6.170 Lecture 12
Debugging

MIT EECS

Michael Ernst

Saman Amarasinghe

three related notions

Validation

Purpose is to uncover problems and increase confidence

Combination of reasoning and test

Debugging

Finding out why a program is not functioning as intended

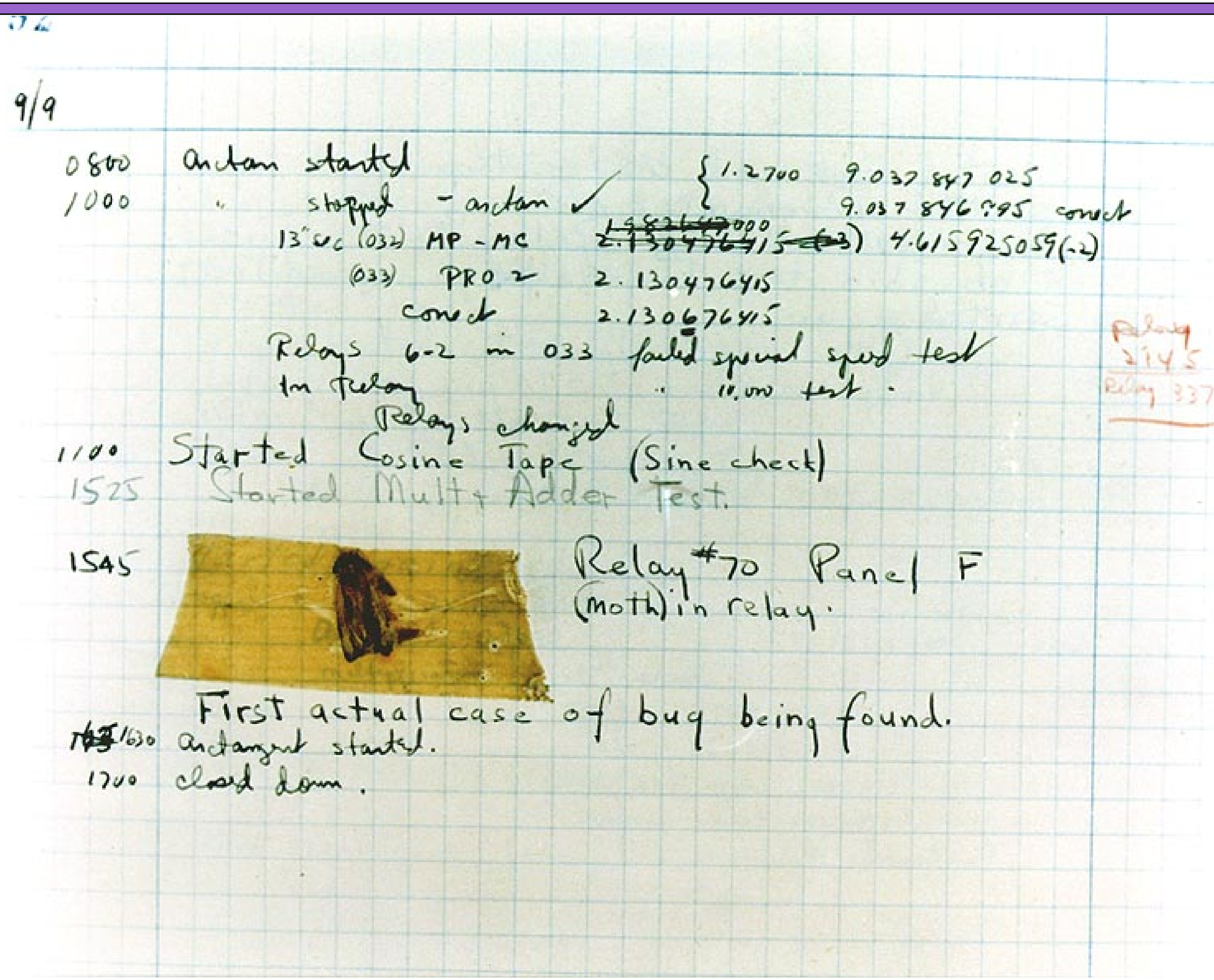
Defensive programming

Programming with validation and debugging in mind

Testing ≠ debugging

test:	reveals existence of problem
debug:	pinpoint location+cause of problem

a bug – September 9, 1947



defense in depth

First defense against bugs is to not make them

Correctness: get things right first time

Second defense is to make bugs immediately visible

Local visibility of errors: if things fail, we'd rather they fail loudly and immediately – e.g. checkRep()

Last resort is debugging

Needed when effect of bug is distant from cause

Design experiments to gain information about bug

- Fairly easy, in a program with good modularity, representation hiding, specs, unit tests etc.
- Much harder and more pain-staking with a poor design, e.g. with rampant rep exposure

first defense: correctness

Get things right first time

Don’t code before you think! Think before you code.

If you're making lots of easy-to-find bugs, you're also making hard-to-find bugs – don't use compiler as crutch

Simplicity is key

Modularity

- Divide program into chunks that are easy to understand
- Use abstract data types with well-defined interfaces
- Use defensive programming; avoid rep exposure

Specification

- Write specs for all modules, so that an explicit, well-defined contract exists between each module and its clients

second defense: immediate visibility

If we can't prevent bugs, we can try to localize them to a small part of the program

- Assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation
- Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (unless it's in the test driver)
- Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed

When localized to a single method or small module, bugs can be found simply by studying the program text

why is this good?

- Key difficulty of debugging is to find the code fragment responsible for an observed problem
  - A method may return an erroneous result, but be itself error free, if there is prior corruption of representation
- The earlier a problem is observed, the easier it is to fix
  - For example, frequently checking the rep invariant helps the above problem
- General approach: fail-fast
  - Check invariants, don't just assume them
  - Don't try to recover from bugs – this just obscures them

don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (true) {
    if (a[i]==k) break;
    i++;
}
```

This code fragment searches an array a for a value k.

- Value is guaranteed to be in the array.
- If that guarantee is broken (by a bug), the code throws an exception and dies.

Temptation: make code more "robust" by not failing

don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
```

Now at least loop will always terminate

- But no longer guaranteed that `a[i]==k`
- If rest of code relies on this, then problems arise later
- *All we've done is obscure the link between the bug's origin and the eventual erroneous behavior it causes.*

don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
assert (i<a.length) : "key not found";
```

Assertions let us document and check invariants

- Abort program as soon as problem is detected

Use built-in Java assertions, or junit framework

- Drawback to built-in: ignored unless `-ea` flag is given

assertions

If it can't happen, use assertions to ensure that it won't

(Hunt&Thomas, "The Pragmatic Programmer")

Figure out conditions you expect to hold

Then, don't just assume them, assert them

Guidelines

Add assertions as you write, not later

But *not* to check the obvious

```
x = y + 1;
assert (x == y + 1); // don't do this
```

And *not* to check resource limitations (these are not bugs)

Novices usually under-assert

responding to failure

How should a program respond to a detected failure?

- Try to transparently fix the failure?
  - Hard to do, and often just makes problems even more obscure (as we've seen)
- Record and Continue?
- Abort the program?
  - Exactly how to do this is program dependent
  - Word processor should offer to save files
  - Rocket controller should try to minimize damage
- *Hard to decide correct action locally*

Often want to pass responsibility back to caller

- *Return null, -1, or other special value to signal error*
- *But this can introduce bugs, silently contaminate data*

exceptions

- Exceptions let us bypass normal control flow
  - No risk of confusion with normal data
  - Two flavors in Java: checked or unchecked
  - See **Bloch chapter 8 (#39 – #47)** for best practice
- Use an "unchecked" exception if:
  - There is a convenient way for the client to avoid ever triggering the exception
    - So forcing the client to check for the exception is redundant
  - Or if the exception reflects an unexpected failure
    - Nothing the client can reasonably do, e.g. broken rep
- Otherwise use a "checked" exception
  - Compiler forces client to deal with such exceptions

exceptions examples

E Queue.remove()

throws `NoSuchElementException` [unchecked]

- Retrieves and removes the head of the queue.
- Expects that client will only call method if `Queue` is non-empty, since client can easily call `isEmpty()` if needed
- So forcing client to catch exception would be a burden

FileInputStream.FileInputStream(String name)

throws `FileNotFoundException` [checked]

- Opens a file for input
- Forces client to consider exception, since there is no easy way to check if the file will exist at time of opening (could be deleted externally after any check)

last resort: debugging

- Bugs happen
  - Industry average: 10 bugs per 1000 lines of code (“kloc”)
- Bugs that are not immediately localizable happen
  - Found during integration testing
  - Or reported by user
- Here's how we deal with such failures
  - step 1 – Clarify symptom
  - step 2 – Find and understand cause
  - step 3 – Fix
  - step 4 – Do regression

the debugging process

step 1 – find a small, repeatable test case that produces the failure (may take effort, but helps clarify the bug, and also gives you something for regression)

- *don't move on to next step until you have repeatable test*

step 2 – narrow down location and proximate cause

- study the data / hypothesize / experiment / repeat
- may change code to get more information
- *don't move on to next step until you understand cause*

step 3 – fix the bug

- Is it a simple typo, or design flaw? Does it occur elsewhere?

step 4 – add test case to regression; run regression to see if:

- (a) the bug appears to be fixed
- (b) no new bugs have been introduced

example

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A,B s.t. full=A+sub+B)
boolean contains(String full, String sub);
```

User reports that method sometimes fails

Points out that it can't find the string `"very happy"` within:

`"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`

Wrong response:

See accented characters, panic about not having thought about unicode, and go diving for your Java texts to see how that is handled.

example

Right response – clarify symptom

Find good, simple test case

Pare test down – can't find **`"very happy"`** within:

- **`"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`**
- **`"I am very very happy to see you all."`**
- **`"very very happy"`**

CAN find **`"very happy"`** within:

- **`"very happy"`**

Can't find **`"ab"`** within **`"aab"`**

(We saw what might cause this bug in lecture 3)

example

Sometimes it is helpful to find two almost identical test cases where one gives the correct answer and the other does not

Can't find **"very happy"** within:

- **"I am very very happy to see you all."**

Can find **"very happy"** within:

- **"I am very happy to see you all."**

general strategy

In general: find simplest input that will provoke bug

- Usually not the input that revealed existence of the bug

Start with data that revealed bug

- Keep paring it down (binary search can help)
- Often leads directly to an understanding of the cause

When not dealing with simple method calls

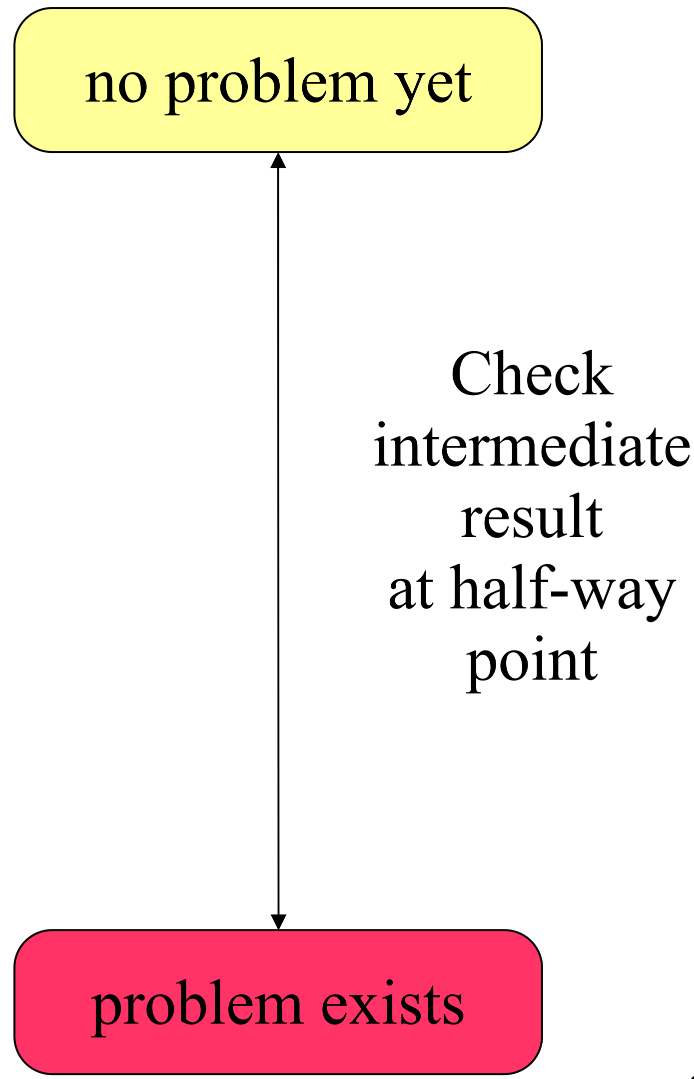
- Think of "test input" as the set of steps needed to reliably trigger the bug
- Same basic idea

searching for bugs

- Take advantage of modularity
  - Start with everything, take away pieces until bug goes
  - Start with nothing, add pieces back in until bug appears
- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results
- Can use binary search to speed things up
  - Bug happens somewhere between first and last statement
  - So can do binary search on that ordered set of statements

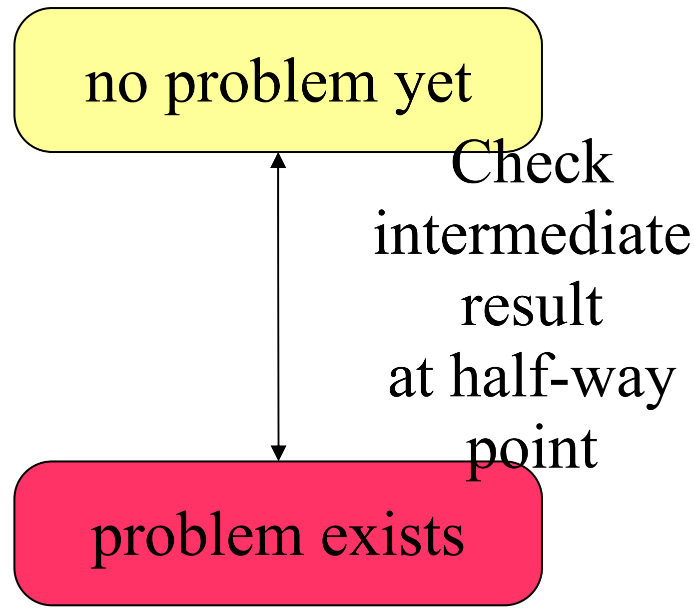
binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```



binary search on buggy code

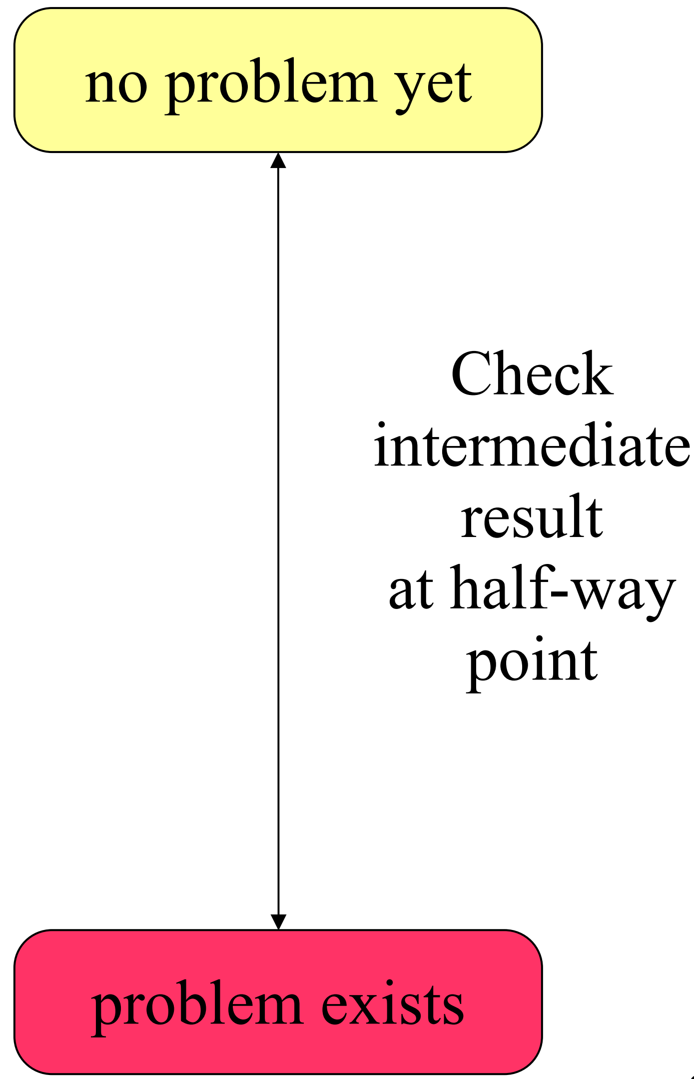
```
public class MotionDetector {
   private boolean first = true;
   private Matrix prev = new Matrix();

   public Point apply(Matrix current) {
      if (first) {
         prev = current;
      }
      Matrix motion = new Matrix();
      getDifference(prev,current,motion);
      applyThreshold(motion,motion,10);
      labelImage(motion,motion);
      Hist hist = getHistogram(motion);
      int top = hist.getMostFrequent();
      applyThreshold(motion,motion,top,top);
      Point result = getCentroid(motion);
      prev.copy(current);
      return result;
   }
}
```



Quickly home in
on bug in O(log n) time
by repeated subdivision

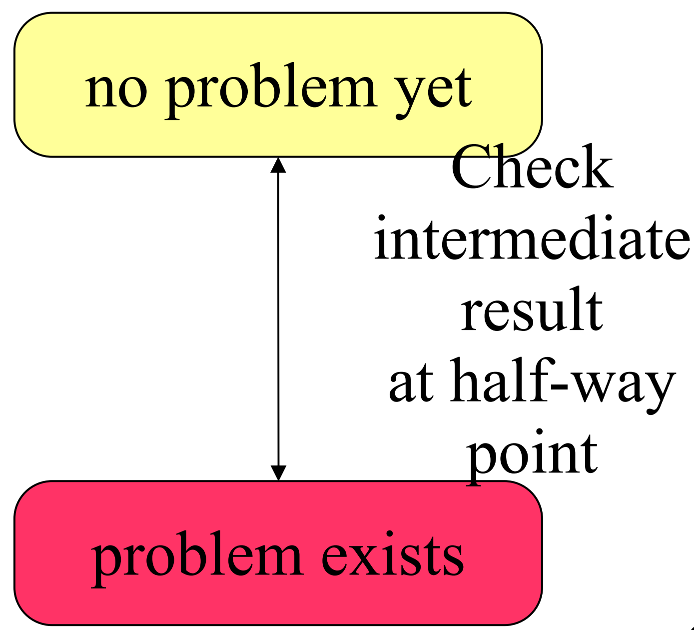
binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;  first = false;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```



binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;  first = false;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```



regression testing

- Whenever you find and fix a bug
  - Add a test for it
  - Re-run all your tests
- Why this is a good idea
  - Often reintroduce old bugs while fixing new ones
  - Helps to populate test suite with good tests
  - If a bug happened once, it could well happen again
- Run regression tests as frequently as you can afford to
  - Automate process
  - Make concise test sets, with few superfluous tests

keep in mind

- The bug is not where you think it is
  - Ask yourself where it cannot be; explain why
- Try simple things first, e.g.,
  - Reversed order of arguments: Collections.copy(src,dest)
  - Spelling of identifiers: int hashcode()
    - @override can help catch method name typos
  - Same object vs. equal: a == b versus a.equals(b)
  - Failure to reinitialize a variable
  - Deep vs. shallow copy
- Make sure that you have correct source code
  - Recompile everything

when the going gets tough

- Reconsider assumptions
  - E.g., has the OS changed? Is there room on the hard drive?
  - Debug the code, not the comments
- Start documenting your system
  - Gives a fresh angle, and highlights area of confusion
- Get help
  - We all develop blind spots
  - Explaining the problem often helps
- Walk away
  - Trade latency for efficiency – **sleep**!
  - One good reason to start early

Detecting Bugs in the Real World

- Real Systems are…
  - Large and complex (duh!)
  - Collection of modules, written by multiple people
  - Complex input
  - Many external interactions
  - Non-deterministic
- Replication can be an issue
  - Infrequent bug
  - Instrumentation eliminates the bug
- Bugs cross abstraction barriers
- Large time lag from corruption to detection