer j
integer array x[1:8]

procedure *considerFirstCol*
    j = 1;
    x[1] = 0;

procedure *considerNextCol*
    j = j + 1;
    x[j] = 0;

procedure *reconsiderPriorCol*
    j = j – 1

procedure *advancePointer*
    x[j] = x[j] + 1

procedure *lastSquare*
    return *x[j] == 8*

procedure *testColDone*
    return *j > 8*

procedure *regressOutOfFirstCol*
    return *j < 1*

8-Queens – Diving Deeper

- Now the program is expressed in the terms:
  - testSquare
  - setQueen
  - removeQueen
- *testSquare,* which is a very frequent method, needs a more efficient way to calculate

8-Queens – More DS

- To help *testSquare* efficiency we introduce the following boolean arrays:
  - a[k] = **true**; row *k* is free
  - b[k] = **true**; diagonal “/” is free
  - c[k] = **true**; diagonal “\” is free
- How to check *b* and *c* efficiently?

8-Queens – More DS (cont)

procedure testSquare

safe := a[x[j]] $\wedge$ b[j + x[j]] $\wedge$ c[j – x[j]]

procedure setQueen

a[x[j]] = b[j + x[j]] = c[j – x[j]] := **false**

procedure removeQueen

a[x[j]] = b[j + x[j]] = c[j – x[j]] := **true**

- Since *x[j]* is examined frequently, the integer *i* is set instead of *x[j]*

8-Queens – Outline

```
considerFirstColumn
repeat tryColumn
    if safe then
        setQueen;
        considerNextColumn
    else regress
until lastColDone V regressOutOfFirstCol
```

8-Queens – Final Program

```
j := 1; i := 0;
repeat
    repeat i := i+1; testSquare
    until safe V (i = 8);
    if safe then
        setQueen;
        x[j] := i; j := j + 1; i := 0;
    else regress
until (j > 8) V (i < 1);
if j > 8 then PRINT(x) else FAILURE
```

8-Queens – Recursion Version

```
procedure TryColumn(j) ;
    begin integer i; i := 0;
    repeat i := i + 1; testSquare;
        if safe then
                setQueen; x[j] := i;
                if j < 8 then TryColumn (j + 1);
                if not safe then removeQueen
    until safe V (i = 8)
```

Generalized 8-Queens

- In certain applications we may want to output more then one solution
- For example, output all possible configurations of the board
- For that we need to:
  - Generate more solutions once one is found
  - Determine if all solutions were generated
  - Store/Output a solution

Generalized 8-Queens (cont)

```
considerFirstColumn
repeat tryColumn;
    if safe then
        setQueen; considerNextColumn;
        if lastColDone then
                PRINT(x); regress
    else regress
until regressOutOfFirstCol
```

Generalized 8-Queens (cont)

- How to determine if all configurations were output?
- Mark configurations as sequences of integers from “00000000” to “88888888”
- Note that the configurations are output in increasing order

Stepwise Refinement – Recap

- Program construction consists of *refinement steps*
- In each step a task is broken into a number of tasks
- A refinement in the task’s description may be accompanied by a refinement data’s description, which constitute the means of communication between subtasks

Stepwise Refinement – Recap

- During the process of *stepwise refinement,* a notation which is natural to the problem in hand should be used as long as possible
- The notation should develop according to the programming language that will be used to implement the solution
- If written correctly, solution can easily be extended for more requirements

How to Write Correct Programs and Know It

“An Old Myth and New Reality”

- Myth: programming is a trial-and-error method (with lots of errors)
- The Author claims that programmers can write entirely correct programs
- Reality (says Mills): “You can learn to consistently write programs which are error free”.

Sounds fantastic, huh?

- Mills begins with a discussion, aimed to convince readers that this is possible
  - Can’t show absence of bugs
  - Can’t prove correctness
  - Acquire confidence in correctness
  - You need to know what you want (be capable enough)

Can’t show absence of bugs

- Like Dijkstra said, testing can’t demonstrate the absence of bugs
- You can never be sure you found the last bug – there may be more…
- In fact, says Mills, your confidence drops with each bug you find
- A better solution?
- Never find the first bug!

Can’t prove correctness

- A Philosophical Discussion: Proofs
- Mills says that proof is relative
  - Mathematical proof may fail to convince
  - Or convince everyone, yet be erroneous
  - An intuitive approach can convince more
- Therefore, you can never really prove that a program is error-free

Acquire confidence in correctness

- Confidence depends mostly on testing
- We are likely to get errors, and as # of errors increases, confidence drops
- But if there are no errors, confidence will increase with each test passed
- That is why we should write them correctly from the start
- The difference between 0 and 1: big!

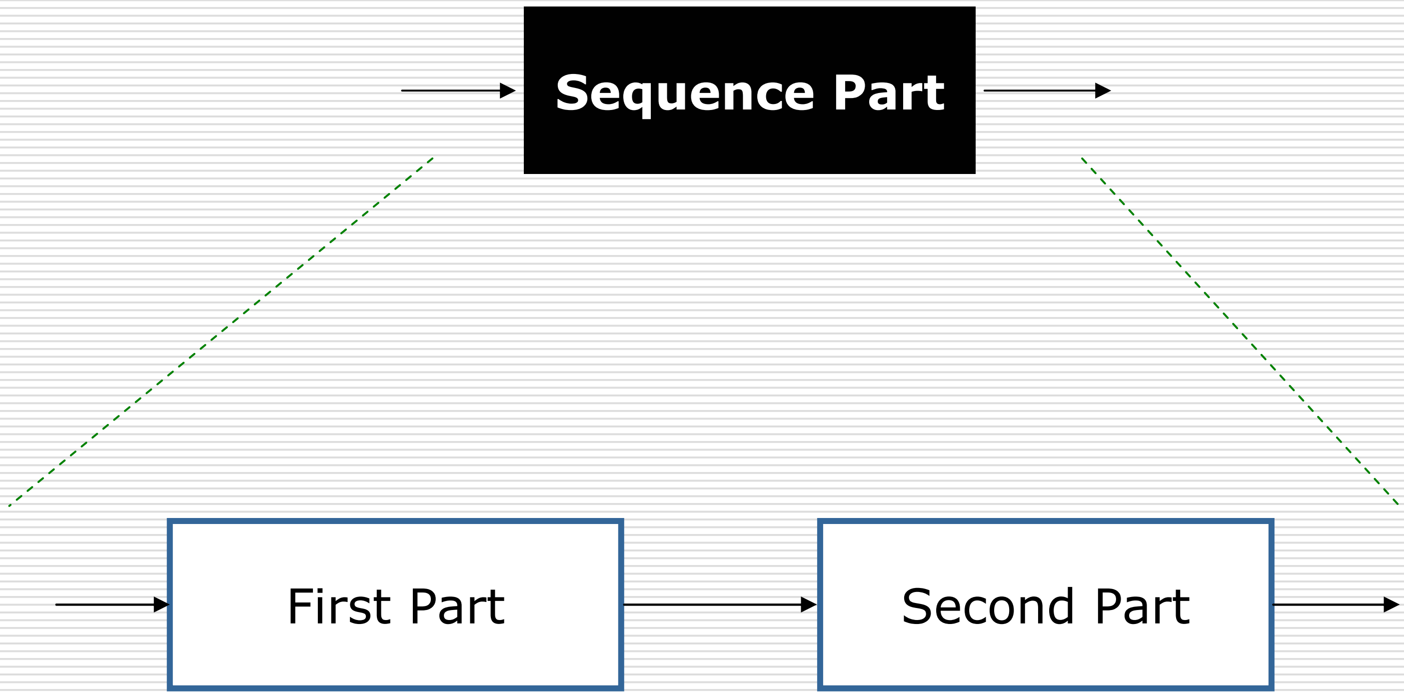
Correctness vs. Capability

- *Correctness* means that your program does what you intended it to do
- "Determining what a program should do is a much deeper problem..."
- *Capability* means you can figure out what the program should do
- But if you know what should happen, you can make a program to do it

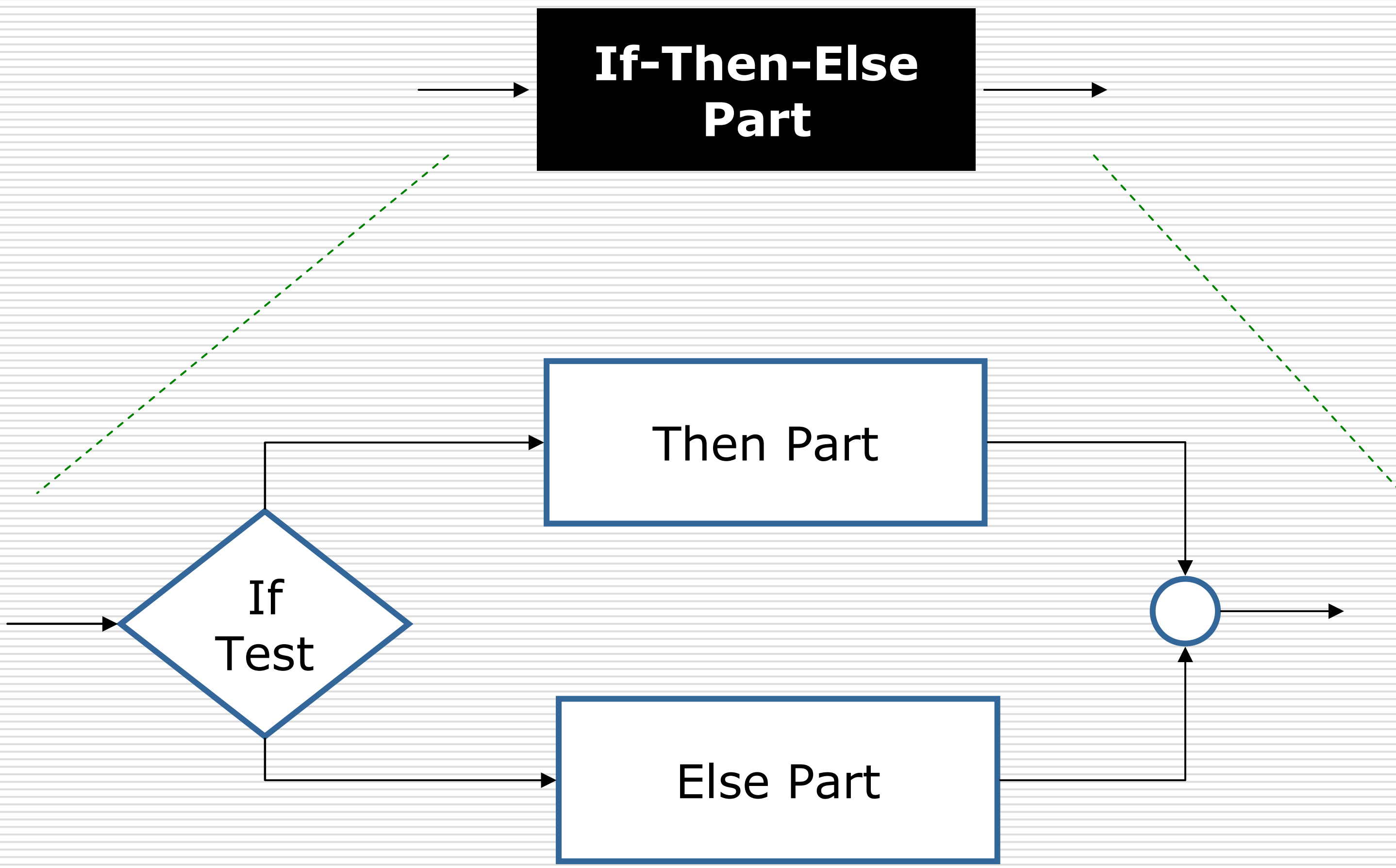
How do we do this?

- Usage of black boxes to describe functionality (input, output)
- Assume that each black box is correct; prove interactions between the boxes
- Interactions such as “sequence”, “if-else”, “while”, (and procedure calls) can be shown by reasoning

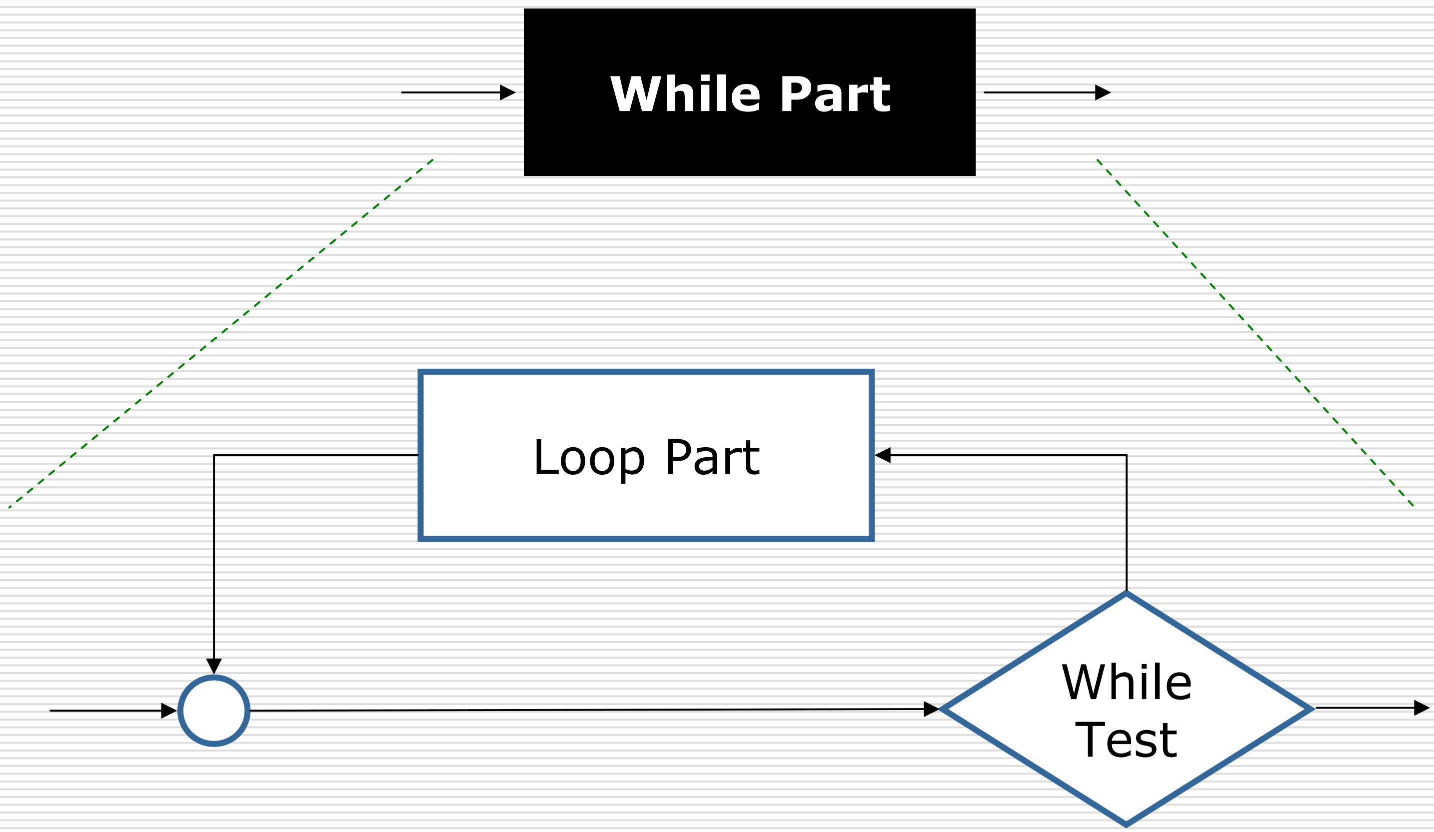
Connectors – Sequence



Connectors – If-Then-Else



Connectors – While



Small Example

```
j = 0; sum = 0;
while (a[j] > 0) {
    sum += a[j];
    j++;
}
```

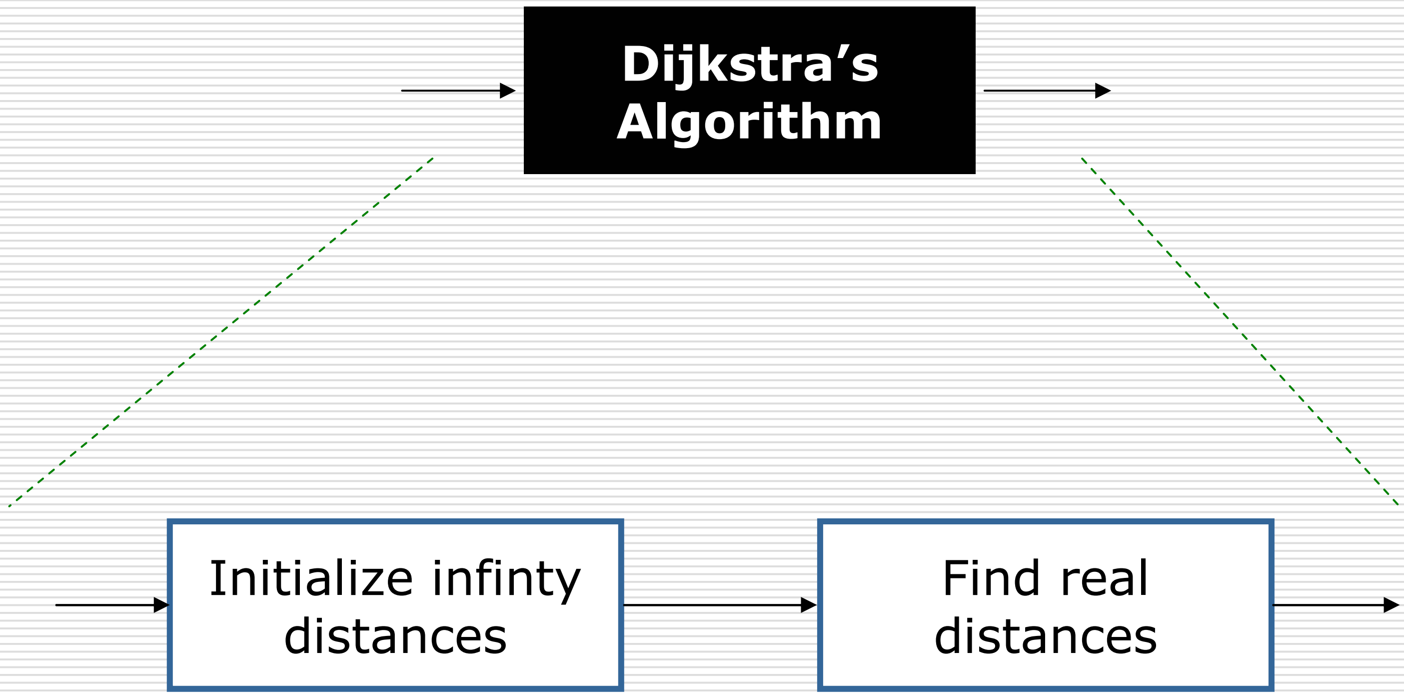
> We haven't checked that the loop terminates

- What's wrong with this?

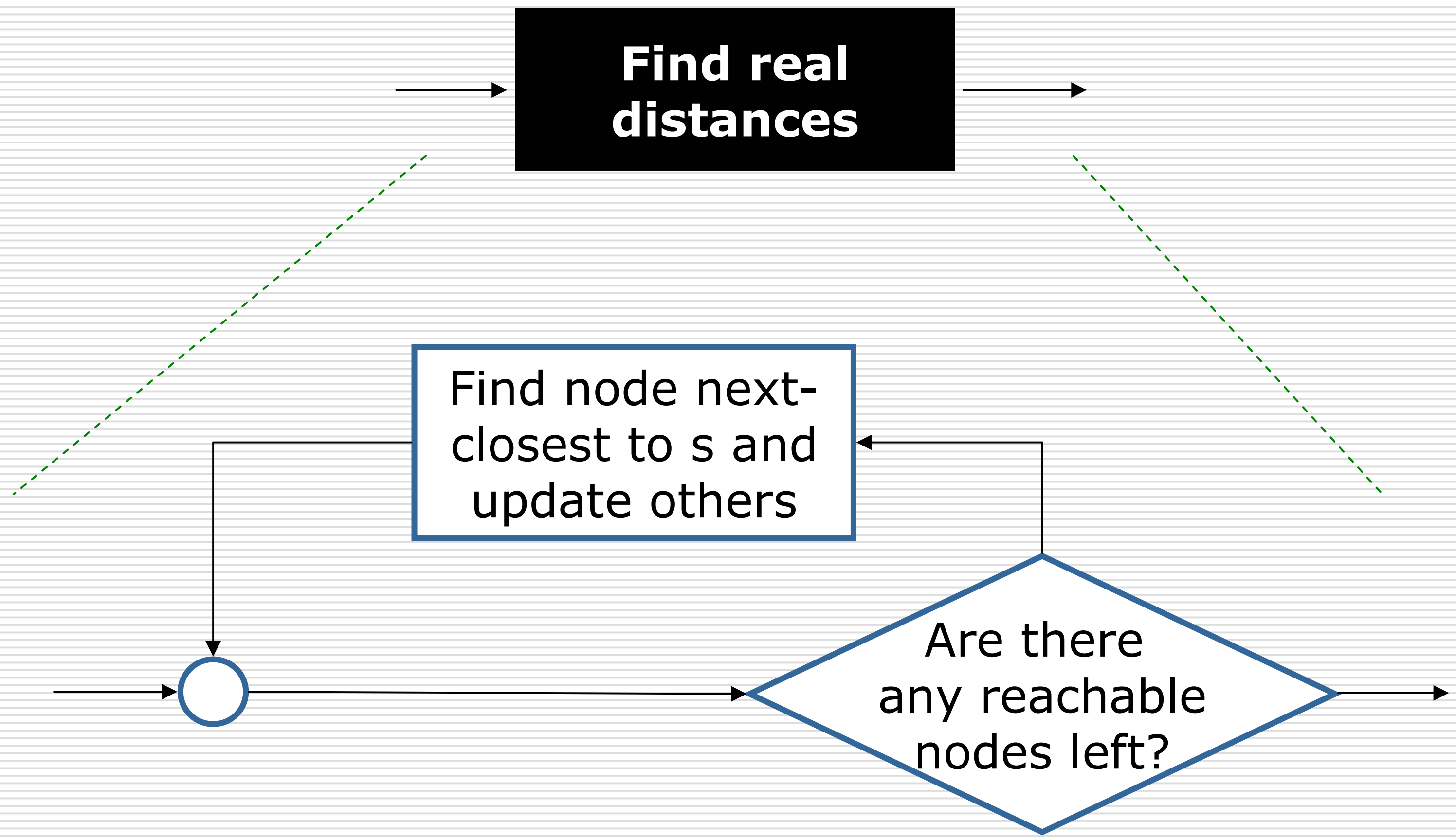
Case Study – Dijkstra Algorithm

- Env: Directed Weighted Graph
- Goal: find shortest path from vertex **s** to all other vertices
- We do this using Dijkstra's Algorithm, which iteratively finds next-closest vertex to ***s***

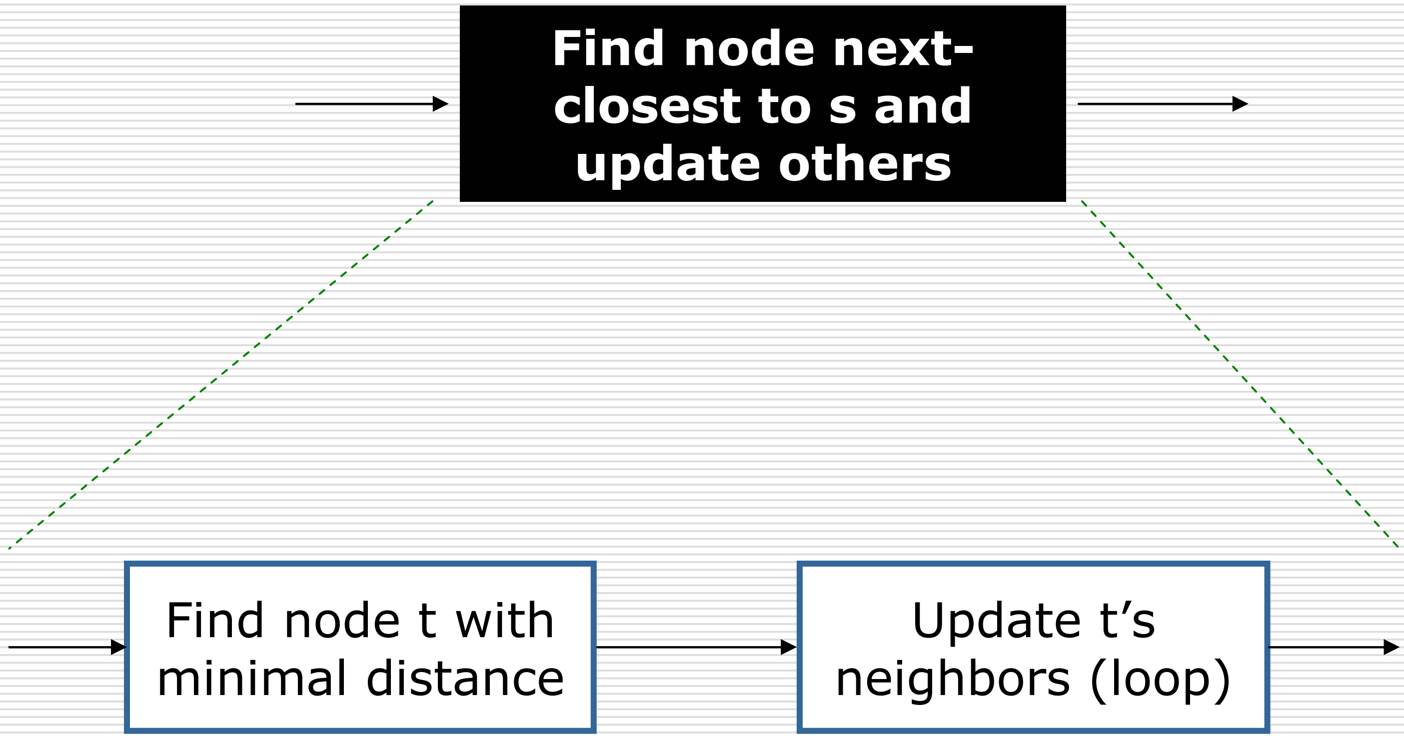
Dijkstra Algorithm - initialize



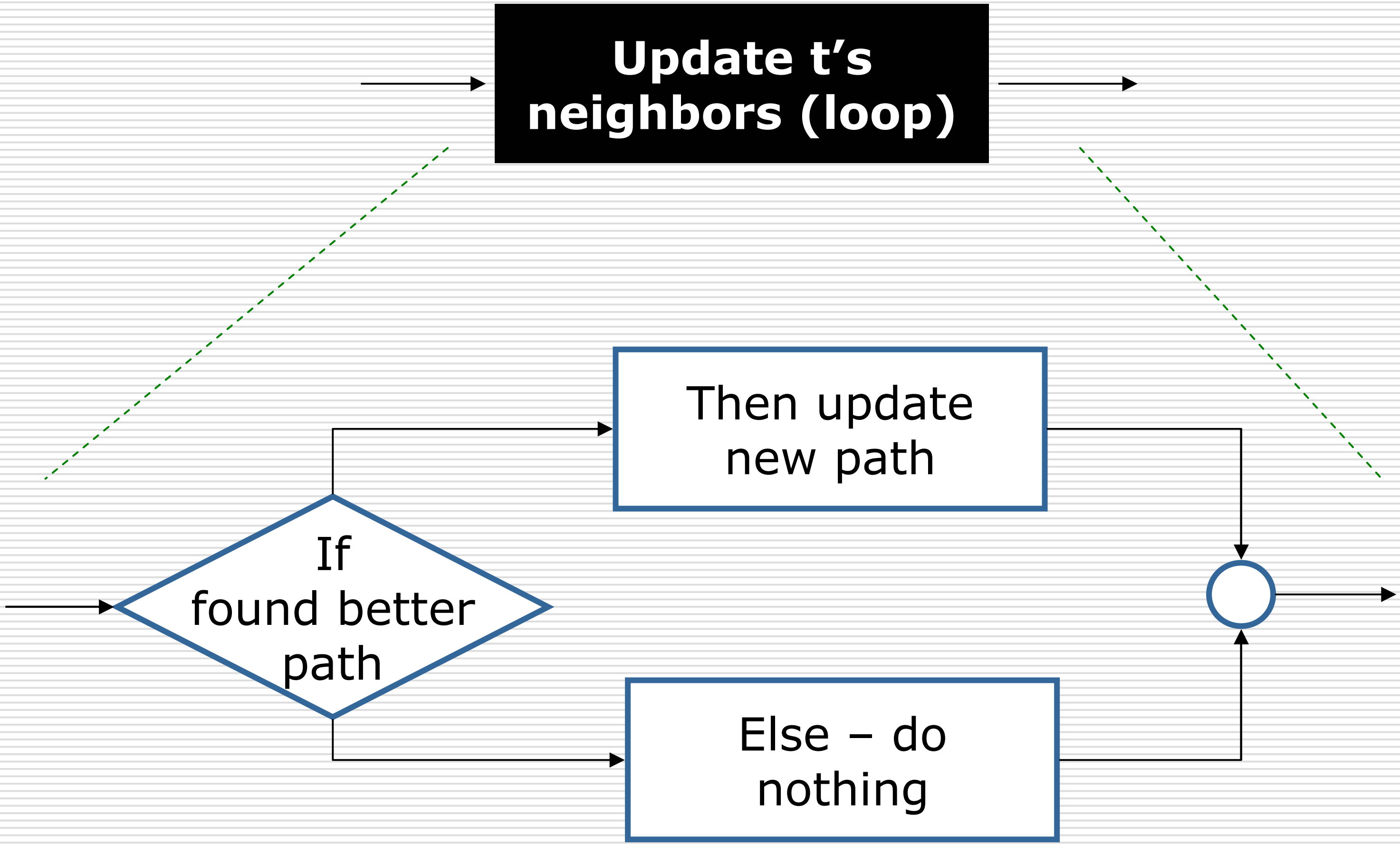
Dijkstra Algorithm – distances



Dijkstra Algorithm - sequence



Dijkstra – Update Neighbors



Dijkstra Case Study – So What?

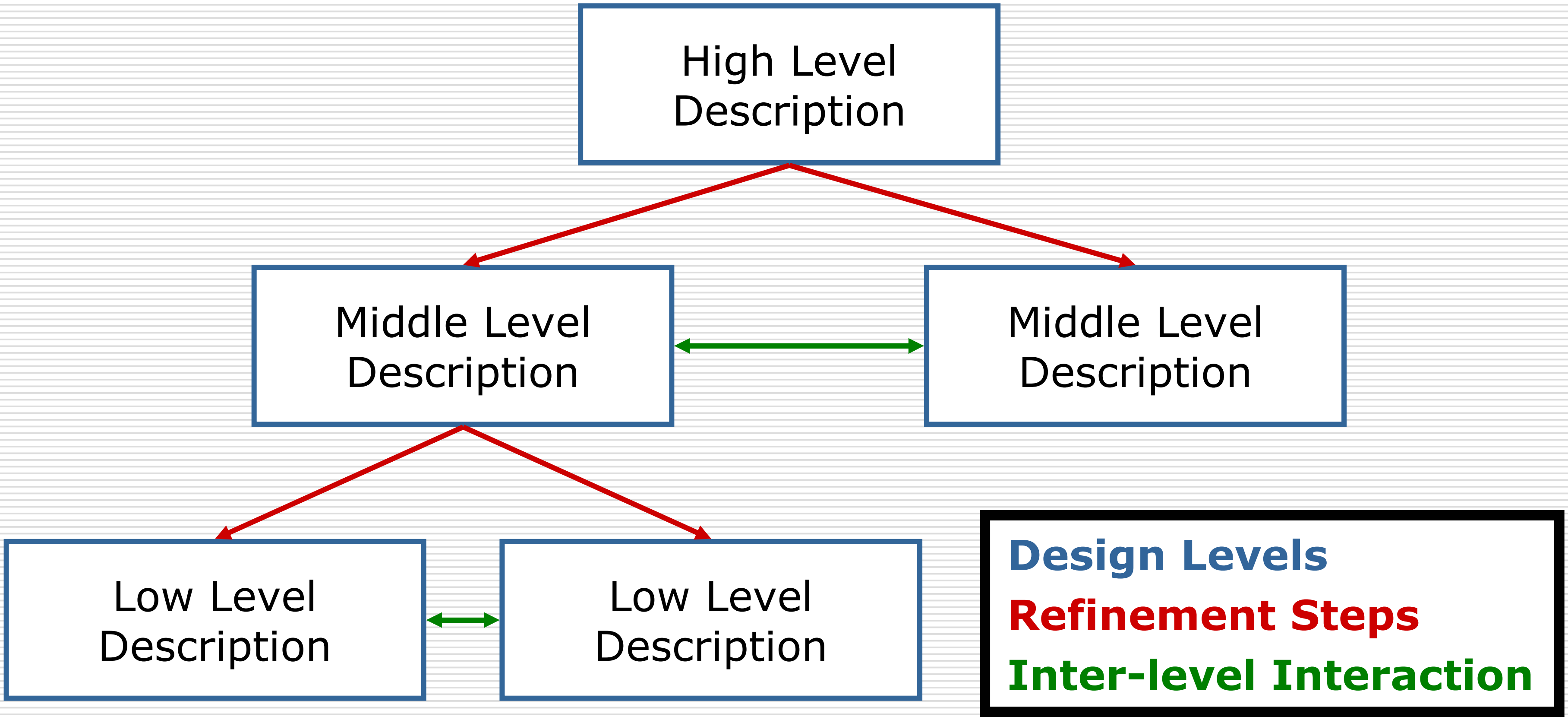
- We can continue this test down to the lowest levels of code (or design)
- And convince ourselves that it works

- By thinking through the process carefully, we are likely to avoid most (if not all) bugs

Conclusion

- Programmers can write entirely correct programs/design that are extensible
- This can be achieved by:
  - Using stepwise refinement,
- And making sure that:
  - Each level is correct, and
  - Integration between parts of each level is correct

Stepwise Refinement and Top-Down Design



The Principles of Design

- Think before you do something!
  - Plan the general framework
  - Be convinced of correctness
- The articles were written in the 70s; discuss code for small systems
- Yet, principles regarding code can be easily extended to design large systems