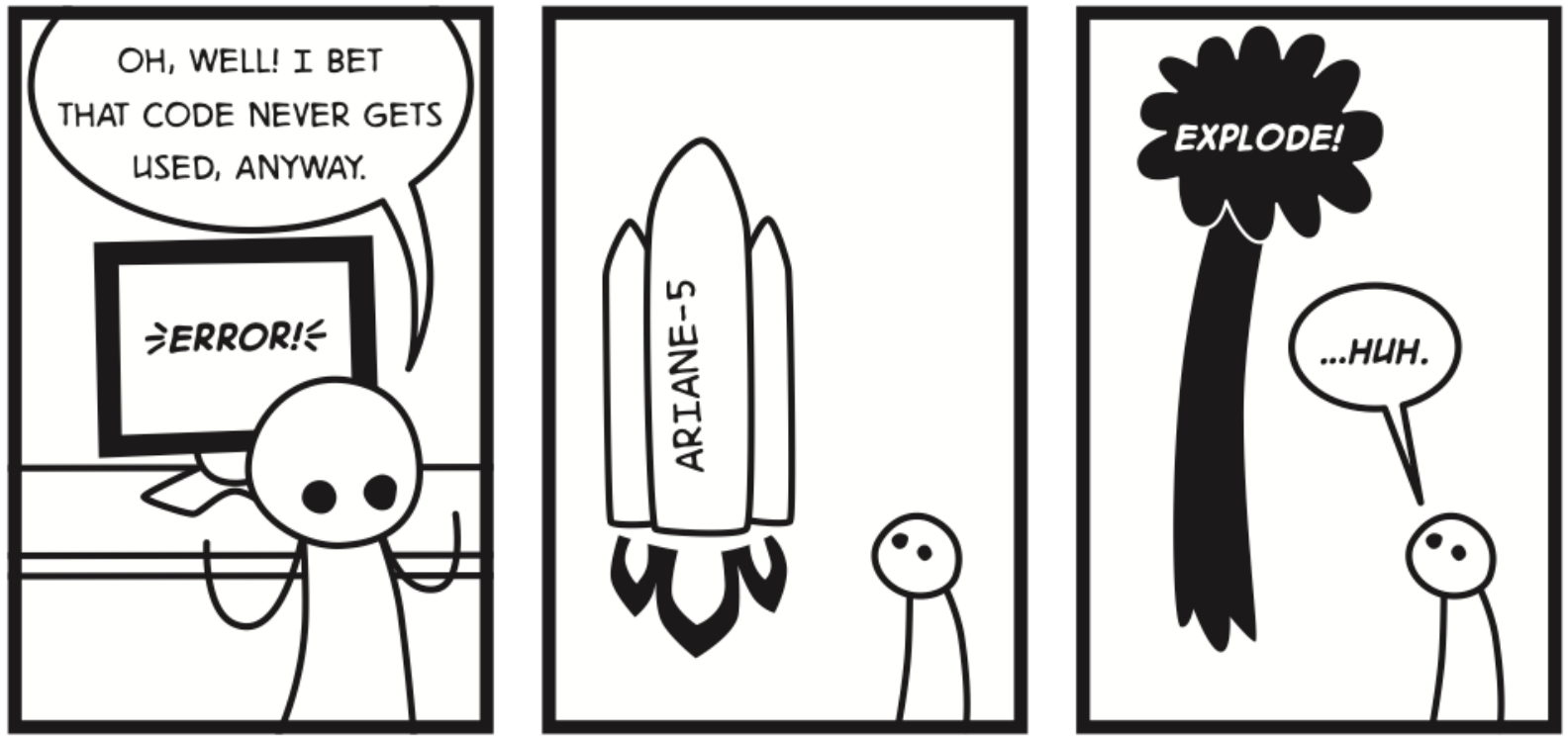
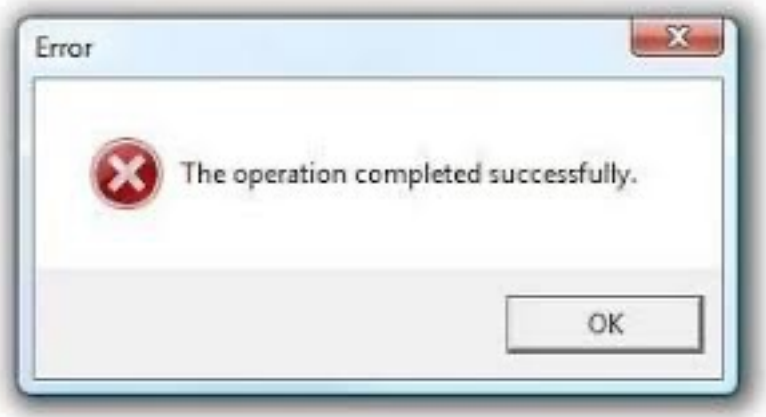

OH, WELL! I BET THAT CODE NEVER GETS USED, ANYWAY.
ERROR!
ARIANE-5
EXPLODE!
...HUH.

Common Coding Mistakes

CSCI 2134: Software Development



Agenda

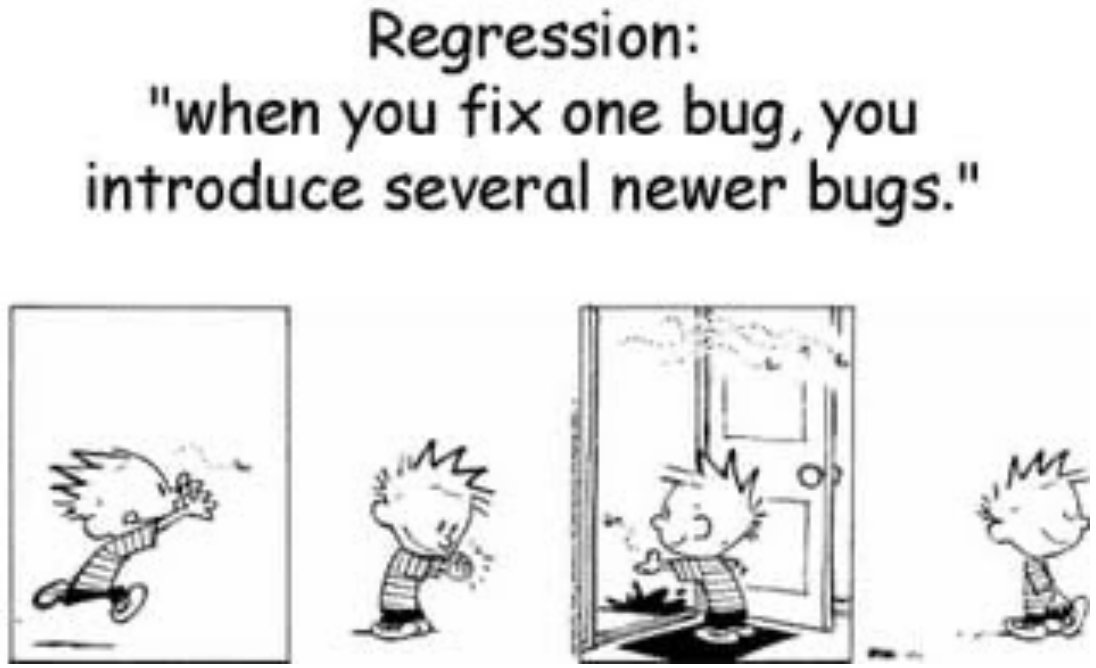
- Lecture Contents
  - Motivation
  - Types of Common Errors
  - Bad code
  - Bad assumptions / Poor robustness
  - Logic errors
  - Faulty processes
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter N/A
  - Next Lecture: Chapter 8

Fixing Bugs: Before the fix (but after debugging)

- **Step 0**: Understand what needs to be fixed
  - Fix the problem (root cause), not the symptom
  - Before making a fix
    - Understand the problem
    - Understand the program
- **Step 1**: Before performing the fix
  - Confirm the bug
  - Relax (Pause)
  - Ensure the current version is committed

Fixing Bugs: Implementation

- **Step 2**: Implement the fix
  - Fix the problem (root cause), not the symptom
  - Change the code only when necessary
  - Make one change at a time
  - **Check your fix (rerun all regression tests)**
- **Step 3**: After the fix
  - Add a unit test that exposes the defect
  - Look for similar defects



Lecture 10

What is the purpose of this ring?

We are obligated to learn from our mistakes.

I ..... in the presence of these my betters and my equals in my Calling, bind myself upon my Honour and Cold Iron, that, of the best of my knowledge and power, I will not henceforward suffer or pass, or be privy to the passing of, Bad Workmanship or Faulty Material in aught that concerns my works before mankind as an Engineer, or in my dealings with my own Soul before my Maker.

MY TIME I will not refuse; my Thought I will not grudge; my Care I will not deny towards the honour, use, stability and perfection of any works to which I may be called to set my hand.

MY FAIR WAGES for that work I will openly take. My Reputation in my Calling I will honourably guard; but I will in no way go about to compass or wrest judgement or

We make plenty of mistakes

Therac 25

Some cost lives (reused bad code)

Ariane 5 first launch

Some cost money (type conversion)

Boeing 737 MAX

Some cost both
(single sensor failure and bad interface)

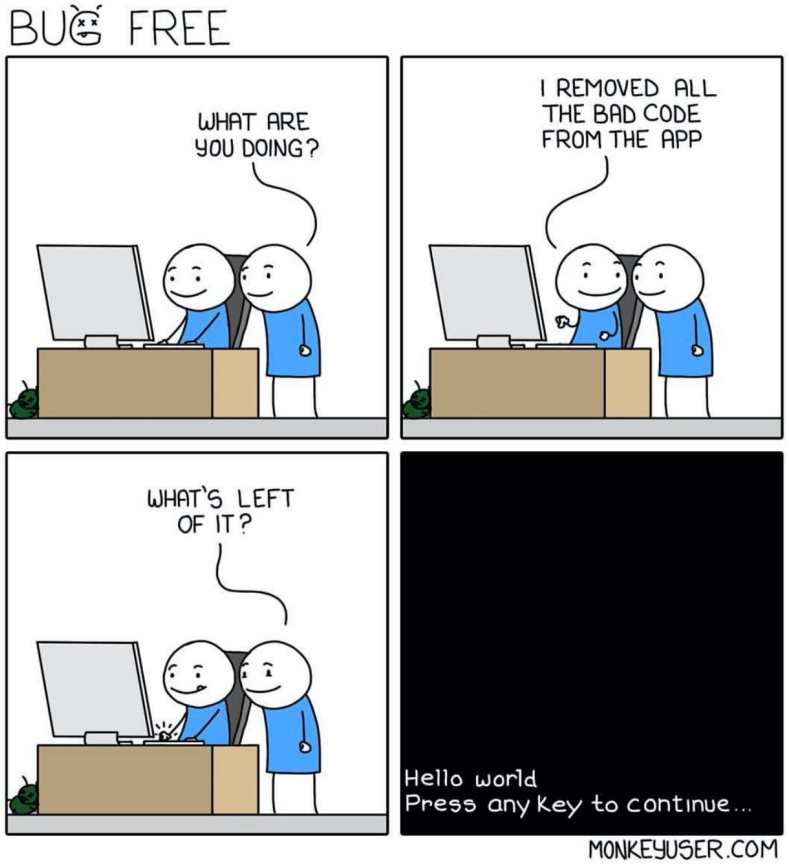
Common Mistakes

- Generic mistakes
  - Bad code (easily avoidable)
  - Bad assumptions / Poor robustness (Somewhat avoidable)
  - Logic errors (Hard to avoid)
  - Faulty processes (Mostly avoidable)
- Domain specific errors
  - Security errors
  - User interface errors
  - Concurrent programming errors
  - Etc.

Programmers can relate.

Bad code

- **Definition**: Bad code contains easily detectable and correctable faults that could lead to other mistakes or bugs
- **Examples:**
  - Code the generates warnings
  - Code that does not meet style guidelines
  - Code that is not documented



Code Should **Never** Generate Warnings

- **Idea**: A compiler warning means that while the code is technically correct, it may result in incorrect behaviour (crash)
- **Rule**: Your code should compile with NO warnings
- Warnings were developed when certain coding patterns were recognized to be error-prone

  Examples:
  - Uninitialized variables
  - Mismatched types
  - Improper casting
  - Etc.

Example 1: Assignment, not equality

- The code:

```
if (a = b) {
    return 1;
}
```

- The warning:

  warnings.c:2:9: warning: using the result of an assignment as a condition

- Why is this a problem?

  This expression will evaluate to false if **a == 0 and b == 0**.

- How do we fix it?

  Replace the **=** with **==**

- The solution:

```
if (a == b) {
    return 1;
}
```

- Notes:
  - Common problem in C/C++
  - This is an error in Java

Example 2: Not for the uninitiated

- The code:

```
int a;
…
if (b == c) {
    a = b + c;
}
return a;
```

- The warning:

  warnings.c:11:7: warning: variable 'a' is used uninitialized whenever 'if' condition is false

- Why is this a problem?

  The value of **a** can be anything (garbage)

- How do we fix it?

  Initialize **a** to 0 or some known value

- The solution:

```
int a = 0; // or some min value
…
if (b == c) {
    a = b + c;
}
return a;
```

Observations on Warnings

- Depending on the language being used, the compiler may generate many or few warnings.
  - C and C++ languages allow for many warning-worthy situations
  - Java allows for very few

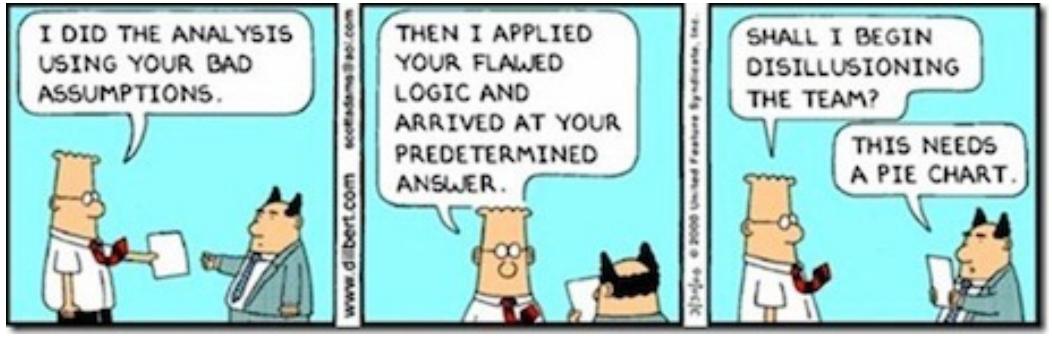
    Warnings in other languages are considered errors in Java.
- Two common warnings in Java:

  `Note: ??????.java uses or overrides a deprecated API.`
  - There is probably a good reason why the API is deprecated.
  - Solution: Don't use deprecated APIs unless you really have a good reason.

  Note: ??????.java uses unchecked or unsafe operations.
  - This occurs when you are using generics and the compiler cannot check the type-safety of an assignment.
  - This is one of the few times that you may not be able to do anything about the warning.

Bad Assumptions and Poor Robustness

- When we write code we make assumptions about
  - What is being passed to a method
  - What a method returns
  - What the input is
  - What the runtime environment provides



- Bad assumptions result in poor robustness
  - Programs will crash on unexpected input
  - Error conditions will be handled in unpredictable ways
  - Use of the code is more restricted
  - More boundary conditions need to be considered, tested, and debugged

Example 3: Arrrr

- The code:

```
int max(int [] arr)) {
  int maxValue = arr[0];
  for (int i = 1; i < arr.length; i++) {
    if (arr[i] > maxValue) {
      maxValue = arr[i];
    }
  }
  return maxValue;
}
```

- What is the problem?
  - Programmer assumes array is not null
  - Programmer assumes array is not length 0
- How do we fix it?
  - Use an assertion or initialized `maxValue` to a minimum value

- The solution:

```
int max(int [] arr)) {
  int maxValue = Integer.MIN_VALUE;
  if ((arr != null) && (arr.length > 0)) {
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > maxValue) {
        maxValue = arr[i];
      }
    }
  }
  return maxValue;
}
```

Example 4: Return it

- The code:

```
String [] arr = list.toArray();

if (arr[0].equals(arr[1])) {
  return true;
}
```

- What is the problem?
  Code assumes that array returned has at least two elements
- How do we fix it?
  Check the length

- The solution:

```
String [] arr = list.toArray();

if (arr.length > 1) {
  if (arr[0].equals(arr[1])) {
    return true;
  }
}
```

Example 5: To err is human

- The code:

```
Scanner scanner = new Scanner(System.in);

System.out.print(”Enter mailbox #:”);
int mailboxId = scanner.nextInt();
Mailbox mailbox = list.get(mailboxId);
...
```

- What is the problem?
  - Programmer assumes user will enter an integer
  - Programmer assumes user will enter correct integer
- How do we fix it?
  - Use an exception or read in a string and check if it is an integer
  - Validate input

- The solution:

```
Scanner scanner = new Scanner(System.in);

int mailboxId = -1;
while (mailboxId < 0) {
  try {
    System.out.print(”Enter mailbox #:”);
    int id = scanner.nextInt();
    if (id < list.size()) {
      mailboxId = id;
    }
  } catch (Exception …) {
    …
  }
}

Mailbox mailbox = list.get(mailboxId);
...
```

Example 6: Allocate this!

- The code:

```
int [][][] data =
  new int[1000][1000][1000];
...
```

- What is the problem?
  - Programmer assumes the program can allocate 4GB of memory
- How do we fix it?
  - Avoid hard-coding resource requirements
  - Use exceptions to catch conditions when resources are exceeded

- The solution:

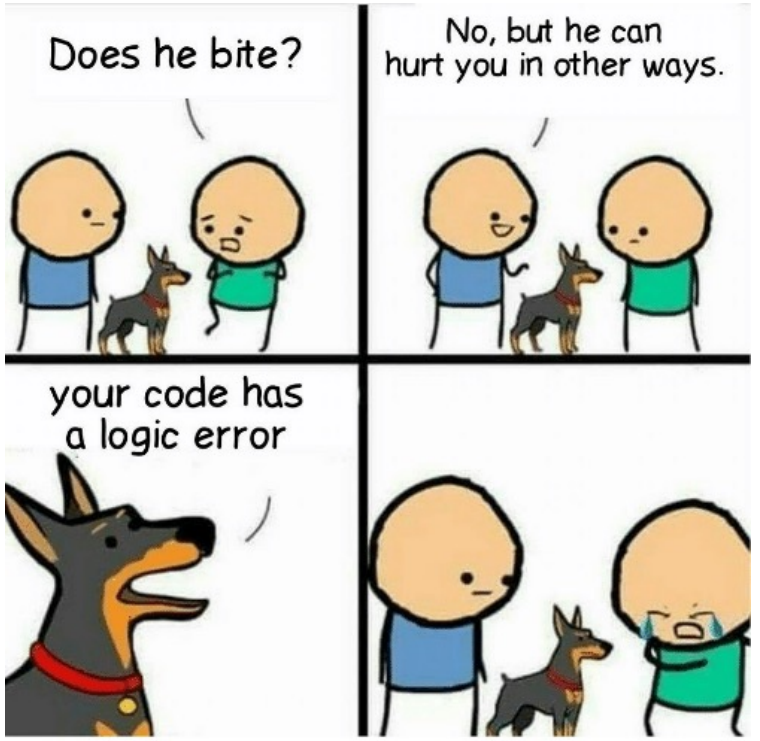
```
int [][][] data = null;

try {
  data =
     new int[1000][1000][1000];
} catch (OutOfMemoryError e) {
  ...
}
...
```

To Avoid Bad Assumptions

- Spell out the assumptions in the method specification
- Do not make assumptions beyond the specification
- Use assertion statements to test all assumptions (next lecture)
- Understand the boundary conditions before
  - Using a method
  - Implementing a method
- Understand the specifications for a piece of code before using it

Logic Errors

- Logic errors are one of the most common causes of bugs!
- Common logic errors include:
  - Off-by-one (O-B-1, sometime called Obi Wan errors)
  - Incorrect comparator
  - Multi-clause condition
  - Incorrect increment in a loop
  - Forgotten conditions
  - Conditions in incorrect order
  - Null variables
  - Reusing a variable without initializing it
- In some cases, these are caused by typos, but in many cases it's a deeper error



Last minute checks before project deadline 😂

Example 7: Poor Comparison

- The code:

```
if (percentage > 80) {
  grade = “A-”;
}
```

- What is the problem?

  Incorrect comparator, a mark of 80 will not get an A-.

- How do we fix it?

  The relational operator should be >=

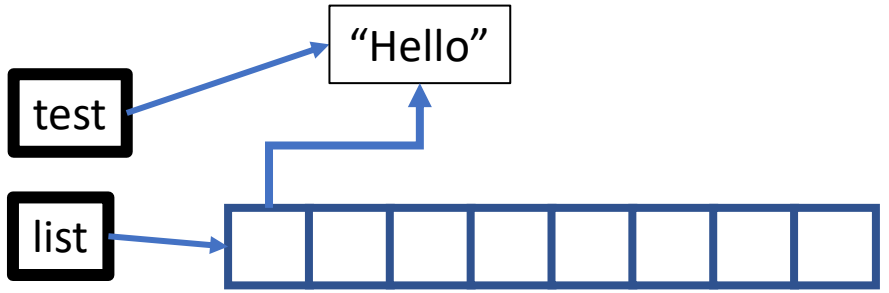
- How do we catch them?

  Unit tests (This is a great example of a boundary condition!)

- The solution:

```
if (percentage >= 80) {
  grade = “A-”;
}
```

Example 8: Incomparable



- The code:

```
boolean contains(String s) {
  for (String str : list) {
    if (str == s) {
      return true;
    }
  }
  return false;
}
```

- What is the problem?
  This code is testing if **`key`** and **`s`** refer to the same object.
- How do we fix it?
  Use `str.equals(s)` instead
- How do we catch them?
  Unit tests (but even unit tests can miss this…)

- The solution:

```
boolean contains(String s) {
  for (String str : list) {
    if (str.equals(s)) {
      return true;
    }
  }
  return false;
}
```

- A slightly broken unit test

```
MyList list = new MyList();
String test = "Hello";
list.add(test);
assertTrue(list.contains(test));
```

This will pass, but contains() is broken

Example 9: Obi Wan (Off by One)

- The code:

```
for (int i = 0; i < arr.length; i++) {
  if (arr[i] > arr[i + 1]) {
    int tmp = arr[i];
    arr[i] = arr[i+1];
    arr[i+1] = tmp;
  }
}
```

- What is the problem?
  - This code will crash when `i` becomes `arr.length - 1`
- How do we fix it?
  - The loop condition should be `i < arr.length - 1`
- These kind of errors are very common. How do we catch them?
  - Unit tests

- The solution:

```
for (int i = 0; i < arr.length - 1; i++) {
  if (arr[i] > arr[i + 1]) {
    int tmp = arr[i];
    arr[i] = arr[i+1];
    arr[i+1] = tmp;
  }
}
```

Example 10: Conditional Surrender

- The code:

```
Node reverse(Node n) {
  if ((n.next == null) || (n == null)) {
    return n;
  }

  Node r = reverse(n.next);
  n.next.next = n;
  n.next = null;
  return r;
}
```

- What is the problem?

  This code has the conditions in the wrong order and will crash when the linked list is empty `(n == null)` but not when the list has one node.

- How do we fix it?

  The conditions should be `(n == null) || (n.next == null)`

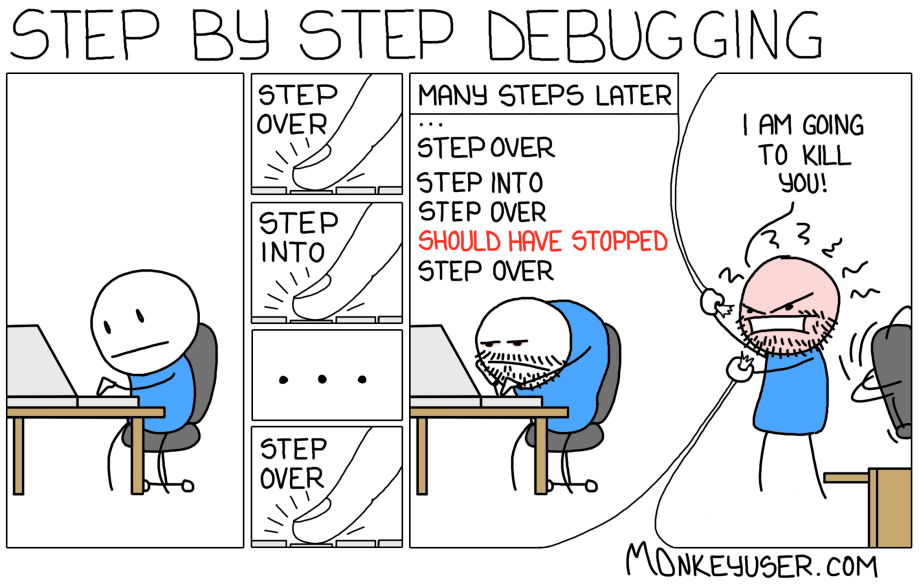
- How do we catch them?

  Unit tests (This is a great example of a boundary condition!)

- The solution:

```
Node reverse(Node n) {
  if ((n == null) || (n.next == null)) {
    return n;
  }

  Node r = reverse(n.next);
  n.next.next = n;
  n.next = null;
  return r;
}
```

Debugging Logic Errors



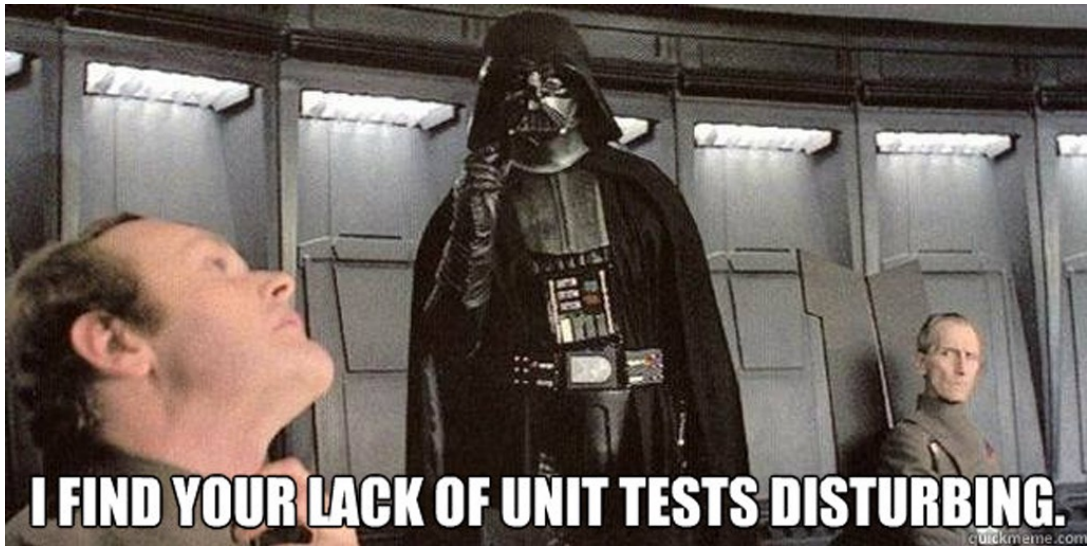
- Use the debugger
  - Step through the code
    - The code “looks right”, but is wrong
    - Until the code is stepped through, it’s easy to miss what’s wrong.
  - Examine the value of variables as you step through the code
- Additional unit tests
  - Logic bugs may exist because someone else (or you) did not test the code properly
  - Especially for boundary cases, be sure to create unit tests
- Use assertions (next lecture)
  - Put in code that enforces assumptions that are made by the code
  - If the assertion fails, then it’s not necessarily a logic error, but rather a bad assumption

More General Approaches

- Technical reviews
  - Bugs are much easier to spot by someone who has not written the code
  - Even an informal review can be very helpful
- Warnings by the IDE
  - Not the same level as compiler, but should be reviewed
- Coding standards and guidelines

Faulty Development Processes

- There are many pitfalls encountered in a rush to implementation
  - Poor requirements : not knowing what you are building
  - Incomplete design : starting an implementation without having most of the design worked out
  - Inconsistent applications of software development guidelines
  - Inconsistent or insufficient version control
  - Infrequent integration
  - Temporary fixes
  - Insufficient testing
  - Insufficient documentation
- What is the root cause of many of these?
  - Time



Poor Requirements

- Poor requirements make it hard (impossible) to get the project right
- Why?
  - Partial knowledge of what is to be built
  - Increased likelihood and scope of change requests (more work)
  - Increased difficulty of finalizing a sound design
  - Increased difficulty of testing (hard to know what is right)
  - Increased likelihood of future conflict with client
- But … getting 100% requirements done is impractical
  - Requirements change as projects evolve
  - Some requirements may be impossible or impractical to implement
  - Expect some change
- Common Practice: (McConnel, CC2, 2004, pg 34)
  - Have 80% of requirements completed before design phase
  - Allocate time to complete requirements in rest of project

Incomplete Design

- Incomplete design creates several challenges
  - Harder to divide programming tasks among a team
  - Harder to integrate components
- Note: The entire design need not be completed and will evolve.
- But, the high-level design needs to be completed in order to proceed.
- What should be in the design?
  - Architecture (later in this course)
  - All (major) components
  - All interactions between components
  - All interfaces/APIs for the components

Poor Implementation Practices

- Poor implementation practices include
  - Inconsistent applications of software development guidelines
  - Inconsistent or insufficient version control
  - Infrequent integration
- These practices will lead to more work later on

What Would You Do?

- **Question**: What should we do if a developer does not follow development guidelines?
  - Option: Block merge permissions
- **Question**: What should we do if a developer does not commit as frequently as they should?
  - Option: Chat with manager?
  - Motivation: Backup of code
- **Question**: What should we do if a developer does not perform sufficiently frequent integrations?
  - Option: Chat with manager?
  - Motivation: Less debugging later

Insufficient Testing and Insufficient Documentation

- Insufficient testing or documentation are a failure to meet software quality objectives
- Insufficient testing leads to
  - More debugging
  - Lower quality product
- Insufficient documentation leads to
  - Poor testability, understandability, and maintainability
- Question: What do we do if a developer does not perform sufficient testing or documentation?
  - Option: **Gating**: Prevent developer to move to next task until they complete the current one to acceptable levels.

Key 

Points

- Common errors include:
  - Bad code (Easily avoidable)
  - Bad assumptions / Poor robustness (Somewhat avoidable)
  - Logic errors (Hard to avoid)
  - Faulty processes (Mostly avoidable)
- Bad code and faulty processes are caused by poor development practices and are for the most part avoidable
- Bad assumptions: can result from poor development practices and from incomplete requirements or design
- Logic errors are unavoidable, and require debugging tools to rectify.

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://i.pinimg.com/originals/72/2c/4b/722c4bef3d62e450fd07d43977beff3f.jpg
- https://pbs.twimg.com/media/Ddx4aENVwAAuuHP.jpg
- https://i.redd.it/0jelwo9pgcv21.jpg
- https://pics.me.me/no-but-he-can-hurt-you-in-other-ways-does-64387921.png
- https://www.google.com/url?sa=i&url=https%3A%2F%2Fmemegenerator.net%2Finstance%2F66662830%2Fyoda-star-wars-incomplete-detailed-design-does-not-a-high-level-design-make&psig=AOvVaw3jUSzYpEIxVu3wRf7U4ZWt&ust=1581042249913000&source=images&cd=vfe&ved=0CA0QjhxqFwoTCMjf4c7vu-cCFQAAAAAdAAAAABAK
- https://play.google.com/?hl=en-GB&tab=i81
- https://pics.me.me/happiness-is-when-your-code-runs-without-error-programmers-can-19548774.png
- https://i.pinimg.com/originals/b2/23/ec/b223ecf6a756dd8053237cc92f75da99.png
- https://i.pinimg.com/originals/3b/da/8f/3bda8f29323262ae6b8d37c8d5776856.jpg

Retrieved October 9, 2020

- https://archive.csfieldguide.org.nz/2.5.0/chapters/software-engineering.html
- https://xkcd.com/2248/