Lecture 2: Overview of code readability

- What is readability and why does it matter?
  - Deep aspects
    - Overall structure (including class structure)
    - Broad solution approach
  - Program self-documentation — comments, use of language and libraries
  - "Superficial" aspects — layout, names
- Studio assignment 1

Alumni survey

- Generally strong emphasis on practical software engineering skills: tools like version control systems; code reviews; testing; teamwork; effective communication
- Some specifics that showed up more than once: functional programming; scripting languages; parallelism; databases
- "I didn't take any classes which taught me the importance of code commenting, code documentation, design before development, and just generally good software practices. I find this is extremely common with college curriculum, having hired grads out of many schools. My take on it (hopefully incorrect) is that the professors themselves don't know good process, having spent all their time in academia. I think this hurts the entire industry ... bugs everywhere!"

Code readability

- Readability = ability of others to understand what your code does and how it works
- Why does it matter?
  - Code is frequently modified, to fix bugs or add functionality.
  - Software companies care a lot about this
    - Code reviews
    - Coding standards

Code readability (cont.)

- Other aspects of code quality basically synonymous with readability:
  - Maintainability: by definition, whatever makes code easy to modify.
  - Modularity: program divided into parts in a logical way, making it easier to understand and modify
  - Simplicity: the ultimate goal of all engineering design

Writing readable code

- Readability must be pursued at multiple levels:
  - Choosing simplest method of solution
  - Modularization into methods and classes
  - Proper use of libraries
  - Proper use of control structures
  - Appropriate comments
  - Choice of names (variables, functions, classes)
  - Visual: layout, spacing
- We will discuss these aspects in several lectures; today is an overview.

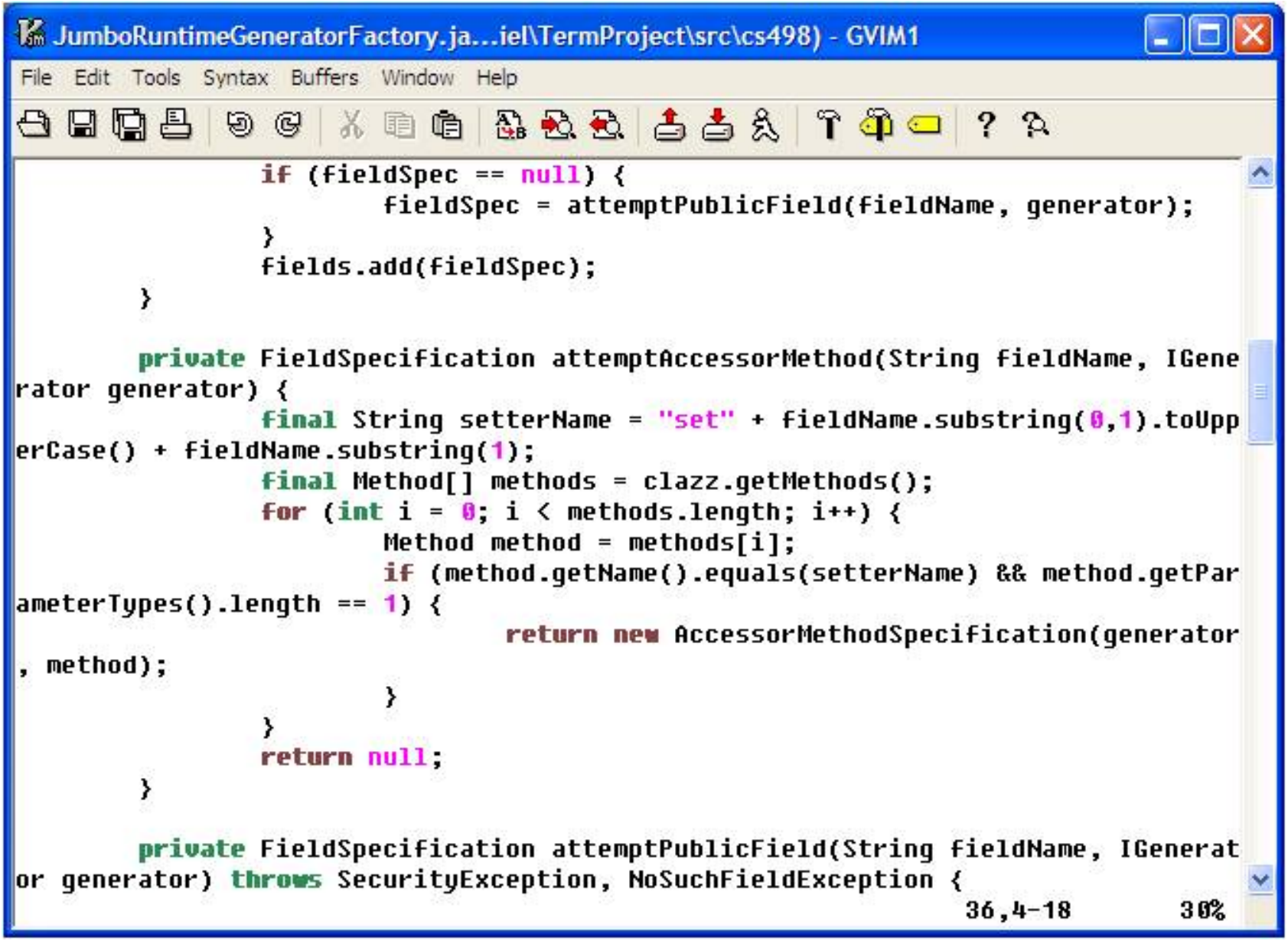
Textbook

- **Steve McConnell, Code Complete, 2nd ed., Microsoft Press, 2004.**
- **Discusses readability at all levels.**
- **Widely considered an invaluable repository of programming wisdom.**

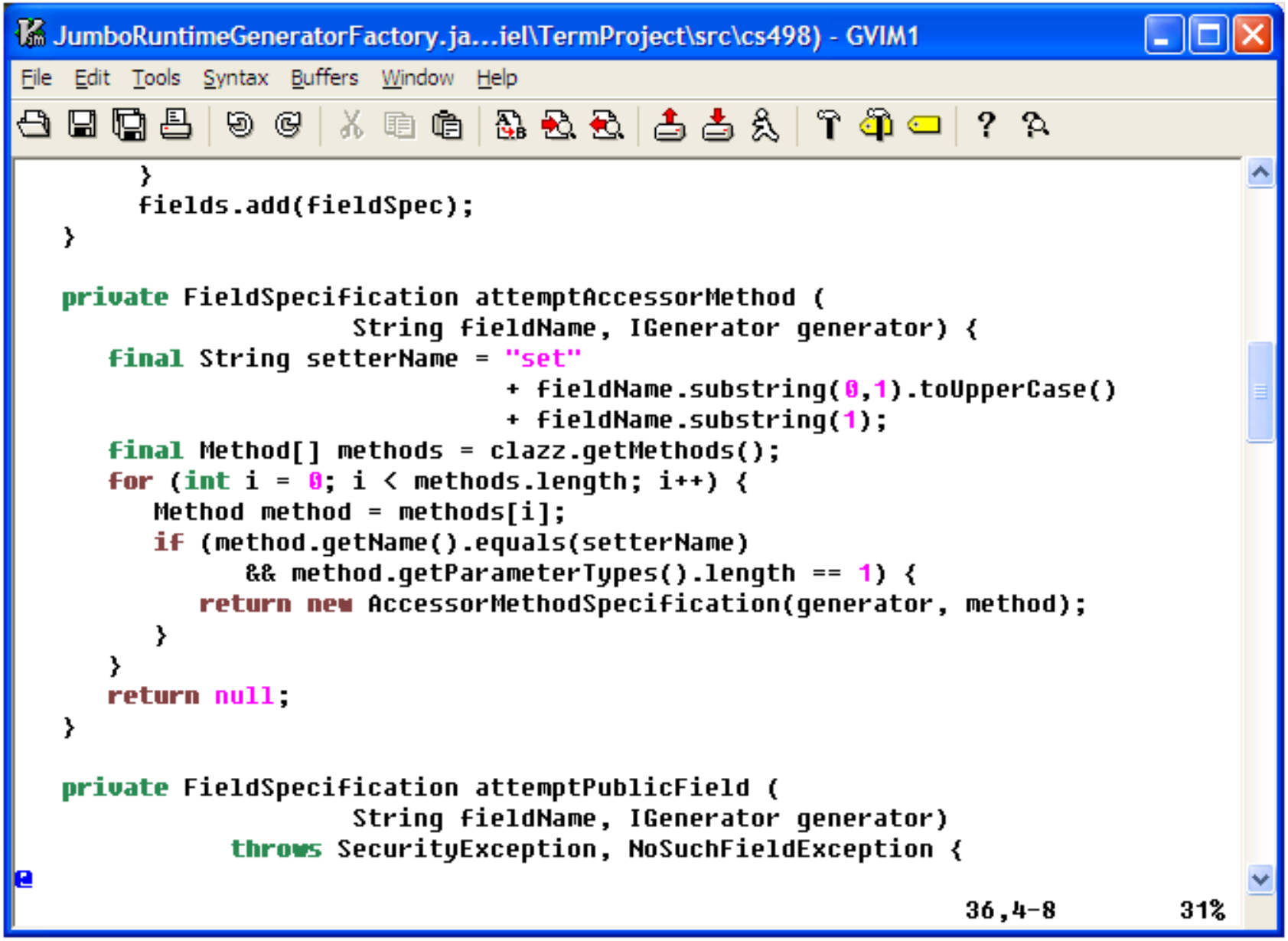
Code layout [Chapter 31]

- Goals of good layout:
  - Display syntactic structure clearly
  - Pleasing appearance
  - High information density
- General rules of layout
  - Consistent indentation (3 or 4 spaces)
  - Avoid long lines
  - Consistent placement of brackets, blank lines, etc.

Example of bad layout



Example of good layout



Commenting [Chapter 32]

- **Goal of commenting:**
  - **State what readers needs to be able to find quickly (e.g. method headers)**
  - **Explain what is not obvious in the code**
  - **Don't overdo it**
- **Programs are rarely completely "self-documenting" — comments are essential**
- **Important to "maintain" comments. The only part of a program that is certain to be "correct" is the actual code; comments may be misleading if code changes under them. (See section 32.3.)**

Names [Chapter 11]

- **Use descriptive names. Most of the words that appear in a program, aside from comments, are user-chosen names (variables, methods, etc.), so these are the "first line of readability."**
- **As with comments, names should be descriptive, but can detract from readability if they're too long.**
- **Examples (CC, pg. 261):**
  - **Good names:** `runningTotal`, `currentDate`, `linesPerPage`
  - **Not-so-good names:** `rTot`, `current`, `lines`
  - **Sometimes good names:** `i`, `j`, `x`

Using the language properly [Chapters 12–17]

- **Many examples can be found on the web. These are from Code Complete:**
- **From page 301:**

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||
   ( elementIndex == lastElementIndex ) ) {
```

should be:

```
boolean finished
           = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex );
boolean repeatedEntry = ( elementIndex == lastElementIndex );
if ( finished || repeatedEntry ) {
```

Using language properly (pg. 369)

```
GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;
}
```

should be:

```
while ( true ) {
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;

   if ( ! ( score < targetScore ) && ( ratingIncrement != 0 ) ){
      break;

   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
}
```

A personal fave...

```
if (x < limit)
   in_range = true;
else
   in_range = false;
```

Using methods [Chapter 7]

- **Two basic rules:**
  - **Methods should contain high-quality code**
  - **The purpose and effect of the method should be easily described.**
- **Rule 2 implies:**
  - **Minimize side effects**
  - **Avoid complex, obscure parameter constraints**

Class structure [Chapter 6]

- **Similar rules as for methods:**
  - **High-quality code**
  - **The purpose of the class should be easily described.**
  - **Interactions with other classes should be minimized.**
  - **Class should have a single "core competence" — don't add unrelated functionality**

Choosing the simplest solution

- Trade efficiency for simplicity, when appropriate
  - Choose appropriate language
  - Use "functional programming" approach
- Trade code for data
  - Interpretive methods
  - Table-driven methods

Example: Knuth vs. McIlroy

- In Communications of the ACM, June 1986, Jon Bentley's "Programming Pearls" column, Donald Knuth offers a solution to this problem posed by Bentley:

  > Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

- Bentley said it "should be able to find the 100 most frequent words in a 20-page technical paper (roughly a 50K byte file)."
- Knuth wrote a beautiful program of about 200 lines of Pascal, including a highly efficient data structure (a "hash trie") to store words.
- Doug McIlroy wrote a review in the same issue...

Example: Knuth vs. McIlroy (cont.)

- **McIlroy pointed out that the problem could be solved with the following unix command:**

```
tr -cs A-Za-z '
' | tr A-Z a-z | sort | uniq -c | sort -rn | sed ${1}q
```

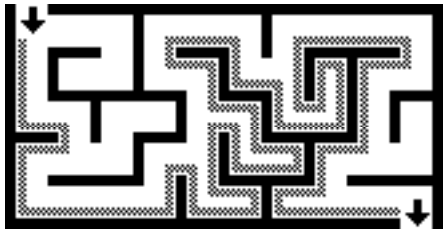
- **Each of the programs used here — tr, sort, uniq, sed — were existing Unix programs.**
- **McIlroy says this ran in 30 seconds on a 10,000–word file.**

Complications

- **Efficiency vs. simplicity**
  - **"Premature optimization is the root of all evil." (D.E. Knuth)**
  - **Modularize to isolate efficiency-critical components**
- **Short-term vs. long-term simplicity**
  - **Design for generality/modifiability**

Assignment: Unit Testing

Wall-follower applied to a simple maze



Perfect Maze

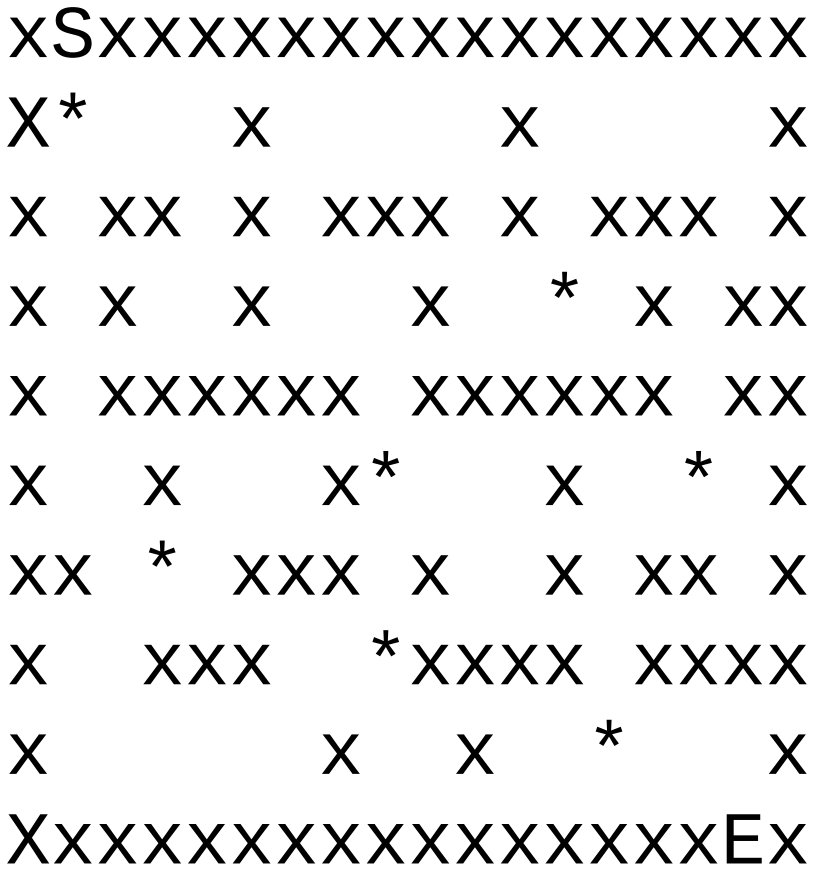
A perfect maze is defined as a maze which has one and only one path from any point in the maze to any other point

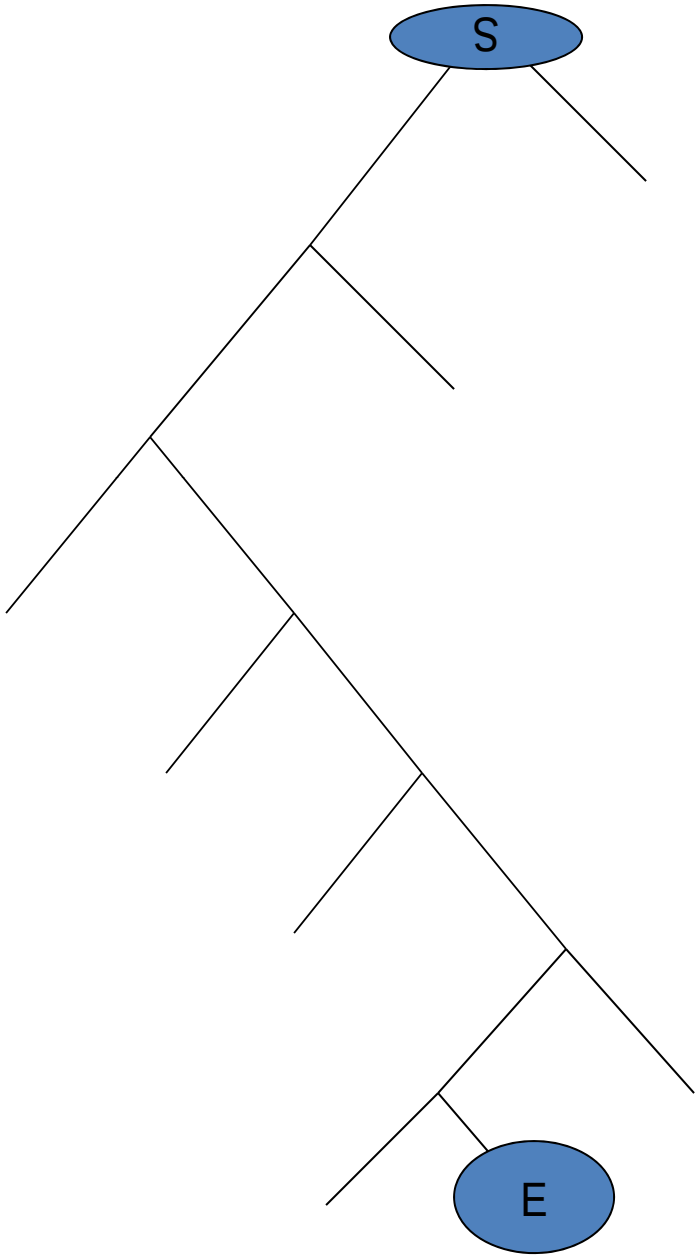
- no inaccessible sections
- no circular paths
- no open areas

“Pefect Maze”

```
xSxxxxxxxxxxxxxxxx
x     x       x      x
x xx x xxx x xxx x
x x   x    x     x xx
x xxxxxx xxxxxx xx
x   x    x     x     x
xx    xxx x   x xx x
x   xxx    xxxx xxxx
x        x   x       x
xxxxxxxxxxxxxxxxxEx
```

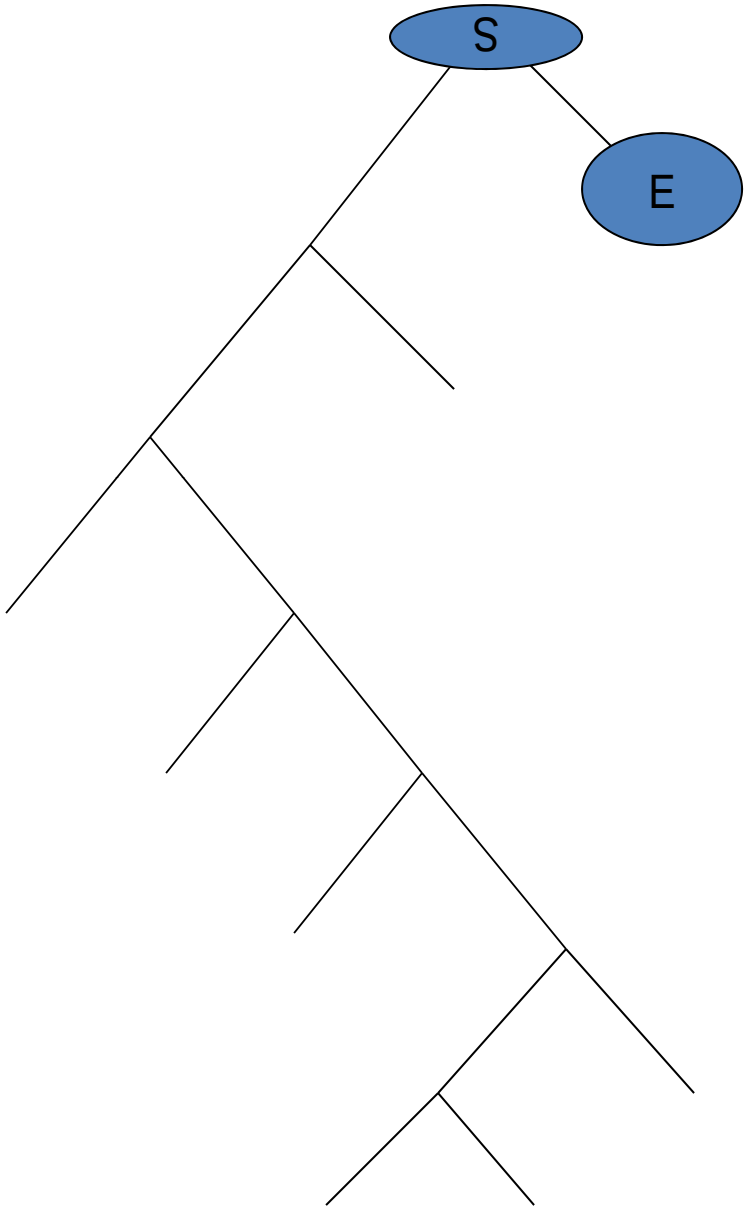
* Denotes a node




S
E

“Pefect Maze”

```
xSxxxxxxxxxxxxxxxx
x     x       x      x
x xx x xxx x xxx x
x xE x    x     x xx
x xxxxxx xxxxxx xx
x   x    x     x     x
xx    xxx x   x xx x
x   xxx    xxxx xxxx
x        x   x       x
xxxxxxxxxxxxxxxxxx
```


S
E

Pseudo-code
in its simplest form

```
InitializePosition
InitializeDirection
done = FALSE
While (!done)
 {
 if IsOpenForward
  {
  MoveForward
  if IsEndNode
    done = TRUE
  else if IsOpenToRight
   TurnRight
  }
 else
  TurnLeft
 }
```

“Units”

- Read a grid
- Clean the grid
- Find start position
- Find start direction
- Test forward cell
- Test right cell
- Test end of maze
- Turn left
- Turn right
- Move forward

For testing, try using:

- ‘ ‘ will denote an open cell
- ‘x’ will denote a wall
- ‘s’ is start
- ‘e’ is end

- Make your application adaptable for these values

Simple Moving and Direction

- movements:
- east: dx=1, dy=0
- south: dx=0, dy=-1
- west: dx=-1, dy=0
- north: dx=0, dy=1

- To turn, increment or decrement through directions

Test Grid

You don’t need an actual “maze” to test but you do need a grid.

Example:

```
xxxxx
x123x
x456x
x789x
Xxxxx
```

By using these types of known inputs, it becomes easy to test your units. In this case, each non-wall cell has a unique value.

Example Test Grid Cases

- Edges not closed
  - Generate a warning
- Not all chars in same case
- No ‘end’ cell
- No ‘start’ cell
- ‘extra’ cells around maze

Maze Assignment: Week 1

Sept 7

- This assignment is not about mazes
- This assignment IS about unit testing and test-driven development
- use Java or C# for the entire assignment
  - Purpose: use JUnit (or csUnit)
  - Write comments for indicated units
  - write tests for units
    - Points off for obvious cases that you did not consider