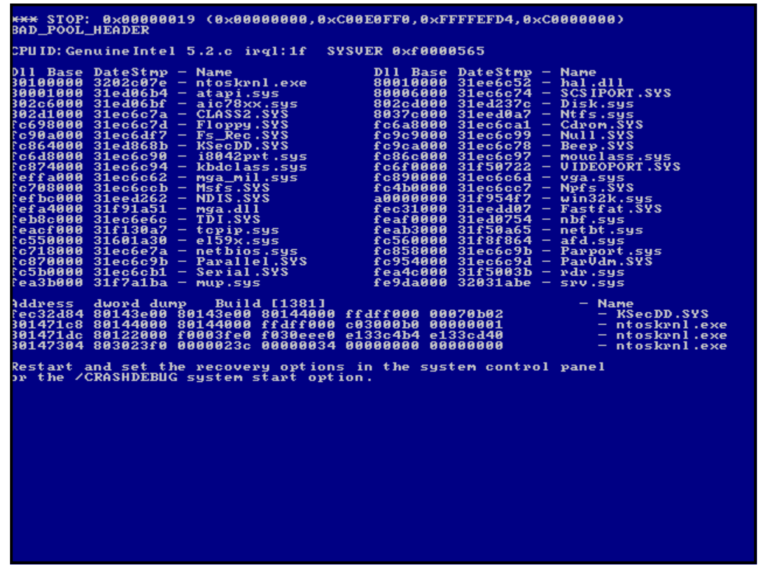
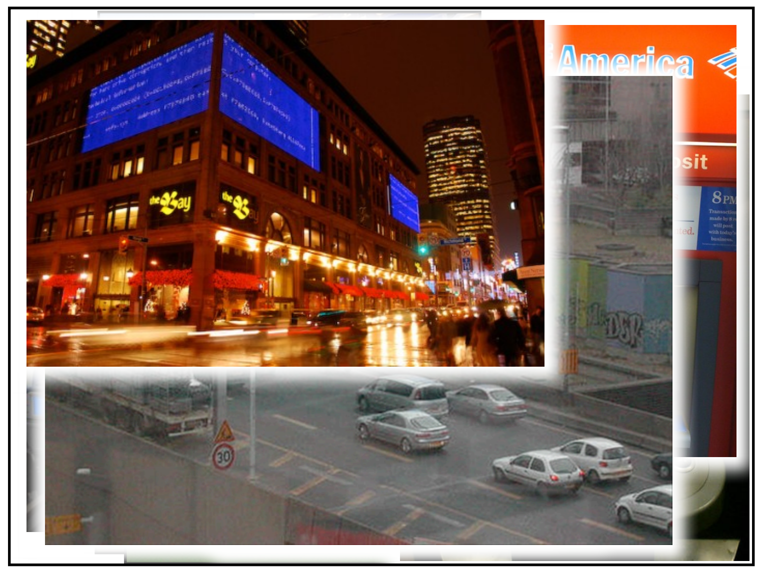
CS 326
Testing

Stephen Freund

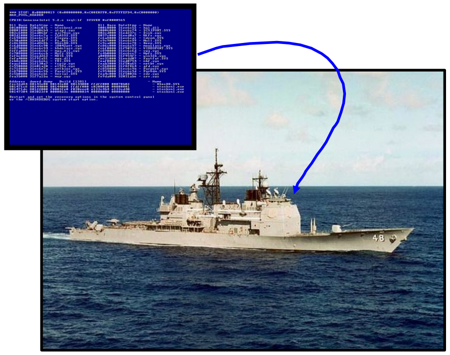
1



The Blue Screen of Death



USS Yorktown



- Smart Ship
  - 27 PCs
  - Windows NT 4.0
- September 21, 1997:
  - data entry error caused a "Divide-By-0" error
  - entire system failed
  - ship dead in the water for over 2 hours

[Wired 1997]

Mars Climate Orbiter

Purpose: Collect data. Relay signals from Mars Polar Lander ($165M)

Failure: Smashed into Mars (1999)

Bug: Failed to convert English to metric units

Mars Polar Lander

Purpose: Lander to study the Mars climate ($120M)

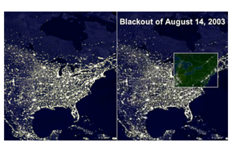
Failure: Smashed into Mars (2000)

Bug: Spurious signals from sensors caused premature engine shutoff

North East Power Failure

Failure: Power grid failed across much of the North East. $6B losses (2001)

Bug: Timing bug in alarm system in Ohio power plant



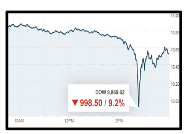
Online Trading Software

well, not really...

Purpose: automatic high-frequency trading

Failure: DOW drops 9.2%, equity markets collapse (2010)

Bug: Bad modeling, and no fail-stops to prevent flooding market with sell orders

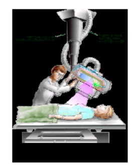


Therac25 Radiation Therapy

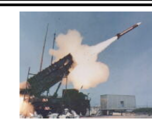
Purpose: Computer-controlled radiation therapy machine

Failure: gave fatal radiation doses to 2 cancer patients (1986)

Bug: timing bug



Patriot Missile



Purpose: Intercept incoming missiles

Failure: missed SCUD missile that killed 28 US soldiers (1991)

Bug: incorrect calculation of distance to target

USS Vincennes

Failure: Shot down an Airbus jet that was mistaken for a F-14. 290 people died. (1988)

Bug: tracking software displayed cryptic and misleading output

Heartbleed SSL Attack

Purpose: OpenSSL is widely-used cryptographic library.

Failure: Library could leak secret information, including keys. (2014)

Bug: Buffer overrun

More Examples

- Mariner I space probe (1962)
- Microsoft Zune New Year's Eve crash (2008)
- iPhone alarm (2011)
- Denver Airport baggage-handling system (1994)
- Air-Traffic Control System in LA Airport (2004)
- AT&T network outage (1990)
- Northeast blackout (2003)
- USS Yorktown Incapacitated (1997)
- Intel Pentium floating point divide (1993)
- Excel: 65,535 displays as 100,000 (2007)
- Prius brakes and engine stalling (2005)
- Soviet gas pipeline (1982)
- Study linking national debt to slow growth (2010)
- Iowa Democratic Caucuses (2020)
- Boeing Starliner Craft (2020)

Software Bugs Cost Money

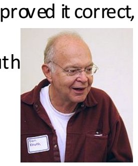
- 2013 Cambridge University study: Software bugs cost global economy $312 Billion per year
  - http://www.prweb.com/releases/2013/1/prweb10298185.htm
- 2012 High-Frequency Trading Error: $440 million loss by Knight Capital Group in 30 minutes
- 2017 Ethereum bug: $300M in crypto-currency
- 2003 NE power blackout: $6 Billon loss

Quality Software

- **External**
  - Correctness — Does it do what it supposed to do?
  - Reliability — Does it do it accurately all the time?
  - Efficiency — Does it do without excessive resources?
  - Integrity — Is it secure?
- **Internal**
  - Portability — Can I use it under different conditions?
  - Maintainability — Can I fix it?
  - Flexibility — Can I change it or extend it or reuse it?
- **Quality Assurance (QA)**
  - Process of uncovering problems and improving software quality
  - Testing is a major part of QA

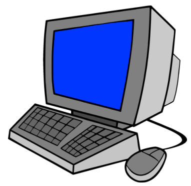
Software Quality Assurance (QA)

No silver bullet:
"Beware of bugs in the above code; I have only proved it correct, not tried it."
-Donald Knuth



"Program testing can be used to show the presence of bugs, but never to show their absence!"
Edsgar Dijkstra

A Bug's Life



Defect: Mistake Committed By Human → **Error:** Incorrect Computation → **Failure:** Visible Error: Program Violates Specification

- **Testing:** Systematically trigger failures.
- **Debugging:** Map failure back to defect.

Design Space for Tests

- **Unit testing** versus **system/integration testing**
- **Black-box testing** versus **clear-box testing**
- **Specification testing** versus **implementation testing**

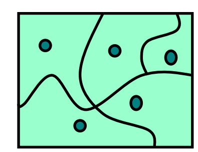
What's the Big Deal?

```
/// -Returns: approximation to square root of x, or
///           nil if x < 0
public func sqrt(x: Double) -> Double?


/// **Requires**: 0 <= x,y,z <= 10,000
///
/// -Returns: f(x,y,z) for some complicated f
public func compute(x:Int, y:Int, z:Int) -> Int
```

Partition the Input Space

- Ideal test suite:
  - Identify sets with same behavior
  - Try one input from each set



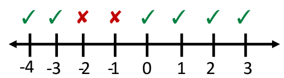
- Two problems:
  - Notion of same behavior is subtle
  - Discovering the sets requires perfect knowledge

Naive Approach: Execution Equivalence

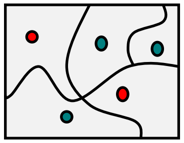
```
/// -Returns:   x < 0      ⇒ returns  -x
///             otherwise  ⇒ returns  x
func abs(x : Int) -> Int {
   if (x < 0) {
     return -x
   } else {
     return x
   }
}
```

Naive Approach: Execution Equivalence

```
/// -Returns:   x < 0      ⇒ returns  -x
///             otherwise  ⇒ returns  x
func abs(x : Int) -> Int {
   if (x < -2) {
     return -x
   } else {
     return x
   }
}
```



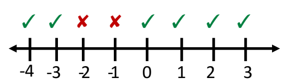
Better: Revealing Subdomains



- A **subdomain** is a subset of possible inputs
- A subdomain is **revealing** for error E if either:
  - Every input in that subdomain triggers error E, or
  - No input in that subdomain triggers error E
- Test only one input from a given subdomain
  - If subdomains cover the entire input space, we are guaranteed to detect the error if it is present
- The trick is to guess these revealing subdomains

Revealing Subdomains (Clear Box)

```
/// -Returns:  x < 0      ⇒ returns -x
///            otherwise ⇒ returns x
func abs(x : Int) -> Int {
   if (x < -2) {
     return -x
   } else {
     return x
   }
}
```



Heuristics for Designing Test Suites

- Good heuristics:
  - Few subdomains
  - ∀ errors in some class of errors E, High probability that some subdomain is revealing for E and triggers E
- Different heuristics target different classes of errors
  - In practice, combine multiple heuristics
  - Really a way to think about and communicate your test choices

Heuristic: Black-Box Testing

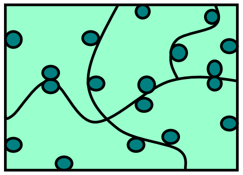
- Heuristic: Explore alternate cases in spec

```
// - Returns:   a > b ⇒ returns  1
//              a < b ⇒ returns -1
//              a = b ⇒ returns  0
func compare(a : Int, b : Int) -> Int
```

```
/// - Returns: the smallest i such
///            that a[i] == value,
///            or nil if no such i exists
func find(a : [Int], value : Int) -> Int?
```

Heuristic: Boundary Testing

- Create tests at the edges of subdomains
  - Off-by-one bugs
  - “Empty” cases (0 elems, nil, …)
  - Overflow errors in arithmetic
  - Largest/Smallest values, 0, …
  - Object aliasing



- Small subdomains at the edges of the “main” subdomains have a high probability of revealing many common errors
  - Also, you might have misdrawn the boundaries

https://en.wikipedia.org/wiki/Signed_number_representations#Two's_complement

Heuristic: Boundary Testing

```
/// - Returns: |x|
public func abs(x : Int) -> Int {…}

class MutableList<T> {
  ...

  /// **Modifies**: self, other
  /// **Effects**:  removes all elements of other and
  ///               appends them in reverse order to
  ///               the end of self
  func append(other: MutableList<T>) {
    while other.count > 0 {
      let element = other.removeLast()
      self.append(element)
    }
  }
}
```

Heuristic: Glass-Box Testing

```
/// primeTable[i] is true if i is prime, for i in
/// 0..<primeTable.count
let primeTable : [Bool] = ...

func isPrime(x : Int) -> Bool {
  if x > primeTable.count {
    for i in 2..<x/2 {
      if x%i == 0 {
        return false
      }
    }
    return true
  } else {
    return primeTable[x]
  }
}
```

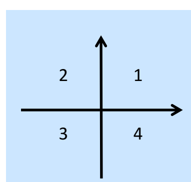
Code Coverage: Statement Coverage

```
func min(a : Int, b : Int) -> Int {
  var result = a
  if a <= b {
    result = a;
  }
  return result
}
```

- `min(1,2)`

Code Coverage: Branch Coverage

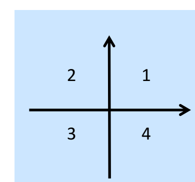
```
func quadrant(x : Int, y : Int) -> Int {
  var ans = 0
  if x >= 0 {
    ans = 1
  } else {
    ans = 2
  }
  if (y < 0) {
    ans = 4
  }
  return ans
}
```



- Test suite: (2,-2) and (-2,2)

Code Coverage: Path Coverage

```
func quadrant(x : Int, y : Int) -> Int {
  var ans = 0
  if x >= 0 {
    ans = 1
  } else {
    ans = 2
  }
  if (y < 0) {
    ans = 4
  }
  return ans
}
```



- Test suite: (2, -2), (2, 2), (-2, 2), and (-2, -2)

Code Coverage: Unbounded Paths…

```
func numPositive(a : [Int]) -> Int {
  var result = 0
  for x in a {
    if x > 0 {
      ans = 1 // should be ans += 1
    }
  }
  return ans
}
```

- {0,0} and {1}?
- {0,1,0}?

```
func numPositive(a : [Int]) -> Int {
  return a.filter( { $0 > 0 } ).count
}
```

Code Coverage: There Are Limits

```
func sumThree(x: Int, y: Int, z: Int) -> Int {
  return x + y
}
```

Pragmatics: Regression Testing

- Whenever you find a bug:
  - Record the input eliciting the bug and the correct output
  - Add these to the test suite
  - Verify that the test suite fails
  - Fix the bug
  - Verify the fix
- Why?
  - Ensures that your fix solves the problem
  - Helps to populate test suite with good tests
  - Protects against reversions that reintroduce bug

Closing Thoughts on Testing

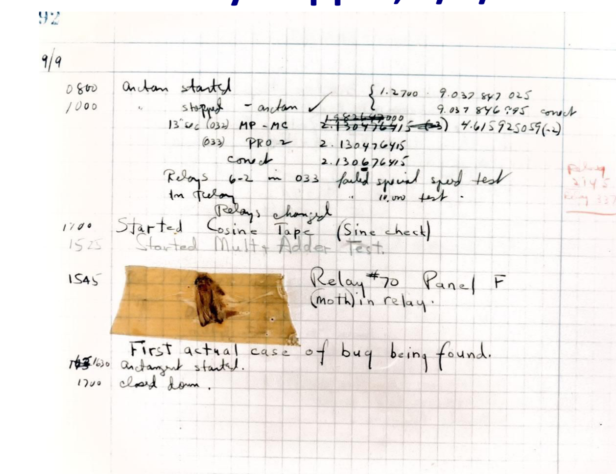
- From Pragmatic Programmer (Read It!):
  - Design To Test.
  - Test Early. Test Often. Test Automatically.
- Quality over Quantity
  - Good tests are hard to write.
  - This will take thinking and time.
- **Every debugging session should end with at least one new test in your repo.**

CS 326
Debugging and Avoiding Failure

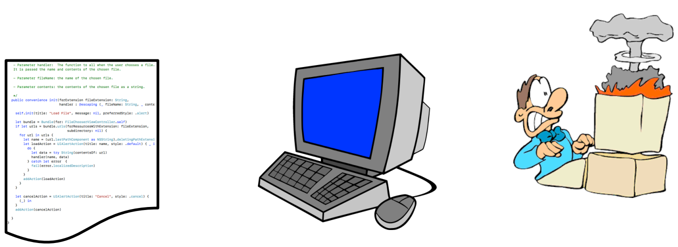
Stephen Freund

33

Grace Murray Hopper, 9/9/47



A Bug's Life



Defect: Mistake Committed By Human → **Error:** Incorrect Computation → **Failure:** Visible Error: Program Violates Specification

← **Debugging**

How To Avoid Failure

1. Design and Verification
   – Ensure there are no defects
2. Testing and Validation
   – Uncover failures
3. Defensive Programming
4. Debugging: you never want to reach this point...

- Testing ≠ Debugging
  – test: trigger failure
  – debug: pinpoint defect (or spec problem)

First Defense: Impossible by Design

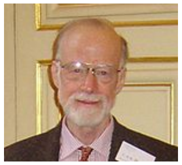
- In the language
  - Swift: no type mismatches, memory overwrite bugs
- In the protocols/libraries/modules
  - TCP/IP guarantees data is not reordered
  - Java BigInteger guarantees there is no overflow
- In self-imposed conventions
  - If-let's to avoid null pointer errors, no rep expsure
  - Immutable structures guarantee behavioral equality
  - Observer methods have no side effects
  - **You must maintain discipline**

Second Defense: Correctness

- Get things right the first time
  - Think before you code
  - Easy-to-find defects implies hard-to-find defects
- Key techniques:
  - Clear and complete specs
  - Well-designed modularity with no rep exposure
  - Testing early and often with clear goals
  - ...
  - **Simplicity!**

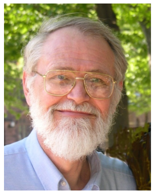
Strive for Simplicity

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.



Sir Anthony Hoare

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.



Brian Kernighan

Third Defense: Immediate Visibility

- If we can't prevent errors, try to localize them:
  - Assertions
  - Unit testing
  - Regression testing
- If we can localize problems to a single method or small module, life is much better.

Run-Time Assertions

- Fail Fast!
- When
  - Preconditions
  - Postconditions
  - Rep Invariants

Run-Time Assertions

- Check: Preconditions, Postconditions, Rep Invariants, potential "hidden errors"
- Example

```
/// **Requires**: x ≥ 0
///
/// - Returns: approximation to square root of x
public func sqrt(_ x : Double) -> Double {
  assert(x >= 0.0, "negative parameter to sqrt")
  let result = … compute result …
  assert(abs(result*result – x) < .0001, "sqrt failed")
  return result
}
```

Hiding an Error

```
// k must be present in a
var i = 0
while (true) {
  if a[i] == k {
    break
  }
  i += 1
}
```

Hiding an Error

```
// k must be present in a
var i = 0
while (i < a.count) {
  if a[i] == k {
    break
  }
  i += 1
}
```

Hiding an Error

```
// k must be present in a
var i = 0
while (i < a.count) {
  if a[i] == k {
    break
  }
  i += 1
}
assert(i != a.count, "key not found")
```

Run-Time Assertions

- Don't clutter code with useless assertions:

```
let x = y + 1
assert(x == y + 1)
```

- Don't perform side effects:

```
assert(list.remove(x))  // won't happen if disabled

// Better:
let found = list.remove(x)
assert(found != nil)
```

- Most assertions better left enabled, even in production

Expensive checkRep() calls

- Eg: `checkRep()` on huge binary search tree
- Best approach (not great):

```
class ADT {

  // set debug to false to disable checkRep tests
  static private let debug = true

  private func checkRep() {
    if (debug) { ... }
  }
}
```

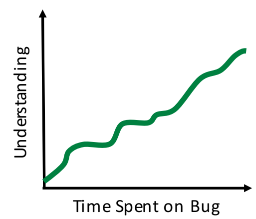
- Also separate expensive tests into different methods to selectively turn only those off

Applying Defenses to CS 326?

- Simplicity of Design!
  - sophisticated vs. complicated
  - If code is hard to write, it is hard to understand
- Which MVC part is easiest to test?
  - Model? UIView? UIViewController?
- Small self-contained abstractions help
  - eg: DotPuzzle, ModelToViewCoordinates
- When to start thinking about tests?
- Time spent writing tests vs. writing code?

Last Line of Defense: Debugging

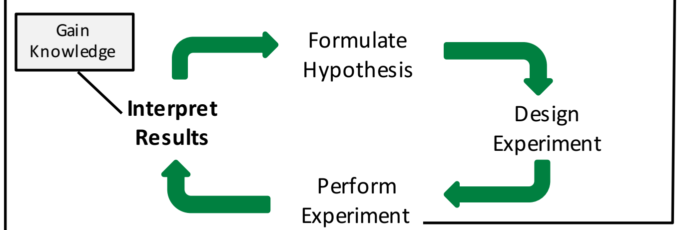
- Clarify symptom
  - Simplify input
  - Find smallest “minimal” test that produces failure
- Gain **knowledge and understanding** of cause
- Fix
- Rerun all tests, old and new
- Reflect on process



Debugging

Revert any changes you made to code/data after experiment

- Be **systematic**
- Keep record of everything you do
- Question assumptions
- Follow iterative scientific method:



Example

```
class String {

  // Returns true iff there exist A, B where
  // self = A : self : B.
  func contains(other: String) -> Bool {...}
}
```

- 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."

Reducing Input Size

- **Absolute Size.** 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"
  - ✔ "very happy"

- Cannot find "ab" within "aab"

Reducing Input Size

- **Relative Size.** Find "very happy" within:
  - ✗ "I am very very happy to see you all."
  - ✔ "I am very happy to see you all."
- General Simplification Rules
  - Simplest may not be related to initial inputs
  - Binary search
  - Input could be sequence of user steps, etc.
    - same rules apply

Localizing A Defect

- Take advantage of modularity
  - Start with everything, take away pieces until failure goes away
  - Start with nothing, add pieces back in until failure appears
- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results
  - Verify pre/post conditions at module boundaries
- Employ binary search
- **Become proficient with available tools**

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;
  }
}
```

no problem yet (at `public Point apply(Matrix current) {`)

problem exists (at `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;
  }
}
```

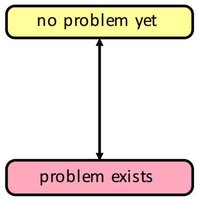
no problem yet (at `public Point apply(Matrix current) {`)

no problem yet (at `applyThreshold(motion,motion,10);`)

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;
  }
}
```



Logging Events

- Log events during execution
  - print, NSLog, ...
- Logs help reconstruct the past
  - Particularly on failing runs
  - And/or compare failing and non-failing runs
- Log may be all you know about a customer's environment
  - Needs to tell you enough to reproduce the failure

After You Fixed Bug: Reflection

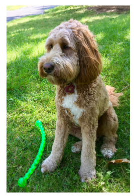
- Debugging is a skill acquired over time
- Reflect on your debugging experience
  - what was the symptom?
  - what was the ultimate cause?
  - was your debugging process effective?
  - how could you have avoided defect? found it sooner?
    - Unit Test? assertion? checkRep()? Better Design? Better Spec? Better Communication? Reading the Docs?
- **Learn from experience**
  - Steve H. garage height story...

Detecting Bugs in the Real World

- Real Systems
  - Collection of modules, written by multiple people
  - Complex input, output
  - Many external interactions
  - Non-deterministic "Heisenbugs"
- "Heisenbugs"
  - Infrequent failure
  - Instrumentation eliminates the failure
- Defects cross abstraction barriers
- Large time lag from defect to failure
- Limited debugging/logging capabilities

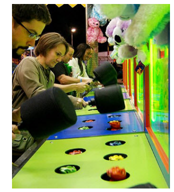
Closing Thoughts on Debugging

- Designing for failure pays off many fold.
- Assume code has bugs. Prove yourself wrong.
- Be pleasantly surprised when code passes tests.
- When the going gets tough:
  1. Make sure it's a bug – check spec
  2. Rule out simple problems (typos, parameter order, ...)
  3. Reconsider assumptions
  4. Take Wally for walk
  5. Talk to friend, rubber ducky
  6. Start documenting system
  7. Go to bed



Know Yourself

- Don't let yourself reach this point:



Whack-a-Mole Debugging

Monkeys-at-Keyboards Time

- Check in with yourself:
  – Are you making progress on understanding?
  – Are you getting frustrated?
  – Reflection is important