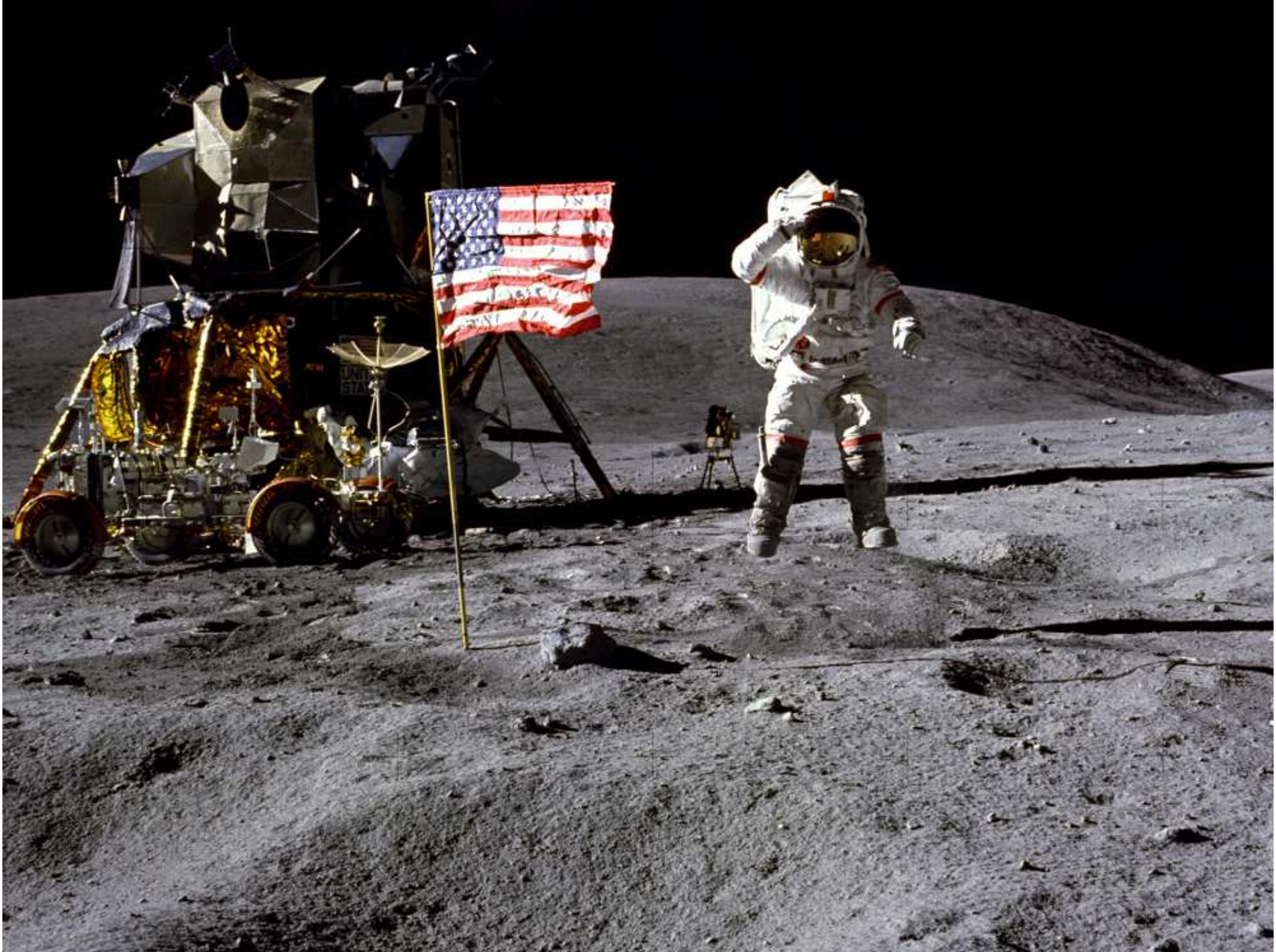
Victoria University
of Wellington, New Zealand
Te Whare Wananga o te Upoko o te Ika a Maui
Aotearoa

SWEN221: Software Development

#3 - Debugging

David J. Pearce & Nicholas Cameron & James Noble & Petra Malik
Computer Science, Victoria University



102:38:26 Armstrong: (With the slightest touch of urgency) Program Alarm.

102:38:28 Duke: It's looking good to us. Over.

102:38:30 Armstrong: (To Houston) It's a 1202.

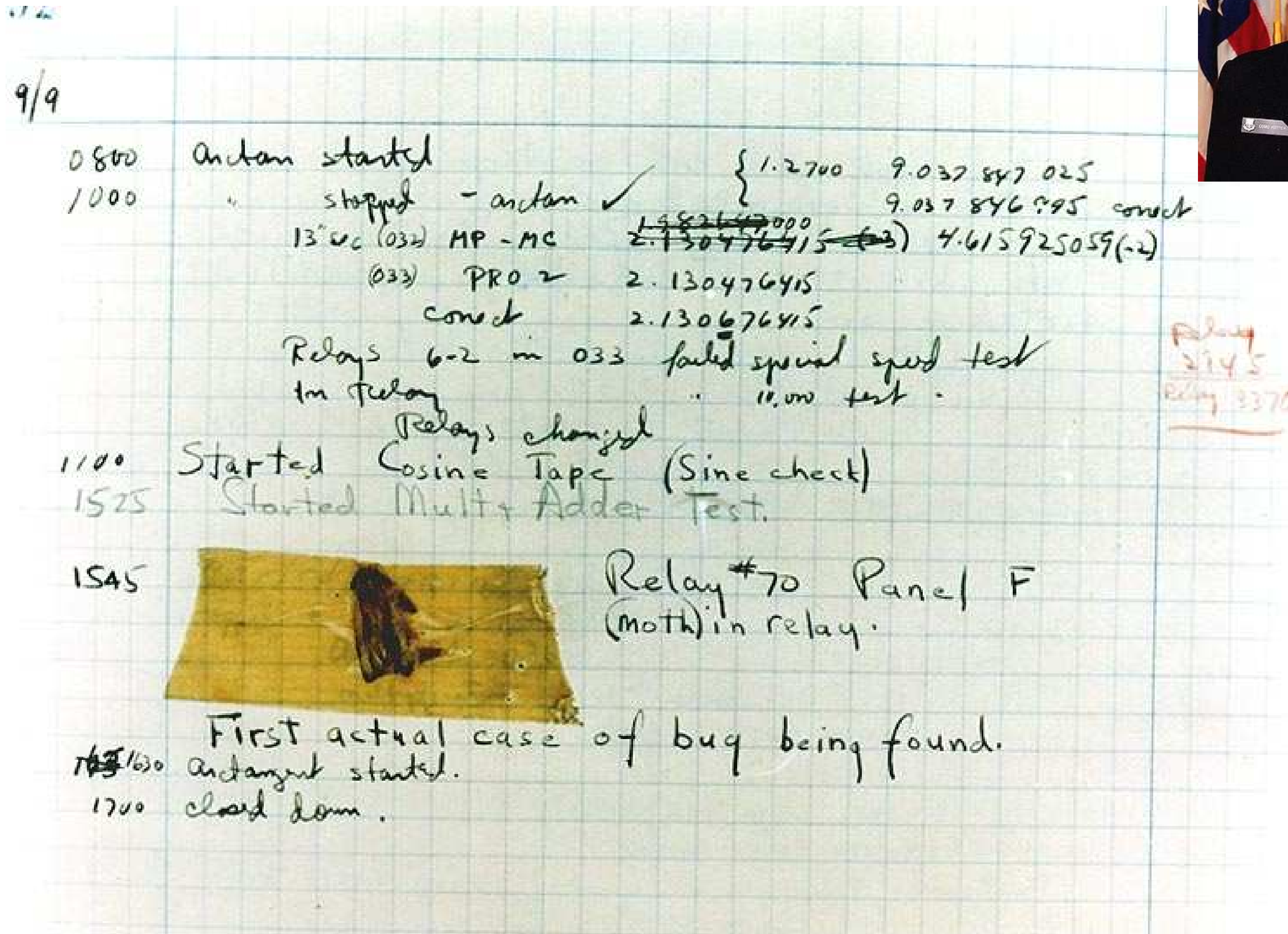
102:38:32 Aldrin: 1202. (Pause)

[Altitude 33,500 feet]

102:38:42 Armstrong (on-board): (To Buzz) What is it? Let's incorporate (the landing radar data). (To Houston) Give us a reading on the 1202 Program Alarm.

[The 1202 program alarm is being produced by data overflow in the computer. It is not an alarm that they had seen during simulations but, as Neil explained during a post-flight press conference "In simulations we have a large number of failures and we are usually spring-loaded to the abort position. And in this case in the real flight, we are spring-loaded to the land position."]

Rear Admiral Grace Hopper



1947, Mark II Relay Calculator

Debugging

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

- Process of finding and eliminating bugs
- Programmers spend more time debugging than writing new code
- Often, locating the **defect** is hardest

Example

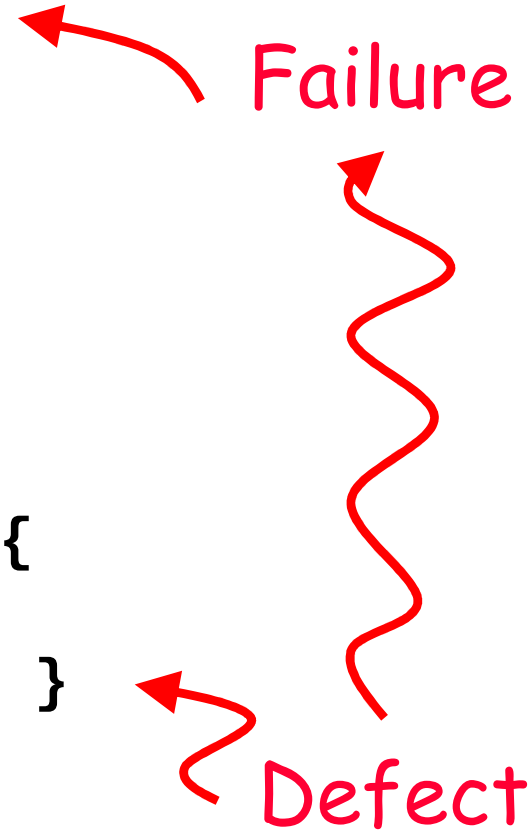
```
class ProblemCase {
 /**
  * @param input – should not be null
  */
 public static char[] convert(String input) {
  char[] cs = new char[input.length()];

  for(int i=0;i!=input.length();++i) {
   cs[i] = input.charAt(i);
  }
  return cs;
 }
 public static void main(String[] args) {
  String input = null;
  if(args.length > 0) { input = args[0]; }
  char[] bs = convert(input);
  …
}}
```

Terminology

- **Defect**: Error in code created by programmer
- **Infection:** Error in program state
- **Propagation:** – Bad program state leads to more bad states
- **Failure:** Program finally does something wrong

Example Revisited

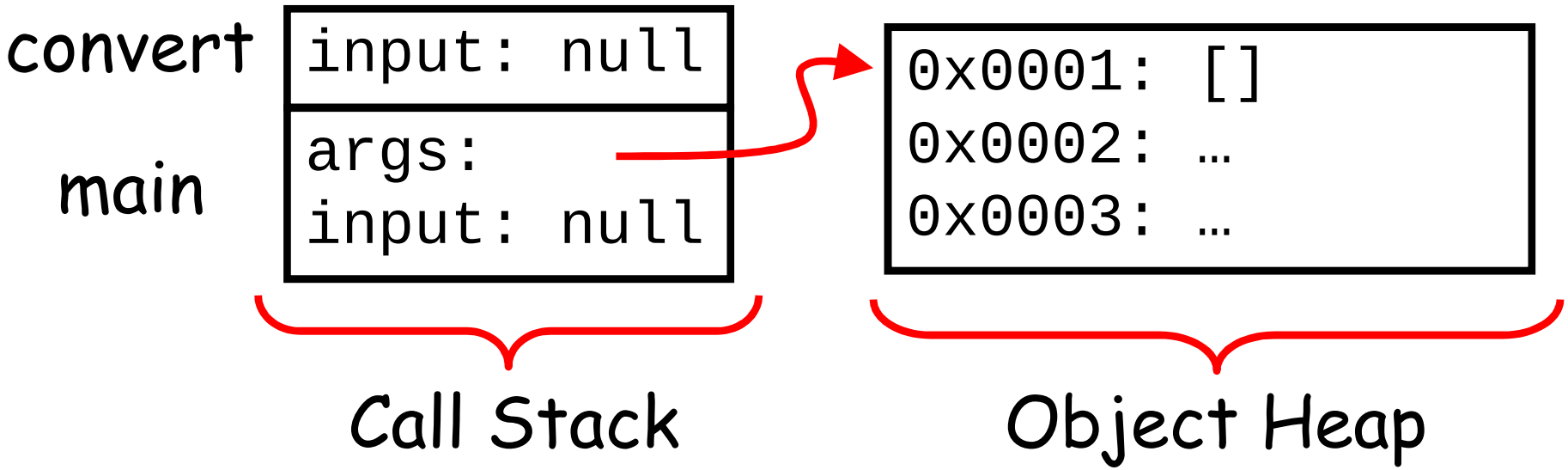
```
class ProblemCase {
 /**
  * @param input – should not be null
  */
 public static char[] convert(String input) {
  char[] cs = new char[input.length()];

  for(int i=0;i!=input.length();++i) {
   cs[i] = input.charAt(i);
  }
  return cs;
 }

 public static void main(String[] args) {
  String input = null;
  if(args.length > 0) { input = args[0]; }
  char[] bs = convert(input);
  …
}}
```



Bad Program States

- What is program state?
  - **Call Stack** for each method, and **Object Heap**
  - Call Stack includes values of all local variables
  - Object Heap includes values of all objects

- **Bad state** for previous example:



Debugging Principles

- Basic approach to debugging:
  - **Observe:** notice failure occurring
  - **Reproduce:** identify input(s) consistently resulting in failure
  - **Focus:** follow propagation trail
  - **Isolate**: identify defect
  - **Record:** add corresponding test case
  - **Correct:** fix the problem!

```
1. class TrafficLights {
2.  private boolean[] lights = {true,false,false};
3.
4.  public TrafficLights next() {
5.   TrafficLights n = new TrafficLights();
6.   n.lights = lights;
7.   if(n.lights[0]) {
8.    n.lights[0]=false; n.lights[1]=true; n.lights[2]=false;
9.   } else if(n.lights[2]) {
10.   n.lights[0]=true; n.lights[1]=false; n.lights[2]=false;
11.  }
12.  return n;
13. }
14.
15. public void print() {
16.  if(lights[0]) { System.out.println(“RED”); }
17.  if(lights[1]) { System.out.println(“AMBER”); }
18.  if(lights[2]) { System.out.println(“GREEN”); }
19. }
20. public static void main(String[] args) {
21. TrafficLights r = new TrafficLights();
22. TrafficLights a = r.next();
23. TrafficLights g = a.next();
24. r.print(); a.print(); g.print();
25. }}
```

Observing and Reproducing

- Observing a failure
  - Need to know what is right and wrong!
  - E.g. **NullPointerException** generally wrong …
  - Good test cases increase chance of observation

- Reproduction
  - Does a given input always cause error?

```
public static void main(String[] args) {
 if(Calendar.getInstance().get(HOUR_OF_DAY) == 13) {
  // defect is in here
  …
 }
 // code continues here
 …
}
```

Finde + Isolate

- Focus on Defect
  - Can be long and laborious task
  - Strategies:
    - Determine smallest input that causes the bug
    - Print debug information and/or use debugger
    - Form hypotheses and eliminate one by one.
  - This is an **art form** !

- Isolate Problem
  - After zeroed in on **defect**, identify **problem**
  - Who is at fault?
    - E.g. wrong method parameters, poor documentation, incorrect algorithm, or something else

Recording + Correcting

- Recording
  - Add appropriate test case to test suite
  - Then, can easily spot same or similar defect
  - Prevents cycle of fixing bugs, then reintroducing them, over and over
- Correcting
  - Sometimes easy, sometimes hard
  - E.g. typo => quick fix
  - But, design limitation => more difficult!

Don’t forget …

Labs start this week!