Quantifying Code 1

Measuring the behavior and performance of software

We train to think as engineers
about the **problems**
we're trying to solve

We also need to think as engineers
about the **process (code)**
we use to solve them

Logging

Parameterization

Profiling

Testing

Workflow

Logging text, structured

Parameterization config files

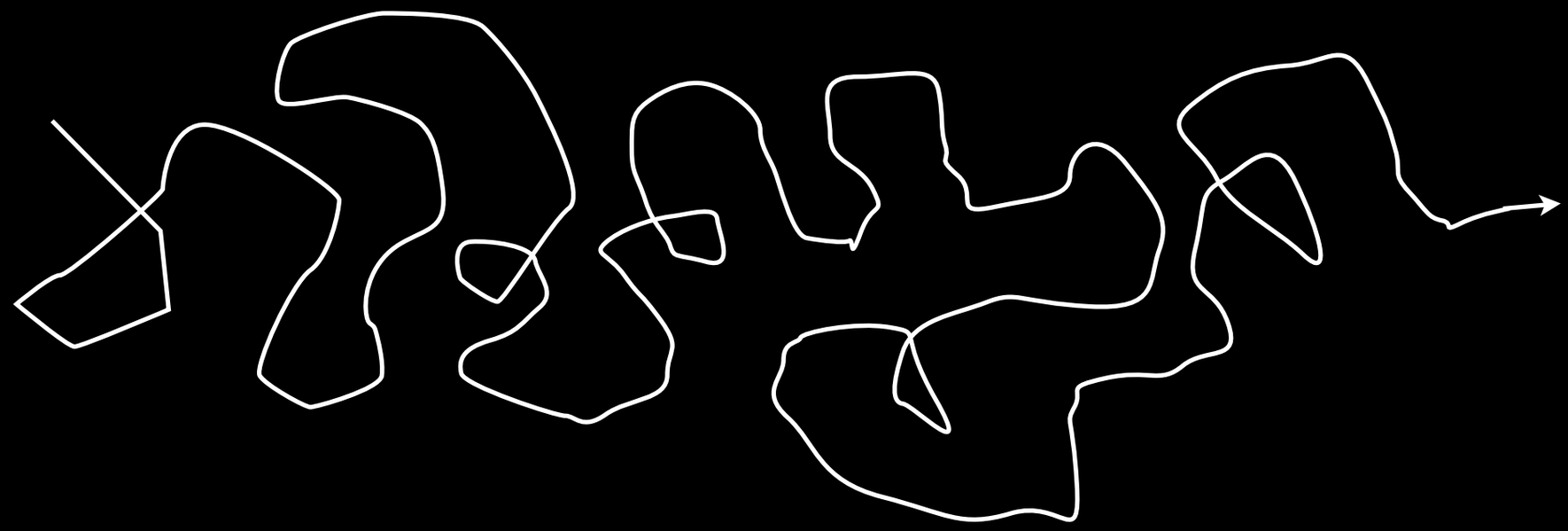
Profiling automatic, manual

Testing styles

Workflow how it fits together

This will help you avoid

Bug hunting process that looks like



Vague characterizations like:

- “it’s fast!”
- “this doesn’t take much memory”
- “it seems to work”

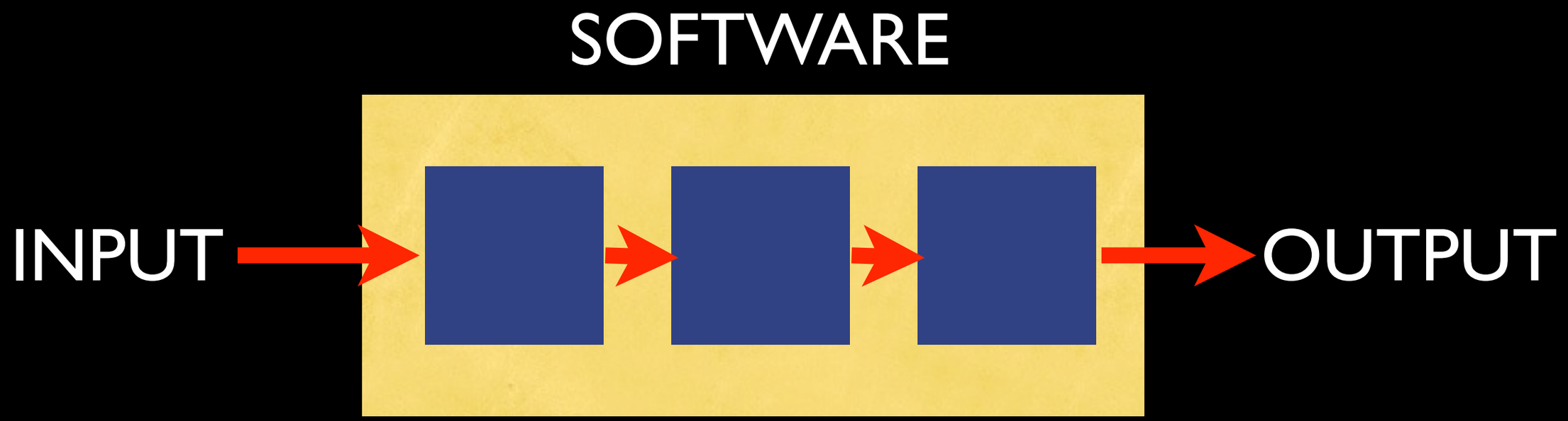
This will help you create

Reproducible results (science!)

Specific knowledge about performance characteristics

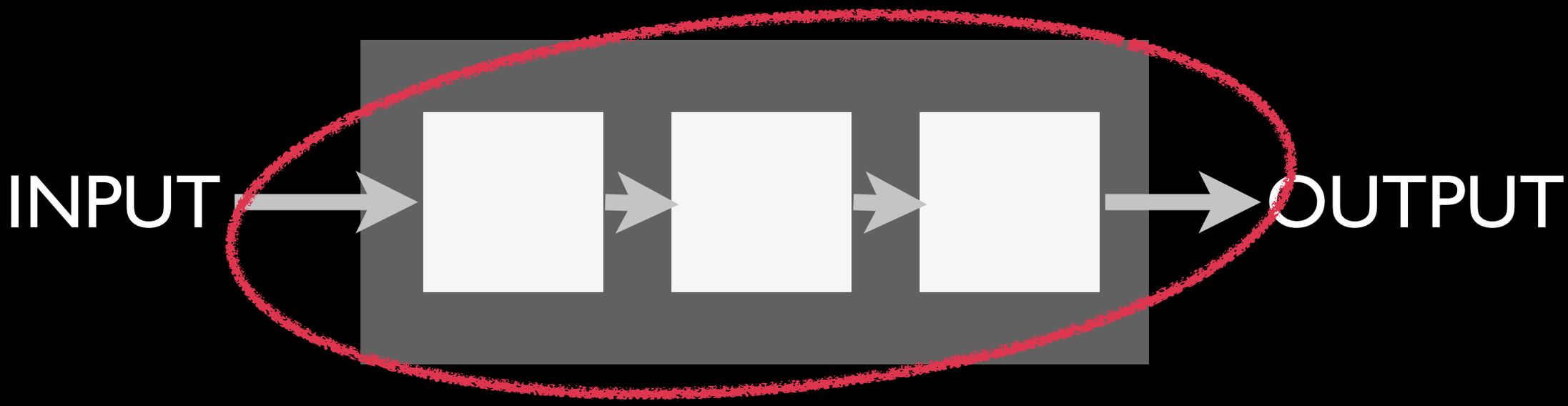
-> helps you know what to improve first!

Archival data about program performance and behavior



Goal: Measure every step of the way

Logging



what's wrong with

```
System.out.println("Value of x:" + x);
```

?

short term solutions like

```
System.out.println("Value of x:" + x);
```

- Can't be shipped
- Don't scale with the size of your software
- Performance drag
- Hard to sort through
- Don't provide "archival"-quality metadata

Logging Libraries

Provide structure

Time | File/Class | Level | Message

Provide “levels”

TRACE DEBUG INFO WARN ERROR FATAL

Provide ecosystems for storage, analysis, collection.

Structure

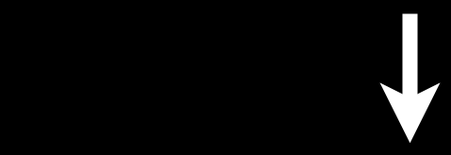
You provide

Time	File/Class	Level	Message

Most logging libraries provide a plethora of output formatting & metadata options.

E.g., you can attach system information, memory information, etc.

```
log.warn(“I’m sorry, Dave. I’m afraid I can’t do that”);
```



```
WARN 2001-8-8 2:30PM [HAL.openPodBayDoors()] [127.0.0.1]
I’m sorry, Dave. I’m afraid I can’t do that.
```

post hoc analysis

Log files provide after-the-fact analysis.

- Many real-world tasks take hours to complete, or run without a human present.

- Entire languages have been created to process logs e.g. Sawzall

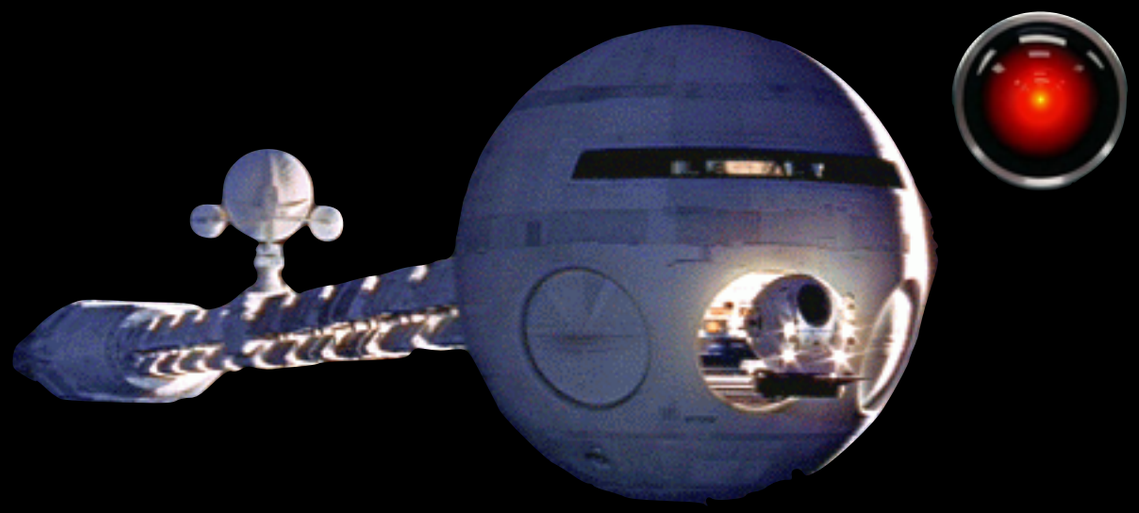
- Can configure different ***sinks*** for different types of logging events

What are some questions you might be able to ask of logs?

What are some questions you might be able to ask of logs?

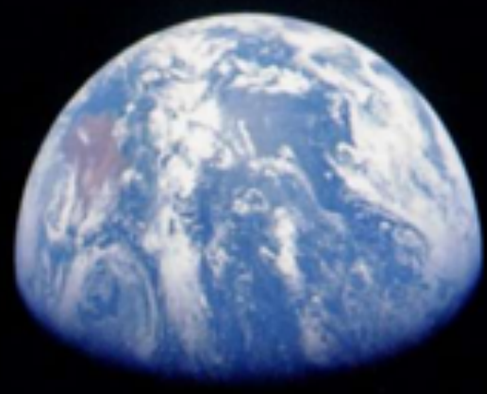
- How many users per day experience a fatal error?
- How many users trigger a warning
  (e.g., a default value is used when it really shouldn't be)
- Are errors coming from a particular computer?
  Maybe it has a bad hard drive
- What is the distribution of errors per class?
  Maybe development efforts can be prioritized this way

monitoring & debugging distributed systems

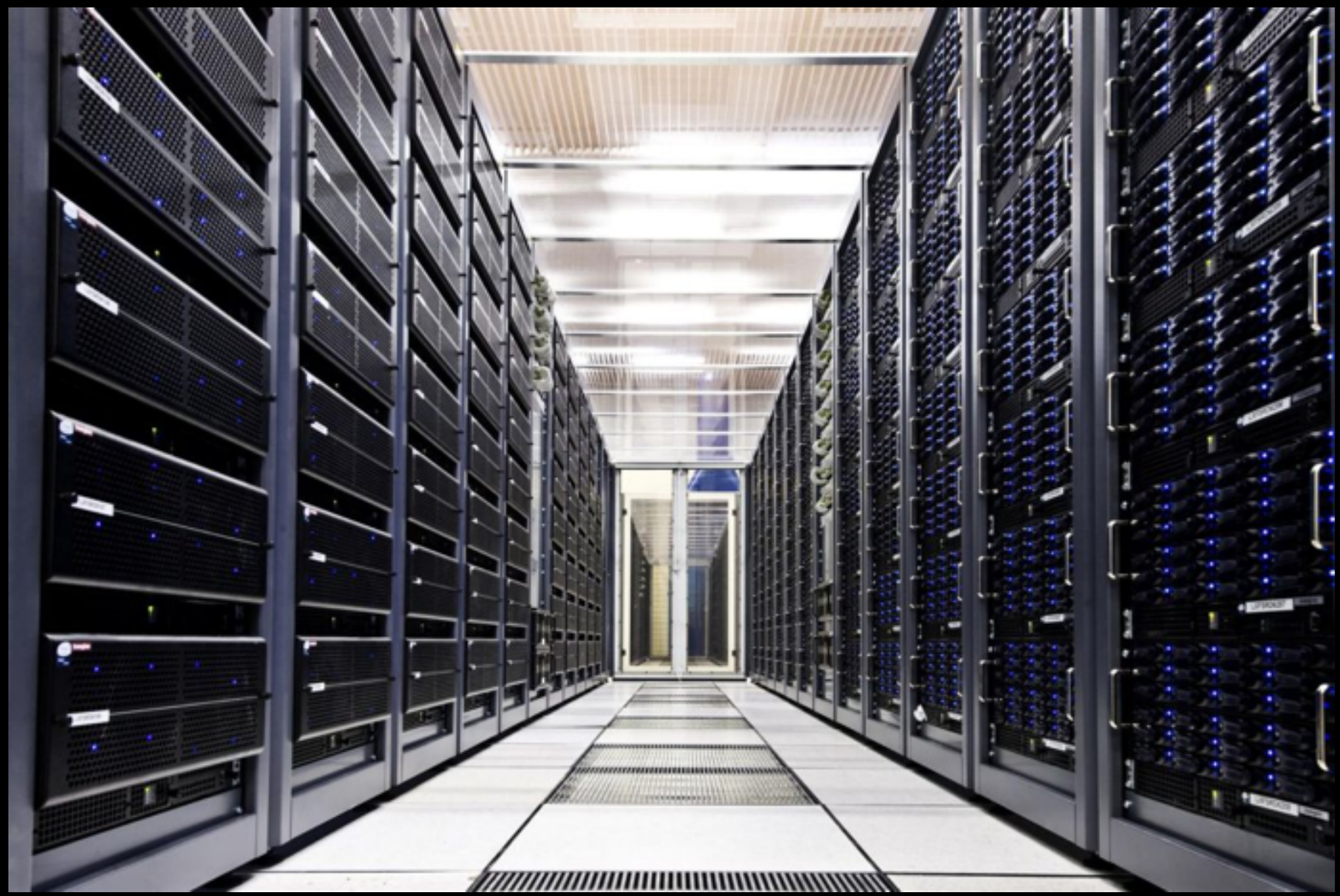


```
log.warn("I'm sorry, Dave.");
```

Remote logging sink



... important when modern day computing looks like this



You've got 100,000s of machines involved.
You can't personally examine what's going on -- programs have to do it for you.

performance

Performance

What might be good about this line?

```
if (log.debugEnabled()) {
  log.debug(“Total prob mass:” + this.pmass());
}
```

```
if (log.debugEnabled()) {
  log.debug("Total prob mass:" + this.pmass());
}
```

1. `this.pmass()` might be an **expensive** computation.

2. The output is sent to a logger object, instead of STDOUT.

This enables more efficient data management than simply dumping to a video card.

Performance matters, even with quick logging statements

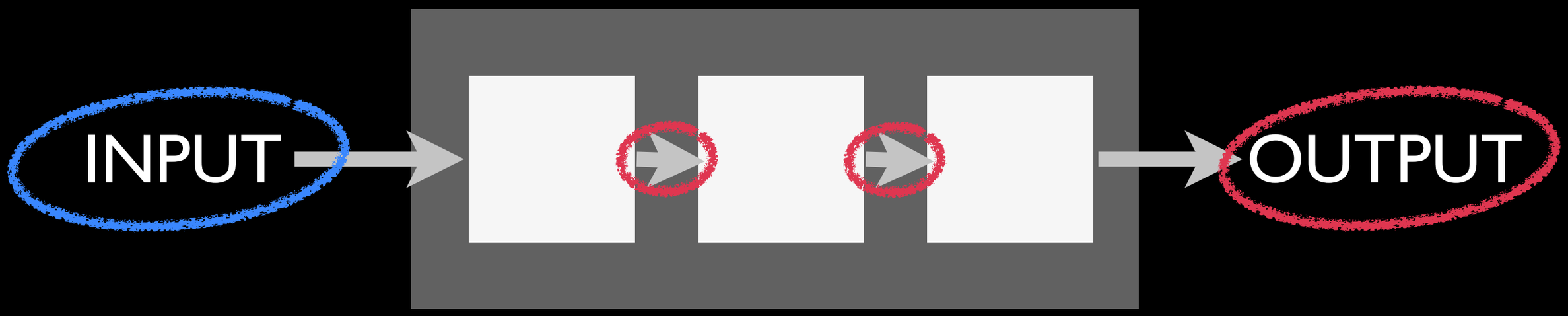
```
if (log.debugEnabled()) {
  log.debug(“Total prob mass:” + this.pmass());
}
```

per word, per tweet, per iteration of a learning process

1M tweets
~30 words per tweet
100 topics per word
1,000 times

-> 3 Trillion Times

Parameterization & Structured Logging



Sometimes text isn't the right data structure

- Matrices
- Arrays
- Images, files, properties, etc

Structured logging is very useful for research code

Some systems support this by enabling JSON-like objects to be passed to the logger.

Instead we'll talk about a project structure, and set of practices, you can use for more complicated recording.

If you decide to go into a research-related field, this will save you many all-nighters, I promise.

```
PROJECT-ROOT
|
*- src/
|
*- lib/
```

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
```

<-- Configuration stored as files

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
   |
   * bare-bones.conf
```

Every language has numerous configuration management libraries. Pick one and learn it.

```
[modelParams]
alpha: 0
theta: 0

[inputs]
tweets: data/tweets-2012.txt

[preProcessing]
englishOnly: yes
```

At run time...

```
x *= Options.modelParams.alpha

          - or -

if (Options.preProcessing.englishOnly) {
   ...
}
```

```
[modelParams]
alpha: 0
theta: 0

[inputs]
tweets: data/tweets-2012.txt

[preProcessing]
englishOnly: yes
```

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
|
*- experiments/
```

<-- Each run of the program persists data to experiments/

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
|
*- experiments/
   |
   *-2013-03-18/
     |
     *-001.run/ <-- directory for Experiment #1 on yyyy-mm-dd
```

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
|
*- experiments/
   |
   *-2013-03-18/
     |
     *-001.run/ <-- directory for Experiment #1 on yyyy-mm-dd
```

```
$ ./run-project
```

```
PROJECT-ROOT
|
*- src/
|
*- lib/
|
*- config/
|
*- experiments/
   |
   *-2013-03-18/
     |
     *-001.run/ <-- directory for Experiment #1 on yyyy-mm-dd
     |
     *-002.run/ <-- directory for Experiment #2 on yyyy-mm-dd
```

The *standard log* goes here
As well as all program:
input, output, and objects

Create an **Experiment** singleton
that manages these directories

Experiment.begin(configurationFile**)**

Creates a new directory
Copies in file inputs
Copies in the configuration file
Records git status

Experiment.rerun(date, run#**)**

Experiment.end()

Add structured logging methods to
Experiment

.saveFile(file**)**

.saveArray(arr, filename**)**

.saveMatrix(arr, filename**)**

.saveDirectory(dir, zip_filename**)**

... all of which place the data in the
experiment directory

Timers

.tick(timerName**)** **.tock(**timerName**)**

```
Experiment.tick(“loadHugeDataFile”)

data = new SparseMatrix(Config.inputFile)

Experiment.tock(“loadHugeDataFile”)
```

timers.txt

```
loadHugeDataFile 20.3s
```

Counters

.inc(counterName, amount=1**)**

```
for (tweet <- tweets) {
  if (! tweet.isEnglish)
    Experiment.inc(“foreignTweets”)
}
```

counters.txt

```
foreignTweets 23121
```

The counter can get really fancy, enabling all sorts of multi-level counting, normalization, histogram output, etc.

Git Integration

.verifyCodeStatus

FAIL to run unless all changes are committed.

Tags the git repository with Date/Experiment#

You can get really fancy.

Develop a personal toolbelt for your languages of choice.

[scala] github.com/eob/researchy

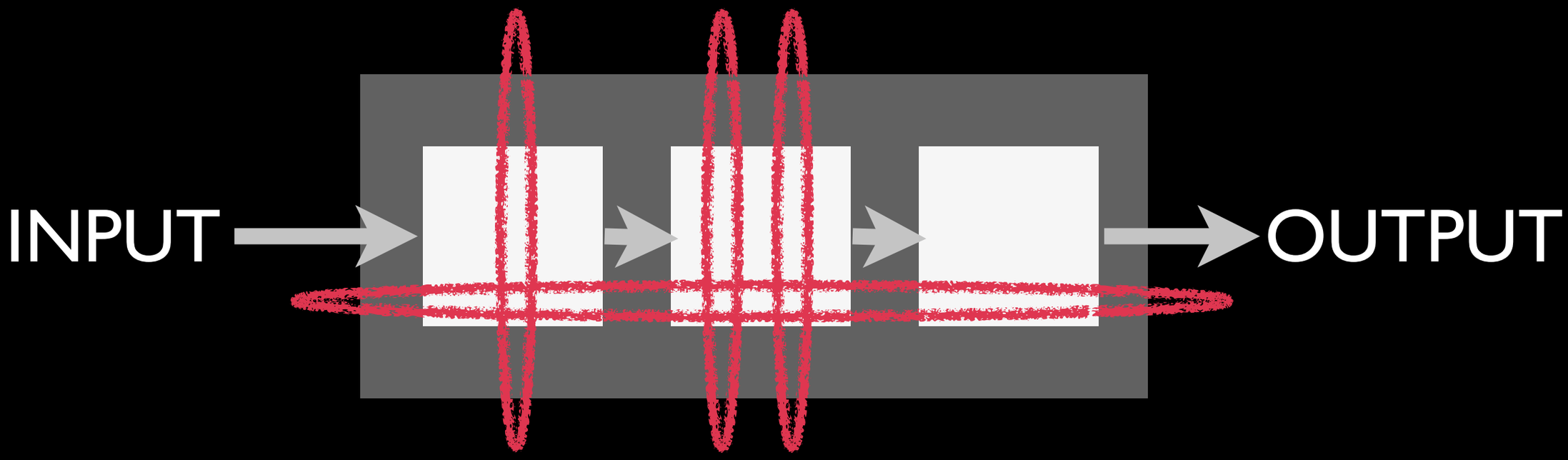
Measuring Timing

```
function measureTime(otherFunction, iterations)
{
  var start = new Date().getTime();

  for (var i = 0; i < iterations; i++) {
    otherFunction();
  }

  var end = new Date().getTime();
  return (end - start);
}
```

Why measure total time instead of per-iteration time?

(js example)

Profiling



Where is **time**
memory
allocations
..etc.. being spent?

Answers the question:

"How do we make this {faster, smaller}"
Results can be surprising

Profiling is usually done through instrumentation of your code

- Manual
- Automatic

- *(coding example)*
- *(string example w/ Cougaar)*