Debugging is **unavoidable** and a major **economical factor**

- Software bugs cost the US economy ca. 60 billion US$/y (2002)
  - In general estimated 0.6% of the GDP of industrial countries
- Ca. 80 percent of software development costs spent on identifying and correcting defects
- Software re-use is increasing and tends to introduce bugs due to changed specifications in new context (Ariane 5)
Motivation

Debugging is **unavoidable** and a major **economical factor**

- Software bugs cost the US economy ca. 60 billion US$/y (2002)
  In general estimated 0.6% of the GDP of industrial countries
- Ca. 80 percent of software development costs spent on identifying and correcting defects
- Software re-use is increasing and tends to introduce bugs due to changed specifications in new context (Ariane 5)

Debugging needs to be **systematic**

- Bug reports may involve **large inputs**
- Programs may have **thousands of memory locations**
- Programs may pass through **millions of states** before failure occurs
Example: memory graph of GCC 2.95.2
### Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to the code by programmer
   - *Not always programmer’s fault:* changing/unforeseen requirements

2. Defect may cause **infection** of the program state during execution
   - *Not all defects cause an infection:* e.g., Pentium bug

3. An infected state **propagates** during execution
   - Infected parts of states may be overwritten, corrected, unused

4. Infection may cause a **failure**: externally observable error
   - May include non-termination

---

Defect — Infection — Propagation — Failure
The Main Steps in Systematic Debugging

Reproduce failure with test input
The Main Steps in Systematic Debugging

Program State

Time

Reduction of failure-inducing problem
The Main Steps in Systematic Debugging

Program State

Time

State known to be healthy
The Main Steps in Systematic Debugging

Program State

Time

State known to be infected
The Main Steps in Systematic Debugging

Program State

Time

State where failure becomes observable
The Main Steps in Systematic Debugging

- Separate healthy from infected states
The Main Steps in Systematic Debugging

- Separate healthy from infected states
- Separate relevant parts from irrelevant ones
Debugging Techniques

The analysis suggests main techniques used in systematic debugging:

- **Bug tracking** — Which start states cause failure?
- **Program control** — Design for Debugging
- **Input simplification** — Reduce state size
- **State observation and watching using debuggers**
- **Tracking causes and effects** — From failure to defect
Debugging Techniques

The analysis suggests main techniques used in systematic debugging:

- **Bug tracking** — Which start states cause failure?
- **Program control** — Design for Debugging
- **Input simplification** — Reduce state size
- **State observation and watching using debuggers**
- **Tracking causes and effects** — From failure to defect

**Common Themes**

- Fighting combinatorial explosion: separate relevant from irrelevant
- Being systematic: avoid repetition, ensure progress, use tools
Bug Tracking Life Cycle

Unconfirmed

Raw problem report, often from end user
Bug Tracking Life Cycle

Unconfirmed → New

Description complete, no duplicate of existing bug
Bug Tracking Life Cycle

Unconfirmed → New → Assigned
Bug Tracking Life Cycle

Unconfirmed ➔ New ➔ Assigned ➔ Resolved

<table>
<thead>
<tr>
<th>Tag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Problem</td>
<td>is fixed</td>
</tr>
<tr>
<td>WontFix</td>
<td>Impossible or undesirable (&quot;feature&quot;) to fix</td>
</tr>
<tr>
<td>WorksForMe</td>
<td>Can’t be reproduced</td>
</tr>
<tr>
<td>Invalid</td>
<td>Not a problem or insufficiently described</td>
</tr>
<tr>
<td>Duplicate</td>
<td>Refers to existing problem</td>
</tr>
</tbody>
</table>
Bug Tracking Life Cycle

Unconfirmed → New → Assigned → Resolved → Verified

Invalid
Duplicate

Only if Fixed
Bug Tracking Life Cycle

Unconfirmed → New → Assigned → Resolved → Verified → Closed

Invalid
Duplicate

Important to avoid cluttering of bug database
Bug Tracking Life Cycle

Unconfirmed → New → Assigned → Resolved → Verified → Closed

Invalid
Duplicate

Reopened

The fix didn’t work after all . . .
Bugzilla’s Bug Lifecycle

Bug is filed by a non-empowered user in a product where the UNCONFIRMED state is enabled.

Bug determined to be present

UNCONFIRMED

Bug determined to be present

CONFIRMED

Developer is working on the bug

Developer stops work on bug

IN_PROGRESS

Fix checked in

QA not satisfied with the solution

RESOLVED

Possible resolutions:
- FIXED
- DUPLICATE
- WONTFIX
- WORKSFORME
- INVALID

QA verifies that the solution works

VERIFIED

Fix turns out to be wrong

Bug is not fixable (e.g., because it is invalid)

Bug is reopened, was never confirmed
Scenario
Assume Firefox crashes while printing a certain URL to file

We need to turn the bug report into an automated test case!
Program Control: From Bug to Test Case

Scenario

Assume Firefox crashes while printing a certain URL to file

We need to turn the bug report into an automated test case!

Automated test case execution essential

- Reproduce the bug reliably (cf. scientific experiment)
- Repeated execution necessary during isolation of defect
- After successful fix, test case becomes part of test suite

Prerequisites for automated execution

1. Program control (without manual interaction)
2. Isolating small program units that contain the bug
Enable automated run of program that may involve user interaction

Example (Sequence of interaction that led to the crash)

1. Launch **Firefox**
2. Open URL location dialogue
3. Type in a location
4. Open Print dialogue
5. Enter printer settings
6. Initiate printing
Alternative Program Interfaces for Testing

User

Presentation Layer

(Common) Functionality Layer

Unit
Unit
Unit
Alternative Program Interfaces for Testing

![Diagram showing layers of a software architecture and interactions between them.]

- User
- Presentation Layer
- (Common) Functionality Layer
- Units
- Automated Test
Automated Testing at Different Layers

Presentation
Scripting languages for capturing & replaying user I/O
- Specific to an OS/Window system/Hardware
- Scripts tend to be brittle

Functionality
Interface scripting languages
1. Implementation-specific scripting languages: VBScript
2. Universal scripting languages with application-specific extension: Python, Perl, Tcl

Unit testing frameworks (as in previous lecture)
JUnit, CPPUnit, VBUnit, ...
Testing Layers: Discussion

The higher the layer, the more difficult becomes automated testing

- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

Test at the unit layer whenever possible! Requires modular design with low coupling

Good design is essential even for testing and debugging!

We concentrate on decoupling rather than specific scripts
The higher the layer, the more difficult becomes automated testing

- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

Test at the unit layer whenever possible!
The higher the layer, the more difficult becomes automated testing

- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

Test at the unit layer whenever possible!

Requires modular design with low coupling

- Good design is essential even for testing and debugging!
- We concentrate on decoupling rather than specific scripts
Circular Dependency Example

- Print-to-file is core functionality
  - calls confirm_loss() to prevent accidental file removal
- Override-if-file-exists question is in UI
  - relies on core functionality to check file existence
Programming to interfaces important even for testability
Isolating Units

Use test interfaces to isolate smallest unit containing the defect

- In the Firefox example, unit for file printing easily identified
- In general, use debugger to trace execution
Scenario
Assume Firefox crashes while printing a loaded URL to file

We need to turn the bug report into an automated test case!

We managed to isolate the relevant program unit, but . . .
Scenario

Assume Firefox crashes while printing a loaded URL to file

We need to turn the bug report into an automated test case!

We managed to isolate the relevant program unit, but ...

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
  <head>
    <title>Mozilla.org</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    ...
    ca 200 lines more
  </body>
</html>
```
Problem Simplification

We need a small test case that fails!
Problem Simplification

We need a **small** test case that fails!

**Divide-and-Conquer**

1. Cut away one half of the test input
2. Check, whether one of the halves still exhibits failure
3. Continue until minimal failing input is obtained
Problem Simplification

We need a **small** test case that fails!

**Divide-and-Conquer**

1. Cut away one half of the test input
2. Check, whether one of the halves still exhibits failure
3. Continue until minimal failing input is obtained

**Problems**

- Tedious: rerun tests manually
- Boring: cut-and-paste, rerun
- What, if none of the halves exhibits a failure?
Automatic Input Simplification

- Automate cut-and-paste and re-running tests
- Increase granularity of chunks when no failure occurs

Example
```java
public static int checkSum(int[] a)
```

is supposed to compute the checksum of an integer array
gives wrong result, whenever `a` contains two identical consecutive numbers, but we don't know that yet
we have a failed test case, eg, from transmission trace:
```
{1,3,5,3,9,17,44,3,6,1,1,0,44,1,44,0}
```
Automatic Input Simplification

- Automate cut-and-paste and re-running tests
- Increase granularity of chunks when no failure occurs

Example

public static int checkSum(int[] a)

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever a contains two identical consecutive numbers, but we don’t know that yet
- we have a failed test case, eg, from transmission trace:

\{1, 3, 5, 3, 9, 17, 44, 3, 6, 1, 1, 0, 44, 1, 44, 0\}
Input Simplification \( (n = \text{number of chunks}) \)

\[
\begin{array}{ccccccccccccc}
1 & 3 & 5 & 3 & 9 & 17 & 44 & 3 & 6 & 1 & 1 & 0 & 44 & 1 & 44 & 0
\end{array}
\]
Input Simplification \((n = \text{number of chunks})\)

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>44</td>
</tr>
</tbody>
</table>

n=2

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Input Simplification (n = number of chunks)**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**n=2**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**n=3**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Input Simplification ($n =$ number of chunks)

- **$n=2$**
  - Incorrect arrangement

- **$n=3$**
  - Incorrect arrangement

- **$n=4$**
  - Incorrect arrangement
Input Simplification (\(n = \text{number of chunks}\))

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>3</th>
<th>9</th>
<th>17</th>
<th>44</th>
<th>3</th>
<th>6</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>44</th>
<th>1</th>
<th>44</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>1</td>
<td>44</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **n=2**

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>3</th>
<th>9</th>
<th>17</th>
<th>44</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>1</td>
<td>44</td>
<td>0</td>
</tr>
</tbody>
</table>

- **n=3**

| 6 | 1 | 1 | 0 |

- **n=4**

| 6 | 1 |

1 0
### Input Simplification ($n =$ number of chunks)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>3</th>
<th>9</th>
<th>17</th>
<th>44</th>
<th>3</th>
<th>6</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>44</th>
<th>1</th>
<th>44</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>n=2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>17</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>1</td>
<td>44</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>n=4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Input Simplification Rules:

- **n=2**:
  - Adjust granularity to input size.
  - The input is divided into 2 chunks: 1, 3, 5, 3, 9, 17, 44, 3 and 6, 1, 1, 0, 44, 1, 44, 0.
  - The first chunk is marked **green** (correct) and the second chunk is marked **red** (incorrect).

- **n=4**:
  - Increase granularity.
  - The input is divided into 4 chunks: 6, 1, 1, 0.
  - The input is marked **red** (incorrect).
  - Increase granularity further.
  - The input is divided into 4 chunks: 6, 1, 1.
  - The input is marked **red** (incorrect).
Input Simplification \((n = \text{number of chunks})\)

<table>
<thead>
<tr>
<th>(n=2)</th>
<th>(n=4) increase granularity</th>
<th>(n=3) adjust granularity to input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 5 3 9 17 44 3 6 1 1 0 44 1 44 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 1 1 0</td>
<td>6 1 1 0</td>
<td>6 1 1 1</td>
</tr>
<tr>
<td>6 1</td>
<td>6 1</td>
<td>6 1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>
Prerequisites

- $\text{test}(c) \in \{\checkmark, \times, ?\}$ runs a test on configuration $c$
- Let $c_\times$ be a failing input configuration with
  - $\text{test}(c_\times) = \times$
  - length $l = |c_\times|$ if $c_\times = \{x_1, \ldots, x_l\}$
  - view at granularity $n \leq l$: $c_\times = c_1 \cup \cdots \cup c_n$, $c_i \neq \emptyset$
  - write $c_i \in_n c$
Simplification Algorithm — Delta Debugging

Prerequisites

- test(c) ∈ {✓, x, ?} runs a test on configuration c
- Let c be a failing input configuration with
  - test(c) = x
  - length l = |c| if c = {x_1, ..., x_l}
  - view at granularity n ≤ l: c = c_1 ∪ ... ∪ c_n, c_i ≠ ∅
  - write c_i ∈_n c

Find minimal failing input: call dd_{Min}(c_0, 2) with test(c_0) = x

dd_{Min}(c, n) =

\[
\begin{cases}
  c & |c| = 1 \\
  dd_{Min}(c - c, \max(n-1, 2)) & c \in_n c \land test(c - c) = x \\
  dd_{Min}(c, \min(2n, |c|)) & n < |c| \\
  c & \text{otherwise}
\end{cases}
\]
Minimal Failure Configuration

- Minimization algorithm is easy to implement
- Realizes input size minimization for failed run
- Implementation:
  - Small program in your favorite PL (Zeller: Python, Java)
  - Eclipse plugin DDINPUT at www.st.cs.uni-sb.de/eclipse/
- Demo: DD.java, Dubbel.java
Minimal Failure Configuration

- Minimization algorithm is easy to implement
- Realizes input size minimization for failed run
- Implementation:
  - Small program in your favorite PL (Zeller: Python, Java)
  - Eclipse plugin DDinput at www.st.cs.uni-sb.de/eclipse/

Consequences of Minimization

- Input small enough for observing, tracking, locating (next topics)
- Minimal input often provides important hint for source of defect

Demo: DD.java, Dubbel.java
Principal Limitations of Input Minimization

- Algorithm computes **minimal** failure-inducting subsequence of the input:
  Taking away any chunk of any length removes the failure
- However, there might be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks
Algorithm computes \textit{minimal} failure-inducting subsequence of the input:

\textbf{Taking away any chunk of any length removes the failure}

- However, there might be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

\textbf{Example (Incompleteness of minimization)}

Failure occurs for integer array when frequency of occurrences of all numbers is even:
Principal Limitations of Input Minimization

- Algorithm computes **minimal** failure-inducing subsequence of the input:
  - Taking away any chunk of any length removes the failure
- However, there might be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

**Example (Incompleteness of minimization)**
Failure occurs for integer array when frequency of occurrences of all numbers is even:
\{1,2,1,2\} fails
Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducing subsequence of the input:
  Taking away any chunk of any length removes the failure
- However, there might be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)

Failure occurs for integer array when frequency of occurrences of all numbers is even:

\{1,2,1,2\} fails
Taking away any chunk of size 1 or 2 passes
Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducting subsequence of the input:
  Taking away any chunk of any length removes the failure
- However, there might be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)
Failure occurs for integer array when frequency of occurrences of all numbers is even:

\{1,2,1,2\} fails
Taking away any chunk of size 1 or 2 passes
\{1,1\} fails, too, and is even smaller
Limitations of Linear Minimization

Minimization algorithm ignores structure of input

Example (.html input configuration)

\[
\text{SELECT NAME="priority"MULTIPLE SIZE=7} \times \\
\begin{itemize}
\item Most substrings are not valid HTML: test result? ("unresolved")
\item There is no point to test beneath granularity of tokens
\end{itemize}
\]

Minimization may require a very large number of steps
Structured Minimization

Linearization of $c_x$:

```
SELECT
    0
    1 2 3
    NAME  MULTIPLE  SIZE
    1.1  3.1
    "priority"  7
```

Input configuration consists of nodes in ABS not characters.
Structured Minimization

Linearization of $c_x$:

```xml
<SELECT NAME="priority" MULTIPLE SIZE=7>
  0
  SELECT
    1 2 3
    NAME MULTIPLE SIZE
    1.1 3.1
    "priority" 7
</SELECT>
```

$c_x = \{0, 1, 1.1, 2, 3, 3.1\}$
Structured Minimization

Linearization of $c_x$:

$\langle \text{SELECT NAME=}'priority'$ MULTIPLE SIZE=7 $\rangle$

$\begin{array}{c|c|c}
\text{NAME} & \text{MULTIPLE} & \text{SIZE} \\
\hline
1.1 & 3.1 & \\
\end{array}$

$c_x = \{0, 1, 1.1, 2, 3, 3.1\}$ infeasible (not a tree) return
Structured Minimization

Linearization of $c_x$:

<select name="priority" multiple size=7>
  <option value="0">0</option>
  <option value="1" selected>1</option>
  <option value="2">2</option>
  <option value="3">3</option>
  <option value="1.1" selected>1.1</option>
  <option value="3.1" selected>3.1</option>
</select>

$c_x = \{0, 1, 1.1, 2, 3, 3.1\}$ Failure occurs, reduce length
Structured Minimization

Linearization of $c_x$:

$<\text{SELECT NAME=}"\text{priority}"\ \text{MULTIPLE SIZE=7}>$

$c_x = \{0, 1, 1.1, 2, 3, 3.1\}$ infeasible (not well-formed HTML) return ?
Structured Minimization

Linearization of $c_x$:

```xml
<SELECT NAME="priority" MULTIPLE SIZE=7>
  <option value="0" selected="selected">0</option>
  <option value="1">1</option>
  <option value="2">2</option>
  <option value="3">3</option>
  <option value="1.1">1.1</option>
  <option value="3.1">3.1</option>
</SELECT>
```

$c_x = \{0, 1, 1.1, 2, 3, 3.1\}$ Failure occurs, can’t be minimized further.
The Bigger Picture

- Minimization of failure-inducing input is instance of delta debugging
- Delta debugging is instance of adaptive testing
Delta Debugging, Adaptive Testing

The Bigger Picture

- Minimization of failure-inducing input is instance of **delta debugging**
- Delta debugging is instance of **adaptive testing**

Definition (Delta Debugging)

Isolating failure causes by narrowing down differences ("δ") between runs

This principle is used in various debugging activities
Delta Debugging, Adaptive Testing

The Bigger Picture

- Minimization of failure-inducing input is instance of delta debugging
- Delta debugging is instance of adaptive testing

Definition (Delta Debugging)
Isolating failure causes by narrowing down differences ("δ") between runs

This principle is used in various debugging activities

Definition (Adaptive Testing)
Test series where each test depends on the outcome of earlier tests
Some Tips

Logging
Log all debugging activities, particularly, test cases and outcomes
Some Tips

Logging
Log all debugging activities, particularly, test cases and outcomes

Add Testing Interfaces
Avoids presentation layer scripts (brittle!) and interaction (tedious!)
Some Tips

Logging
Log all debugging activities, particularly, test cases and outcomes

Add Testing Interfaces
Avoids presentation layer scripts (brittle!) and interaction (tedious!)

Set Time Limit for Quick-and-Dirty Debugging
Use “naive” debugging when bug seems obvious, but 10 mins max!
Some Tips

**Logging**
Log all debugging activities, particularly, test cases and outcomes

**Add Testing Interfaces**
Avoids presentation layer scripts (brittle!) and interaction (tedious!)

**Set Time Limit for Quick-and-Dirty Debugging**
Use “naive” debugging when bug seems obvious, but 10 mins max!

**Do not Test the Wrong Program**
Is the path and filename correct? Did you compile?
What Next?

- Bug tracking
- Program control — Design for Debugging
- Input simplification
What Next?

- Bug tracking
- Program control — Design for Debugging
- Input simplification

- Execution observation
  - With logging
  - Using debuggers
- Tracking causes and effects
|--------------------|------------|--------------------------------------------------------------------------|------------------|