Reproducing Problems Andreas Zeller

20.30 (111)

The First Task

- Once a problem is reported (or exposed by a test), some programmer must fix it.
- The first task is to reproduce the problem.

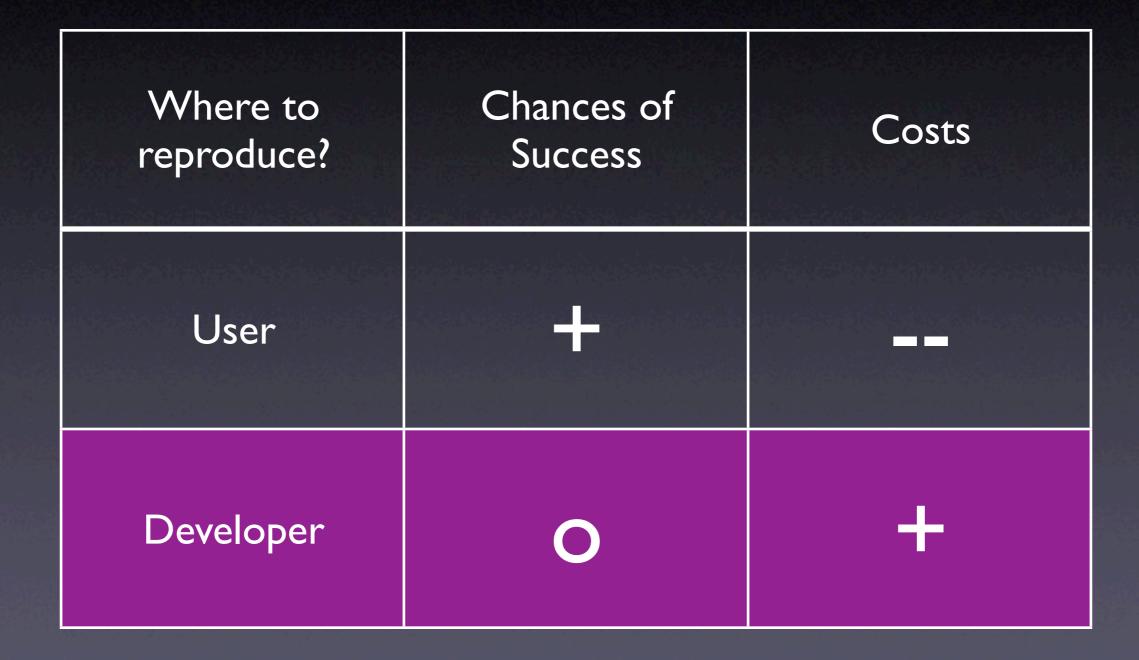
Why reproduce?

- Observing the problem. Without being able to reproduce the problem, one cannot observe it or find any new facts.
- Check for success. How do you know that the problem is actually fixed?

A Tough Problem

- Reproducing is one of the toughest problems in debugging.
- One must
 - recreate the environment in which the problem occurred
 - recreate the problem history the steps that lead to the problem

Reproducing the Environment



Iterative Reproduction

• Start with your environment

- While the problem is not reproduced, adapt more and more circumstances from the user's environment
- Iteration ends when problem is reproduced (or when environments are "identical")
- Side effect: Learn about failure-inducing circumstances

Millions of configurations Testing on dozens of different machines All needed to find & reproduce problems

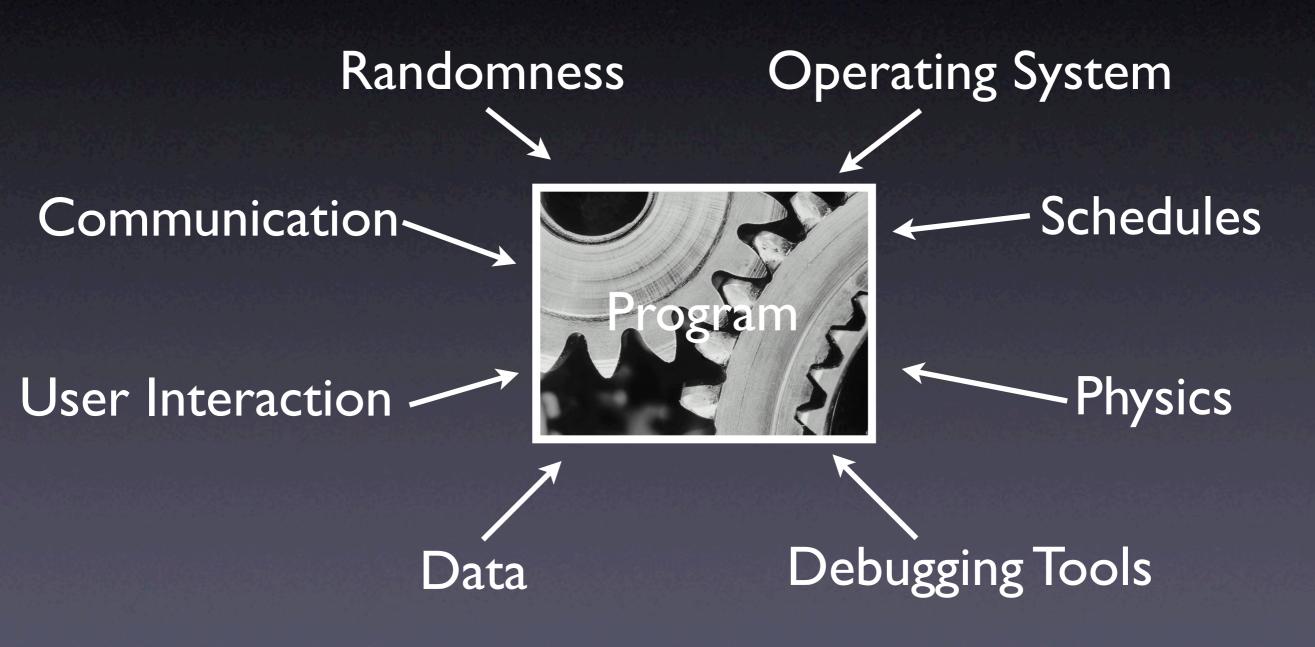
Ironment

Virtual Machines

O Back ▼ O ▼ I B C Search ☆ Favorites I Media Ø D ▼ Search ☆ Favorites I Media Ø D ♥ Search ☆ Favori	🖉 gsxwin2k3: VMware Management Interface - Microsoft Internet Explorer						
Address Address https://gsxwin2k3333/vmware/en/ Image: Solution of the server e.x.p build-7357 Administrator@gsxwin2k3 Status Monitor Options Refresh Help Log O Last updated Thu Feb 19 15:47:32 PST 2004 System Summary Processors (2) Virtual Machines 19 % Other 20 % System Total 39 % Windows 2000 Cluster Node 2 Powered off Image: Supended I	Eile Edit View Favorites Tools Help						
Wiware GSX Server e.x.p build-7357 Administrator@gsxwin2k3 Status Monitor Options Last updated Thu Feb 19 15:47:32 PST 2004 System Summary Processors (2) Virtual Machines 19 % Other 20 % System Total 39 % Windows XP Professional Powered on PID 13800 Windows 2000 Cluster Node 2 Powered off Windows 2000 Cluster Node 1 Suspended Windows 2000 Cluster Node 1 Suspended Novell NetWare 6.5 Powered on PID 3224 22 hours 31 metware 6.5 Powered on PID 3224 22 hours 23 hours							
Status Monitor Options Refresh Help Log C Last updated Thu Feb 19 15:47:32 PST 2004 System Summary 5 Minute Average Processors (2) Memory (1.5 G) Virtual Machines 894.0 M Other 20 % Other 642.0 M 642.0 M System Total 39 % System Total 1.5 G 0 Virtual Machines (8) Virtual Machines 9% CPU RAM Virtual Machines (8) Image: System Total 1.5 G 0 Virtual Machines (8) Image: System Total 1.0 G 0 Virtual Machines (8) Image: System Total 1.0 G 0 Virtual Machines (8) Image: System Total 1.0 G 0 Virtual Machines (8) Image: System Total 1.0 M 0 Image: Nowered on PID 1380 Image: System Total 0 0 0 Image: Nowered on PID 1380 Image: Suspended	Address 🔄 https://gsxwin2k3:8333/vmware/en/						
Last updated Thu Feb 19 15:47:32 PST 2004 System Summary Processors (2) Virtual Machines 19 % Memory (1.5 G) Virtual Machines 19 % 0ther 642.0 M System Total 39 % System Total 1.5 G Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) Windows XP Professional Powered on PID 1380 Powered on PID 1380 Powered on PID 1380 Image: Imag	Wind the second						
System Summary S Minute Average Processors (2) Memory (1.5 G) Virtual Machines 894.0 M Memory (1.5 G) Other 20 % Other 642.0 M Memory (1.5 G) System Total 39 % System Total 1.5 G Memory (1.5 G) Virtual Machines 894.0 M Memory (1.5 G) Other 642.0 M Memory (1.5 G) System Total 39 % System Total 1.5 G Memory (1.5 G) Memory (1.5 G) Virtual Machines 894.0 M Memory (1.5 G) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) Virtual Machines (8) System Total 1.5 G Memory (1.5 G) Virtual Machines (8) Windows XP Professional Powered on PID 1380 6 hours 7 301.0 M Virduws 2000 Cluster Node 2 Powered off Windows 2000 Cluster Node 1 Suspended Memory (1.5 G) Memory (1.5 G) Virduws 2000 Cluster Node 1 Suspended Novell NetWare 6.5 Powered on PID 3224 22 hours 3 157.0 M Virduws Server 2003 25 hours 2 25 hours 2 176.0 M	Status Monitor Options Refresh Help Log Out						
Processors (2) Memory (1.5 G) Virtual Machines 19 % Other 20 % System Total 39 % Other 642.0 M System Total 1.5 G Virtual Machines (8) System Total Virtual Machines (8) Windows XP Professional Powered on PID 1380 Image: Supended 6 hours Image: Supended Suppended Image: Suppended Image: Suppended Image: Suppended 22 hours Image: Supended 25 hours Image: Supended 25 hours Image: Supended 25 hours Image: Supended 25 hours	Last updated Thu Feb 19 15:47:32 PST 2004						
Virtual Machines 19 % Virtual Machines 894.0 M Other 20 % Other 642.0 M System Total 39 % System Total 1.5 G Virtual Machines (8) Virtual Machines (7) Windows XP Professional Powered on PID 1380 Other Virtual Machines (7) Windows 2000 Cluster Node 2 Powered off Windows 2000 Cluster Node 1 Suspended Windows 2000 Cluster Node 1 Suspended Novell NetWare 6.5 Powered on PID 3224 Virtual Machines Virtual Machines Suspended Virtual Machines Vindows 2000	System Summary 5 Minute Average					Average	
Other 20 % Other 642.0 M Minimum System Total 39 % System Total 1.5 G System Total Virtual Machines (8) Yethod was 2000 Cluster Node 2 Powered on PID 1380 6 hours 7 301.0 M Yethod was 2000 Cluster Node 2 Yethod was 2000 Cluster Node 1 Yethod was 2000 Cluster Node 1 Yethod was 2000 Cluster Node 1 Yethod was 2000 Cluster Node 1	Processors (2)	Memory (1.5 G)				
System Total 39 % System Total 1.5 G Virtual Machines (8) Windows XP Professional Powered on PID 1380 0 hours 7 10000 Image: Constraint of the stress of the							
Virtual Machines (8) HB Display Name Up % CPU RAM Mindows XP Professional Powered on PID 1380 6 hours 7 301.0 M Mindows 2000 Cluster Node 2 Powered off Mindows 2000 Cluster Node 1 Suspended Mindows 2000 Cluster Node 2 Suspended Mindows 200							
HB Display Name Up % CPU RAM Image: Im							
Image:	Virtual Machines (8)						
Image: Second	HB			Up	% CPU	RAM	
Image: Second	🚂 Ť 🕨 🔳			6 hours	7	301.0 M	
Image: Suspended Image: Suspended Image: Suspended					000000000		
Image: Suspended Image: Suspended Image: Suspended							
Image: Second							
		Powered on PID 3224			3	157.0 M	
	🌆 T 🕨 🔳	Powered on PID 2560			8	176.0 M	
Image: Section of the section of th		Powered on PID 1592		35 hours	1 00000000	260.0 M	
Image: Superiod Image: Superiod Image: Superiod Image: Superiod	🛃 🚺 🗏				000000000		
Download VMware Virtual Machine Console: Windows (exe) Linux (rpm) Linux (tar.gz)	achine						
Copyright © 1998-2004 VMware, Inc. All rights reserved. Protected by one or more of U.S. Patent Nos. 6,397,242 and 6,496,847; patents pending.							
🙆 Active 🕒 🔒 🥥 Internet	Calive Active				🧐 Internet		

Reproducing Execution

- After reproducing the environment, we must reproduce the execution
- Basic idea: Any execution is determined by the input (in a general sense)
- Reproducing input → reproducing execution!







Data

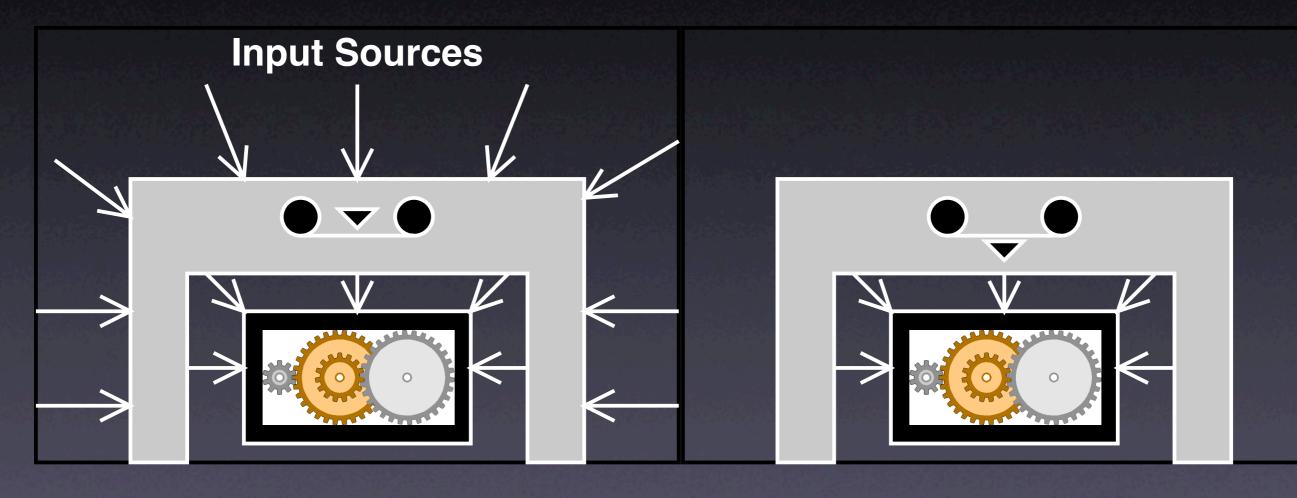
Easy to transfer and replicate
Caveat #1: Get all the data you need
Caveat #2: Ger only the data you need
Caveat #3: Privacy issues



User Interaction -

Data

User Interaction



Record

Replay

Recorded Interaction

send_xevents key H @400,100 send_xevents wait 376 send_xevents key T @400,100 send_xevents wait 178 send_xevents key T @400,100 send_xevents wait 214 send_xevents key P @400,101 send_xevents wait 537 send_xevents keydn Shift_L @400,101 send_xevents wait 218 send_xevents key ";" @400,101 send_xevents wait 167 send_xevents keyup Shift_L @400,101 send_xevents wait 1556 send_xevents click 1 @428,287 send_xevents wait 3765

Communication ~

User Interaction -



Data

Communication

- General idea: Record and replay like user interaction
- Bad impact on performance
- Alternative #I: Only record since last checkpoint (= reproducible state)
- Alternative #2: Only record "last" transaction

Randomness

Data

Communication~

User Interaction -



Randomness

 Program behaves different in every run Based on random number generator • Pseudo-random: save seed (and make it configurable) • Same applies to time of day • True random: record + replay sequence

Randomness

Data

Operating System

Communication

User Interaction -



Operating System

- The OS handles entire interaction between program and environment
- Recording and replaying OS interaction thus makes entire program run reproducible

A Password Program

#include <string>
#include <iostream>
using namespace std;

```
string secret_password = "secret";
```

```
$ c++ -o password password.C
$ ./password
Enter your password: secret
Access granted.
$
```

```
int main()
{
```

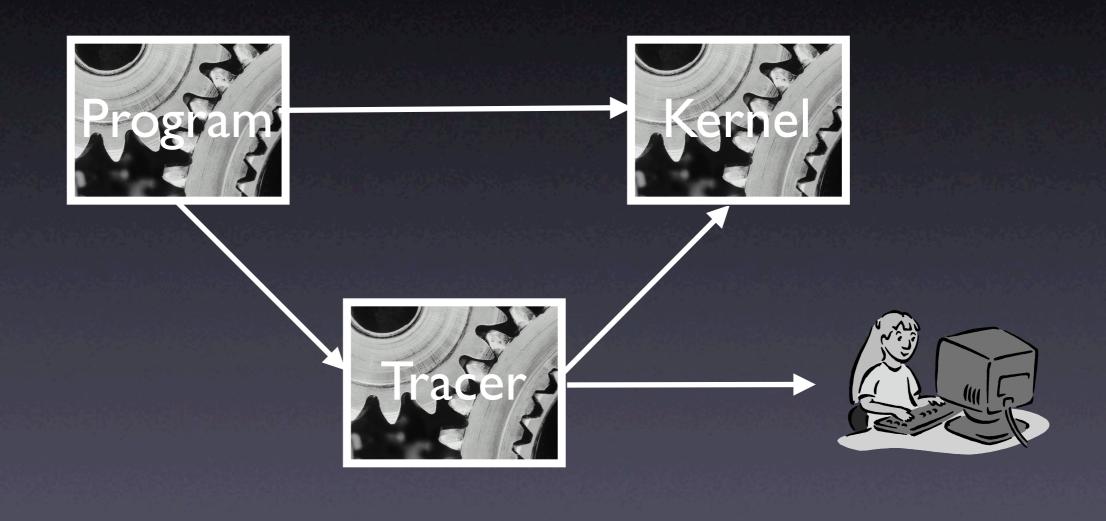
```
string given_password;
cout << "Please enter your password: ";
cin >> given_password;
if (given_password == secret_password)
    cout << "Access granted." << endl;
else
    cout << "Access denied." << endl;</pre>
```

Traced Interaction

```
$ c++ -o password password.C
$ strace ./password 2> LOG
Enter your password: secret
Access granted.
$ cat LOG
```

```
write(1, "Please enter your password: ", 28) = 28
read(0, "secret\n", 1024) = 7
write(1, "Access granted.\n", 16) = 16
exit_group(0) = ?
```

How Tracing works



Replaying Traces







Trace File

Challenges

- Tracing creates lots of data
- Example: Web server with 10 requests/sec
 A trace of 10 k/request means 8GB/day
- All of this must be replayed to reproduce the failure (alternative: checkpoints)
- Huge performance penalty!

Randomness

Data

Operating System

Communication~

User Interaction -





Accessing Passwords

Thread A

Thread B

open(".htpasswd") read(...) modify(...) write(...) htpasswd file open(".htpasswd") read(...)modify(...) write(...) close(...)

Lost Update

Thread A

Thread B

open(".htpasswd") open(".htpasswd") read(...) read(...) modify(...) write(...) close(...) modify(...) write(...) close(...)

A's updates get lost!

Reproducing Schedules

Thread changes are induced by a scheduler
 It suffices to record the schedule (i.e. the moments in time at which thread switches occur) and to replay it

• Requires deterministic input replay

Constructive Solutions

- Lock resource before writing
- Check resource update time before writing
- ... or any other synchronization mechanism

Randomness

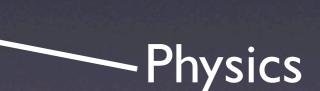
Data

Operating System

Communication~

User Interaction -





Schedules

Physical Influences

- Static electricity
- Alpha particles (not cosmic rays)
- Quantum effects
- Humidity
- Mechanical failures + real bugs

Rare and hard to reproduce

Randomness

Data

Communication ~

User Interaction -



Debugging Tools

Operating System

Schedules

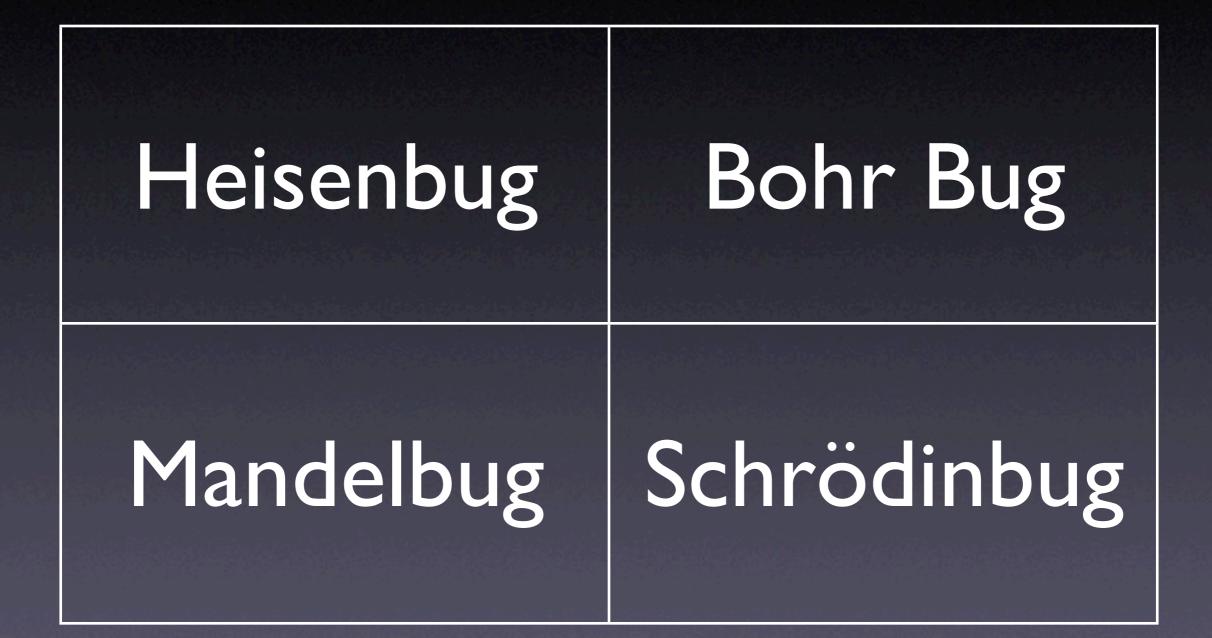
Physics

A Heisenbug

Code fails outside debugger only
 int f() {
 int i;
 return i;

In program: returns random value In debugger: returns 0

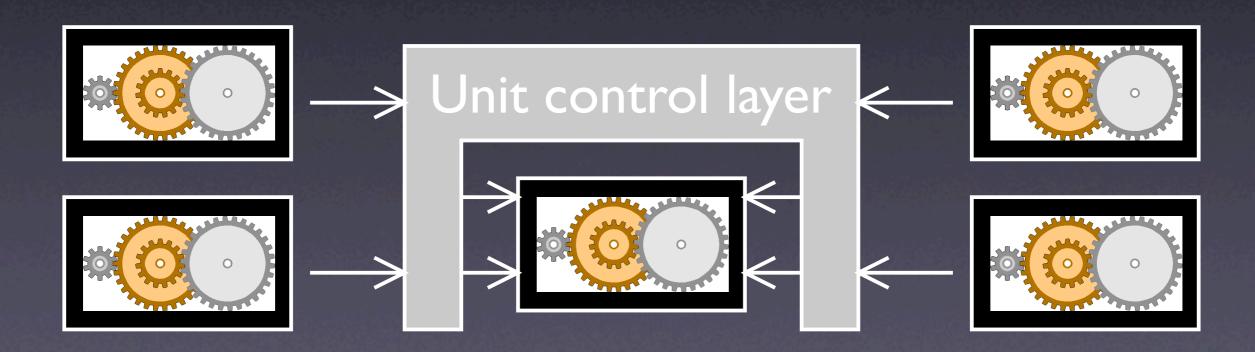
More Bugs



Isolating Units

• Capture + replay unit instead of program

• Needs an unit control layer to monitor input



Isolated Units

- Databases. Replay only the interaction with the database.
- Compilers. Record + replay intermediate data structures rather than the entire front-end.
- Networking. Record + replay communication calls.

A Control Example

```
class Map {
public:
    virtual void add(string key, int value);
    virtual void del(string key);
    virtual int lookup(string key);
};
```

• How do we control this?

A Log as a Program

#include "Map.h"
#include <assert>

}

```
int main() {
    Map map;
    map.add("onions", 4);
    map.del("truffels");
    assert(map.lookup("onions") == 4);
    return 0;
```

This is a log file (and also a program)
How do we get this?

Controlled Map

class ControlledMap: public Map {
 public:
 typedef Map super;

virtual void add(string key, int value); virtual void del(string key); virtual int lookup(string key);

ControlledMap();
~ControlledMap();

};

// Constructor
// Destructor

Logging

```
void ControlledMap::add(string key, int value) {
    clog << "map.add(\"" << key << "\", "
         << value << ");" << endl;
    Map::add(key, value);
}
                                         map.add("onions", 4);
void ControlledMap::del(string key) {
    clog << "map.del(\"" << key << "\");" << endl;
    Map::del(key);
}
                                          map.del("truffels");
virtual int ControlledMap::lookup(string key) {
    clog << "assert(map.lookup(\"" << key << "\") == ";</pre>
    int ret = Map::lookup(key);
    clog << ret << ");" << endl;
    return ret;
}
                           assert(map.lookup("onions") == 4);
```

Logging Fixture

```
ControlledMap::ControlledMap()
{
    clog << "#include \"Map.h\"" << endl</pre>
         << "#include <assert>" << endl
         << "" << endl
         << "int main() {" << endl
         << " Map map;" << endl;
}
ControlledMap::~ControlledMap()
{
    clog << " return 0;" << endl;</pre>
         << "}" << endl;</pre>
}
```

More Interaction

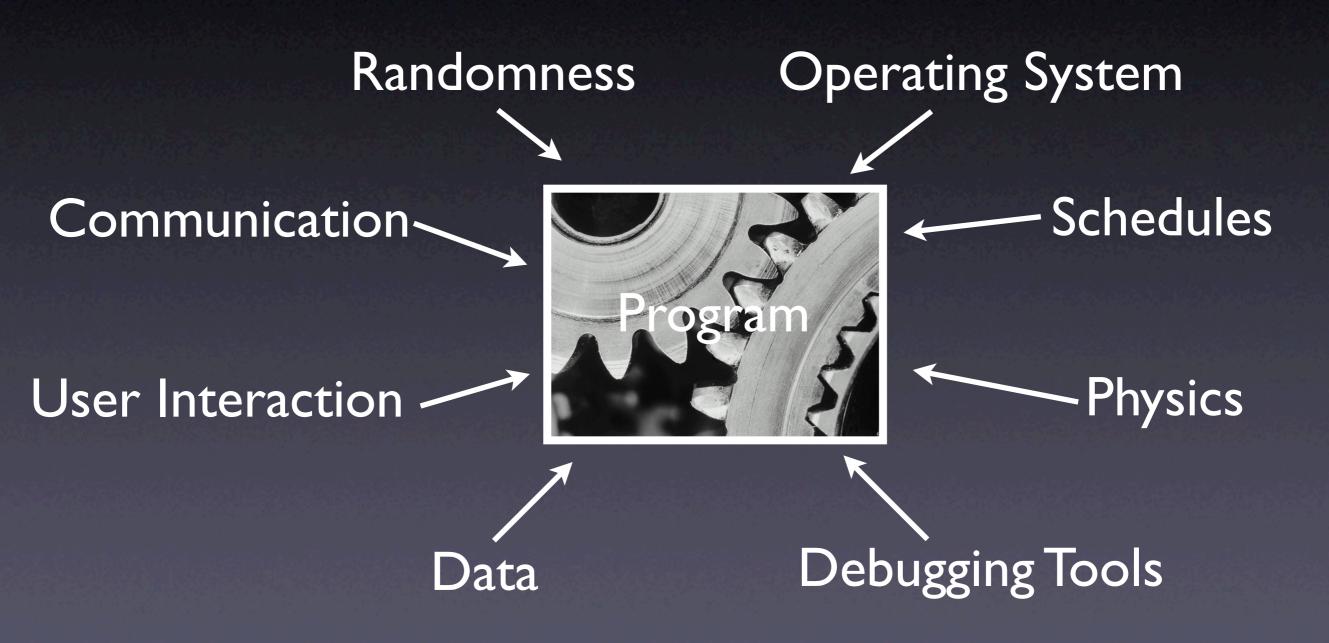
- Variables (hard to detect)
- Other units (break dependency if needed)
- Time (record + replay, too)

Concepts

* Once a problem is tracked, one must reproduce it in the own environment

★ To reproduce a problem...

- reproduce the environment (by adopting one circumstance after the other)
- reproduce the execution (by controlling the input of the program or a unit)



This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit

http://creativecommons.org/licenses/by/1.0

or send a letter to Creative Commons, 559 Abbott Way, Stanford, California 94305, USA.