Announcements

Project 3 is out. Key deadlines:

- Form a team as soon as possible (theoretically by today, but if you need to form later, that is OK, too).
- Phase 1: World Generation (Due 4/26 at 11:59 PM).
- Phase 2: Interactivity (Due 5/1 at 11:59 PM).
  - Must submit to gradescope by this time.
- In-lab demo: Whenever you sign up during last week (5/1-5/3).
- Gold points: Make a demo video by 5/3.

CS61B

Lecture 34: Software Engineering II

- Teamwork
- Cast Study in Complexity: Build Your Own World
- Modular Design

Build Your Own World

Build Your Own World

In the previous software engineering lecture, we talked about complexity.

- "Complexity is anything related to the structure of a software system that makes it hard to understand and modify the system."
- None of the assignments in 61B have really given you enough room to create truly complex code.
  - Project 3 will give you a chance.

In Project 3, you'll be building a system in two phases:

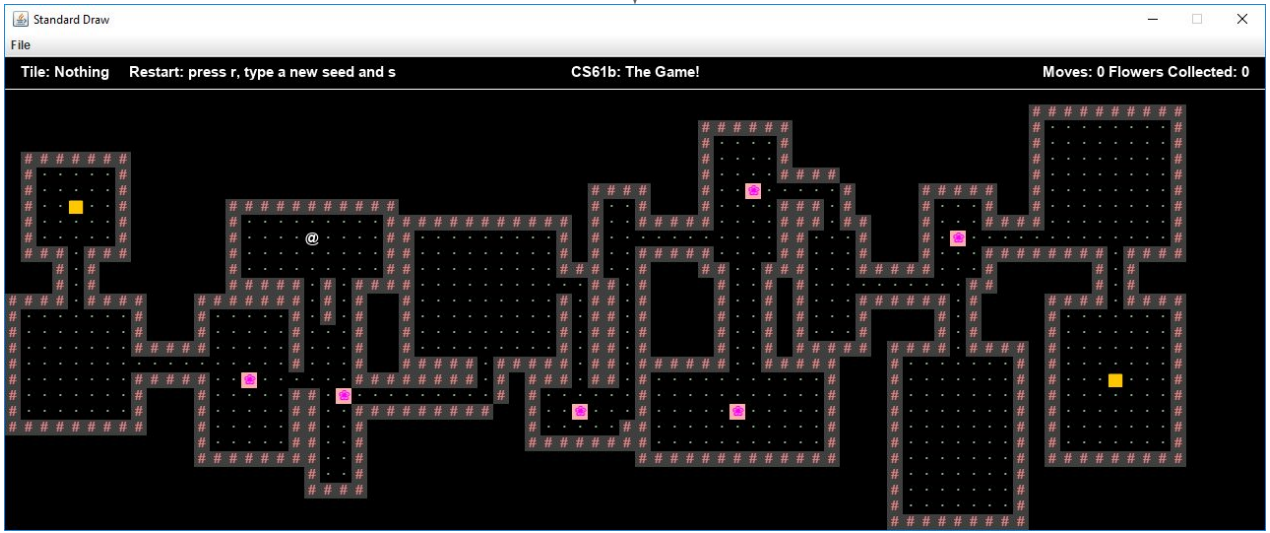
- World Generation
- Interactivity

Part 1: World Generation

Given a random seed (long), generate a 2D world (TetTile[][]) with rooms and hallways.

- N: Create new world.
- 343434: Random seed.
- S: End of seed marker.

```
interactWithInputString("N343434S")
```



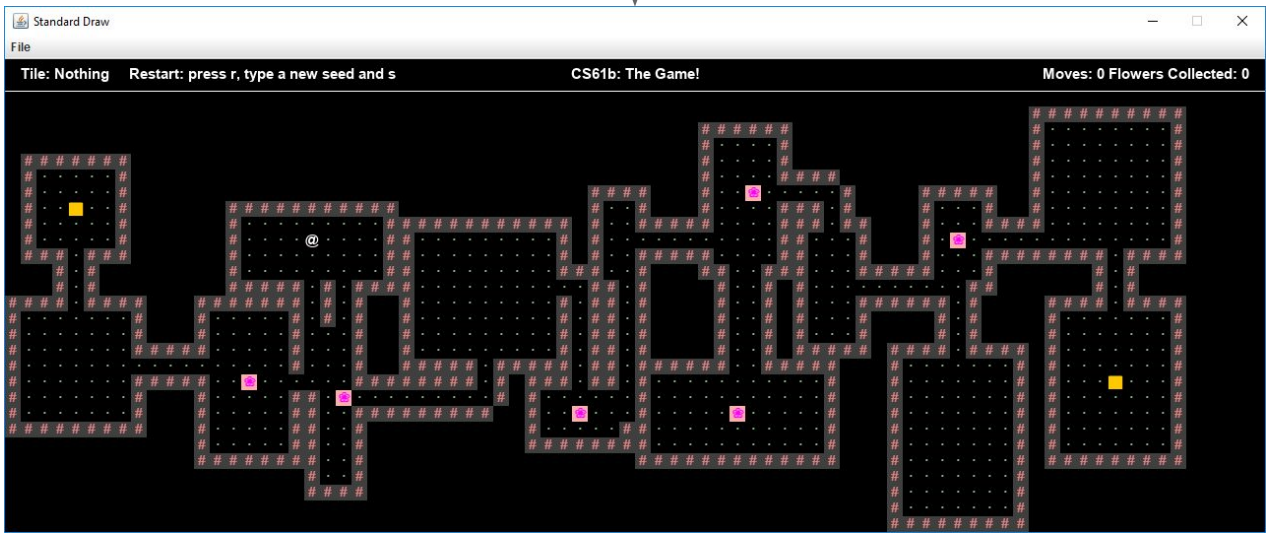
Part 2: Interactivity

In part 2, you'll add:

- An interactive keyboard mode.

```
interactWithKeyboard()
```

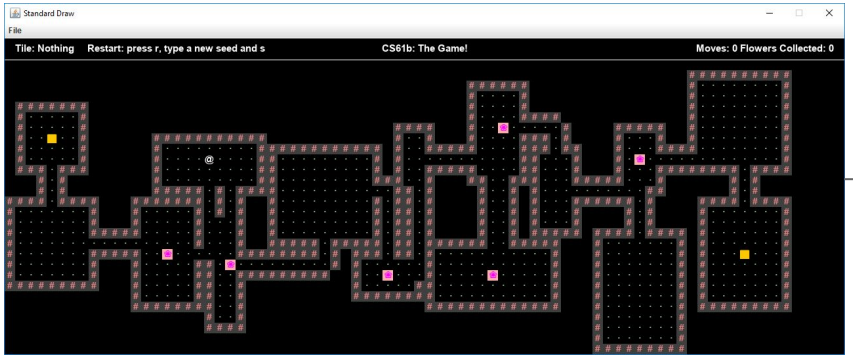
User types "N343434S"



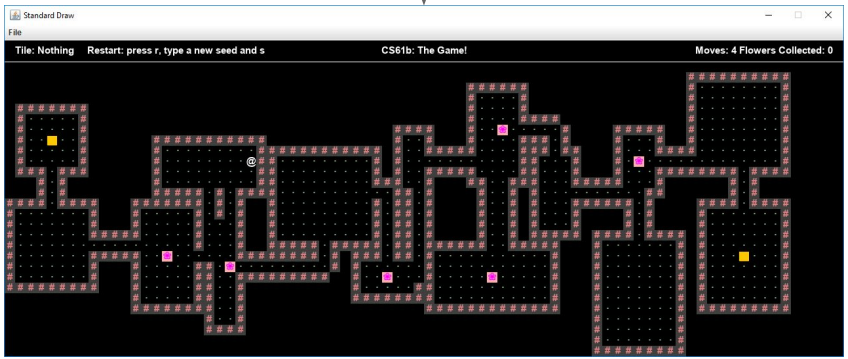
Part 2: Interactivity

In part 2, you'll add:

- An interactive keyboard mode.
- The ability for the avatar to move around in the world.



User types "dddd". Avatar moves four spaces east.

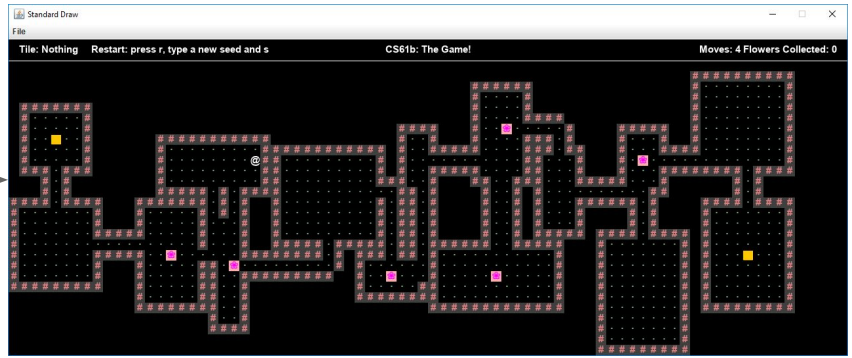


Part 2: Interactivity

In part 2, you'll add:

- An interactive keyboard mode.
- The ability for the avatar to move around in the world.
  - Must also be able to handle movements given via `interactWithInputString`.

Let's see an actual example where complexity got out of hand due to tactical programming.




```
interactWithInputString("N343434SDDDD")
```

The End Result of Tactical Programming

What is “complex” about this code?

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

The End Result of Tactical Programming

What is “complex” about this code?

- It’s repetitive.
- You can see everything that’s happening on a low level.
- It is very hard to change the code if you find a problem.

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

The End Result of Tactical Programming

What is “complex” about this code?

- Feels a little funny to have this same constant hard coded in so many places.
- The conditionals are long and hard to read.
- There’s no commenting -- which isn’t always a problem, but it is here maybe.

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

The End Result of Tactical Programming

What is “complex” about this code?

- Writing everything at a low level makes it very hard to spot errors and debug it. One approach is assigning intermediate calculations to named variables.
- Not very extensible, if things other than WALLS can block you, need more cases.

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

The End Result of Tactical Programming

What is “complex” about this code?

- Complex manual computation of west, east, north, and south.
- Lots of variables that need to be manipulated exactly so.
- Repetitive code (steps += 1, setting equal to PLAYER and FLOOR).

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Strategic Programming

Give some examples of changes you'd make to simplify this code.

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Strategic Programming

Give some examples of changes you'd make to simplify this code.

- Above all this code, have a single quartet of if statements that results in an targetX and targetY, and we use those variables just once.
- Method that help: occupied(WEST, here)

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Strategic Programming

Give some examples of changes you'd make to simplify this code.

- Avoid having the = PLAYER, = FLOOR in all statements, instead just do them after the loop.
- Instead of four if statements, create a function that computes EAST, WEST, UP, DOV

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Strategic Programming

Give some examples of changes you'd make to simplify this code.

- movePlayer(WEST)

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Strategic Programming

Give some examples of changes you'd make to simplify this code.

- move.equals("a") could simply set a variable equal to "WEST", "EAST", ...
- TETile getNeighbor("WEST") would return tile to the west.
- void move(player, world, "WEST") could move player tile to the west.

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Ousterhout's Take on Complexity

There are two primary sources of complexity:

- **Dependencies**: When a piece of code cannot be read, understood, and modified independently.
- **Obscurity**: When important information is not obvious.

Both of these happen in the code we just saw.

- You must remain vigilant to remain letting dependencies and obscurities from infesting your code.

The End Result of Tactical Programming

What is “complex” about this code?

- Complex manual computation of west, east, north, and south.
- Lots of variables that need to be manipulated exactly so.
- Repetitive code (steps += 1, similar logic in many places).

obscurity!

dependencies!

```
if (move.equals("a")
        && !world[player.xxcenter - 1][player.yycenter ].equals(Tileset.WALL)) {
    player.xxcenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter + 1][player.yycenter] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("s") &&
        !world[player.xxcenter][player.yycenter - 1].equals(Tileset.WALL)) {
    player.yycenter -= 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter][player.yycenter + 1] = Tileset.FLOOR;
    steps += 1;
}
if (move.equals("d")
        && !world[player.xxcenter + 1][player.yycenter].equals(Tileset.WALL)) {
    player.xxcenter += 1;
    world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
    world[player.xxcenter - 1][player.yycenter] = Tileset.FLOOR;
```

Another Example of Complexity

As mentioned earlier, in part 2, you'll add:

- An interactive keyboard mode.
- The ability for the avatar to move around in the world.
  - Must also be able to handle movements given via `interactWithInputString` as well as `interactWithKeyboard`.

Handling these two different sources of input in a non-complex way is tricky.

More Tactical Programming

In the solution below, `interactWithKeyboard` calls `moveKeyboard`, and `interactWithInputString` calls `moveChar`.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ....
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

More Tactical Programming

How could we simplify this code?

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ....
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

More Tactical Programming

How could we simplify this code?

- Take core logic and put it in a function that is shared.
- Have first one call the second one, pass the character.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ...
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

More Tactical Programming

How could we simplify this code?

- Make character some default value, check when the function is run, if is null, then request a character from keyboard.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ...
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

More Tactical Programming

How could we simplify this code?

- Put all move input into ONE place. So, have one source of input.
  - Using interface inheritance / dynamic method selection.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ...
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

More Tactical Programming

How could we simplify this code?

- My suggestion: Create an interface called `InputDevice` with a `nextChar()` method, and pass an `InputDevice` as an argument to the method.
  - Means you only need one `move` method that handles any `InputDevice`.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
    ....
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        player.yycenter += 1;
        world[player.xxcenter][player.yycenter] = Tileset.PLAYER;
        world[player.xxcenter][player.yycenter - 1] = Tileset.FLOOR;
        steps += 1;
    }
```

Don’t Be This Dog

Modular Design

Inspired partially by “A Philosophy of Software Design” chapters 4 and 5.

Hiding Complexity

One powerful tool for managing complexity is to design your system so that programmer is only thinking about some of the complexity at once.

- By using helper methods (i.e. `getNeighbor(WEST)`) and helper classes (`InputDevice`), you can hide complexity.

Modular Design

In an ideal world, system would be broken down into modules, where every module would be totally independent.

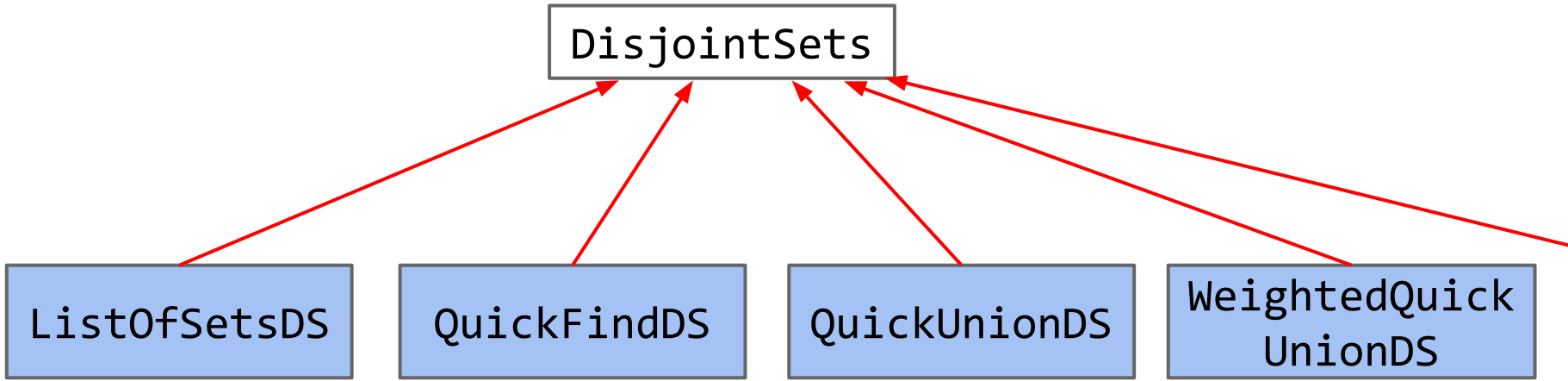
- Here, “module” is an informal term referring to a class, a package, or other unit of code.
- Not possible for modules to be entirely independent, because code from each module has to call other modules.
  - e.g. need to know signature of methods to call them.

In modular design, our goal is to minimize dependencies between modules.

Interface vs. Implementation

As we've seen, there is an important distinction between Interface and Implementation.

- Map is an interface.
- HashMap, TreeMap, etc. are implementations.



Interface vs. Implementation

Ousterhout: “The best modules are those whose interfaces are much simpler than their implementation.” Why?

- A simple interface minimizes the complexity the module can cause elsewhere. If you only have a `getNext()` method, that’s all someone can do.
- If a module’s interface is simple, we can change an implementation of that module without affecting the interface.
  - Silly example: If `List` had an `arraySize` method, this would mean you’d be stuck only being able to build array based lists.

Interface

A Java interface has both a formal and an informal part:

- Formal: The list of method signatures.
- Informal: Rules for using the interface that are not enforced by the compiler.
  - Example: If your iterator requires hasNext to be called before next in order to work properly, that is an informal part of the interface.
  - Example: If your add method throws an exception on null inputs, that is an informal part of the interface.
  - Example: Runtime for a specific method, e.g. `add` in `ArrayList`.
  - Can only be specified in comments.

Be wary of the informal rules of your modules as you build project 3.

Modules Should Be Deep

Ousterhout: “The best modules are those that provide powerful functionality yet have simple interfaces. I use the term *deep* to describe such modules.”

For example, the KdTree from project 2b is a deep module.

- Simple interface:
  - Constructor that takes a list of points.
  - `nearest` method that takes a point and returns closest point in set.
  - Nothing informal that user needs to know.
- Powerful functionality:
  - List converted into a complex searchable binary tree.
  - Nearest method has complex and subtle pruning rules for efficiency.

Information Hiding

The most important way to make your modules deep is to practice “information hiding”.

- Embed knowledge and design decision in the module itself, without exposing them to the outside world.

Reduces complexity in two ways:

- Simplifies interface.
- Makes it easier to modify the system.

Information Leakage

The opposite of **information hiding** is **information leakage**.

- Occurs when design decision is reflected in multiple modules.
  - Any change to one requires a change to all.
- Example: Code from before. Two move methods exist that process input in very similar ways.
  - Information is embodied in two places, i.e. it has “leaked”.

```
public void moveKeyboard(TETile[][] world) {
    String move = solicitOneCharsInput();
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        ...
    }
    ...
}

/** Method to call moves on the player when playing with input of string. */
public void moveChar(char m, TETile[][] world) {
    String move = Character.toString(m);
    if (move.equals("w") &&
            !world[player.xxcenter][player.yycenter + 1].equals(Tileset.WALL)) {
        ...
    }
    ...
```

Information Leakage

Ousterhout:

- “Information leakage is one of the most important red flags in software design.”
- “One of the best skills you can learn as a software designer is a high level of sensitivity to information leakage.”

Temporal Decomposition

One of the biggest causes of information leakage, especially on BYOW, is “temporal decomposition.”

In temporal decomposition, the structure of your system reflects the order in which events occur.

Common Bad Pattern in BYOW

Many students do something like this:

- Game is started with an input string, so call `interactWithInputString`.
  - Parse the string and find the seed by extracting N#####S.
  - Generate the world.
  - Process each character using `move(World, char)`.
  - …
- Game is started with no input string, so call `interactWithKeyboard`.
  - Display a menu and collect the seed.
  - Generate the world.
  - Until done, call `moveWithKeyboard(World)`.
  - …

Example, code that collects and extracts the seed should be shared.

Temporal decomposition leads to leaking information all over the place!

Summary and BYOW Suggestions

Some suggestions as you embark on BYOW:

- Build classes that provide functionality needed in many places in your code.
- Create “deep modules”, e.g. classes with simple interfaces that do complicated things.
- Avoid over-reliance on “temporal decomposition” where your decomposition is driven primarily by the order in which things occur.
  - It’s OK to use some temporal decomposition, but try to fix any information leakage that occurs!
- Be strategic, not tactical.
- Most importantly: Hide information from yourself when unneeded!

Teamwork

Project 3

Project 3 is a team project.

- This semester, we're doing teams of 2.

Two main reasons:

- Get practice working on a team.
- Get more creativity into the project since it's so open ended.

Ancillary reason: Also reduces programming workload per person, but the project is small enough that a single person can handle it.

Some material for this section of lecture drawn from www.teamingxdesign.com.

Teamwork is Hard

History of the software engineering project in 61B:

- 2015: Gitlet (solo)
- 2016: Editor (solo)
- 2017: Databases (partner)
- 2018: Build Your Own World (partner)

When I moved from solo to partner, I thought life would be strictly easier.

- In fact, 8% of 2017 partners were unhappy with their partnership!

Teamwork

In the real world, some tasks are much too large to be handled by a single person.

When faced with the same task, some teams succeed, where others may fail.

Individual Intelligence

In 1904, Spearman very famously demonstrated the existence of an "intelligence" factor in humans. As described in Woolley (2010):

- "People who do well on one mental task tend to do well on most others, despite large variations in the tests' contents and methods of administration." This mysterious factor is called "intelligence."
- Intelligence can be quickly measured (less than an hour).
- Intelligence reliably predicts important life outcomes over a long period of time, including:
  - Grades in school.
  - Success in occupations.
  - Even life expectancy.

Note: There is nothing in Spearman's work that says that this factor is genetic.

Group Intelligence

In the famous "Evidence for a Collective Intelligence Factor in the Performance of Human Groups", Woolley et. al investigated the success of teams of humans on various tasks.

They found that performance on a wide variety of tasks is correlated, i.e. groups that do well on any specific task tend to do very well on the others.

- This suggests that groups do have "group intelligence" analogous to individual intelligence as demonstrated by Spearman.

Group Intelligence

In the famous "Evidence for a Collective Intelligence Factor in the Performance of Human Groups", Woolley et. al investigated the success of teams of humans on various tasks.

Studying individual group members, Woolley et. al found that:

- Collective intelligence is not significantly correlated with average or max intelligence of each group.
- Instead, collective intelligence was correlated with three things:
  - Average social sensitivity of group members as measured using the "Reading the Mind in the Eyes Test" (this is really interesting).

Group Intelligence

In the famous "Evidence for a Collective Intelligence Factor in the Performance of Human Groups", Woolley et. al investigated the success of teams of humans on various tasks.

Studying individual group members, Woolley et. al found that:

- Collective intelligence is not significantly correlated with average or max intelligence of each group.
- Instead, collective intelligence was correlated with three things:
  - Average social sensitivity of group members as measured using the "Reading the Mind in the Eyes Test" (this is really interesting).
  - How equally distributed the group was in conversational turn-taking, e.g. groups where one person dominated did poorly.

Group Intelligence

In the famous "Evidence for a Collective Intelligence Factor in the Performance of Human Groups", Woolley et. al investigated the success of teams of humans on various tasks.

Studying individual group members, Woolley et. al found that:

- Collective intelligence is not significantly correlated with average or max intelligence of each group.
- Instead, collective intelligence was correlated with three things:
  - Average social sensitivity of group members as measured using the "Reading the Mind in the Eyes Test" (this is really interesting).
  - How equally distributed the group was in conversational turn-taking, e.g. groups where one person dominated did poorly.
  - Percentage of females in the group (paper suggests this is due to correlation with greater social sensitivity).

Teamwork and Project 3

Presumably, learning habits that lead to greater group intelligence is possible.

- We hope that project 3 helps with this.

Recognize that teamwork is also about relationships!

- Treat each other with respect.
- Be open and honest with each other.
- Make sure to set clear expectations.
- … and if those expectations are not met, confront this fact head on.

… to be continued Monday

Reflexivity

Important part of teamwork is “reflexivity”.

- “A group’s ability to collectively reflect upon team objectives, strategies, and processes, and to adapt to them accordingly.”
- Recommended that you “cultivate a collaborative environment in which giving and receiving feedback on an ongoing basis is seen as a mechanism for reflection and learning.”
    - It’s OK and even expected for you and your partner to be a bit unevenly matched in terms of programming ability.

You might find this description of best practices for team feedback useful, though it’s targeted more towards larger team projects.

- Some key ideas from this document follow.

Feedback is Hard: Negativity

Most of us have received feedback from someone which felt judgmental or in bad faith.

- Thus, we're afraid to give even constructive negative feedback for fear that our feedback will be misconstrued as an attack.
- And we're conditioned to watch out for negative feedback that is ill-intentioned.

Do you have any examples?

- If you have feedback that is not actually actionable, it feels bad.
- In high school: Offended every time someone would edit my essays -- put lots of work, and felt like the person giving feedback was saying you didn't work hard enough.
- Advisors said I am not making enough progress fast enough. Think I should

Feedback is Hard: Can Seem Like a Waste of Time

Feedback also can feel like a waste of time:

- You may find it a pointless exercise to rate each other twice during the project. What does that have to do with a programming class?
- In the real world, the same thing happens. Your team has limited time to figure out “what” to do, so why stop and waste time reflecting on “how” you’re working together?

Feedback is Hard: Coming Up With Feedback is Tough

Feedback can simply be difficult to produce.

- You may build incredible technical skills, but learning to provide useful feedback is hard!
- Without confidence in ability to provide feedback, you may wait until you are forced to do so by annual reviews or other structured time to provide it.
  - If you feel like your partnership could be better, try to talk about it without waiting until you have to review each other at the end of next week.

Team Reflection

We are going to have you reflect on your team's success twice:

- This partnership has worked well for me.
- This partnership has worked well for my partner.
- Estimate the balance of work between you and your partner, briefly explain.
  - 50/50 means you estimate you each contributed about equally.
  - Note, contributing does not mean lines of code. We mean contribution in a more general sense.
- If applicable, give a particularly great moment from your partnership.
- What's something you could do better?
- What's something your partner could do better?

Note: Except in extreme cases, we will not be penalizing partners who contributed less!