Reproducing Problems

Andreas Zeller

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

Where to reproduce?	Chances of Success	Costs
User	+	--
Developer	o	+

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

Setting up the Environment

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

Virtual Machines

gsxwin2k3: VMware Management Interface - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address https://gsxwin2k3:8333/vmware/e/ Go Links

VMware GSX Server 3.x.p Build 7857 | Administrator@gsxwin2k3

Status Monitor | Options | Refresh | Help | Log Out

Last updated Thu Feb 19 15:47:32 PST 2004

System Summary — 5 Minute Average

Processors (2)		Memory (1.5 G)	
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)

		HB	Display Name	Up	% CPU	RAM
	▶		Windows XP Professional Powered on | PID 1380	6 hours	7	301.0 M
	■		Windows 2000 Cluster Node 2 Powered off			
	❚❚		Windows 2000 Cluster Node 1 Suspended			
	❚❚		WinNT IIS Web Server Suspended			
	▶		Novell NetWare 6.5 Powered on | PID 3224	22 hours	3	157.0 M
	▶		Windows Server 2003 Powered on | PID 2560	35 hours	8	176.0 M
	▶		Red Hat Enterprise Linux 3 Powered on | PID 1592	35 hours	1	260.0 M
	❚❚		SuSE Linux Enterprise Server 8 Suspended			

Download VMware Virtual Machine Console: Windows (exe) | Linux (rpm) | Linux (tar.gz)

Add Virtual Machine

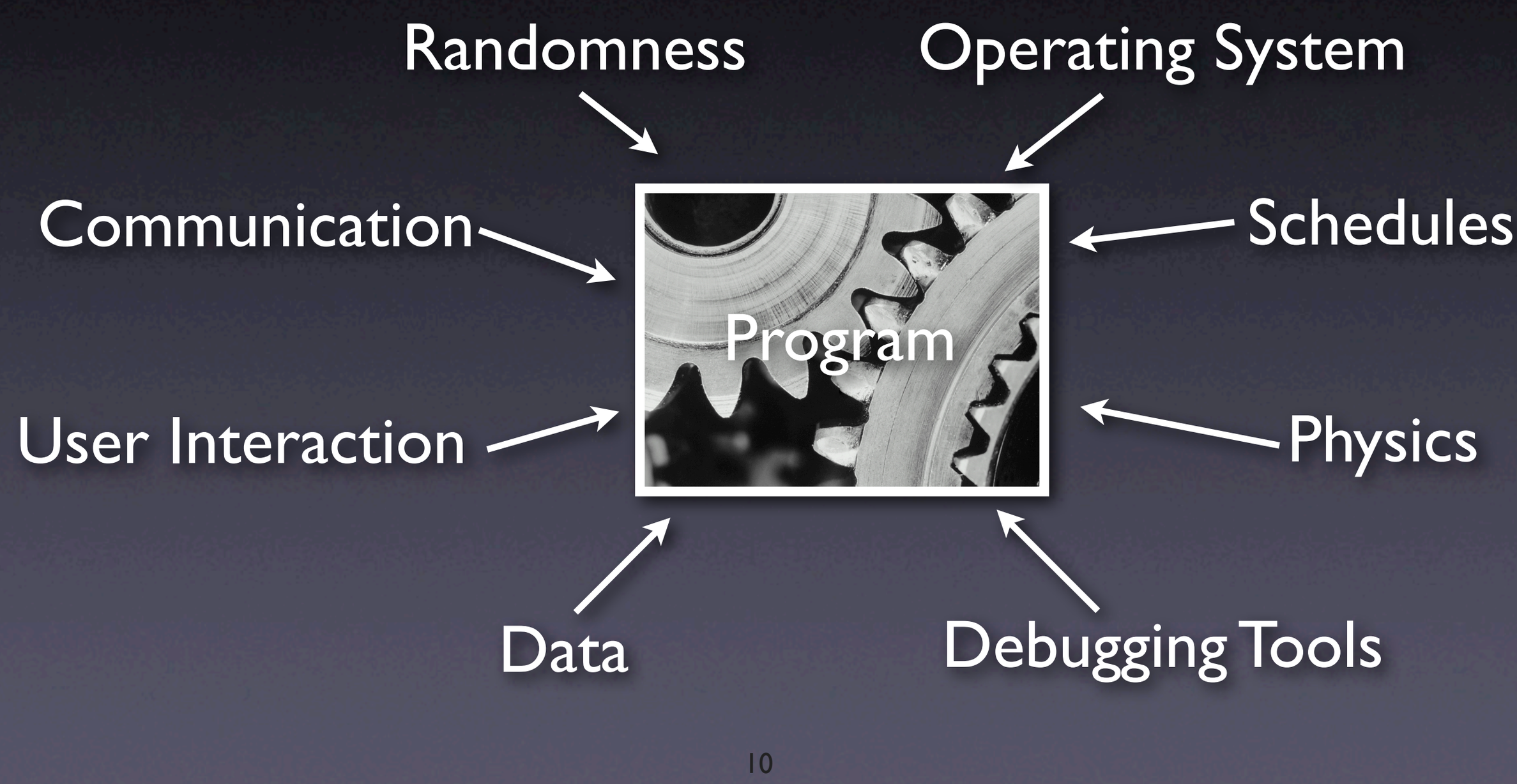
Copyright © 1998-2004 VMware, Inc. All rights reserved.
Protected by one or more of U.S. Patent Nos. 6,397,242 and 6,496,847; patents pending.

Active Internet

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!

Program Inputs



Program Inputs

Data

Data

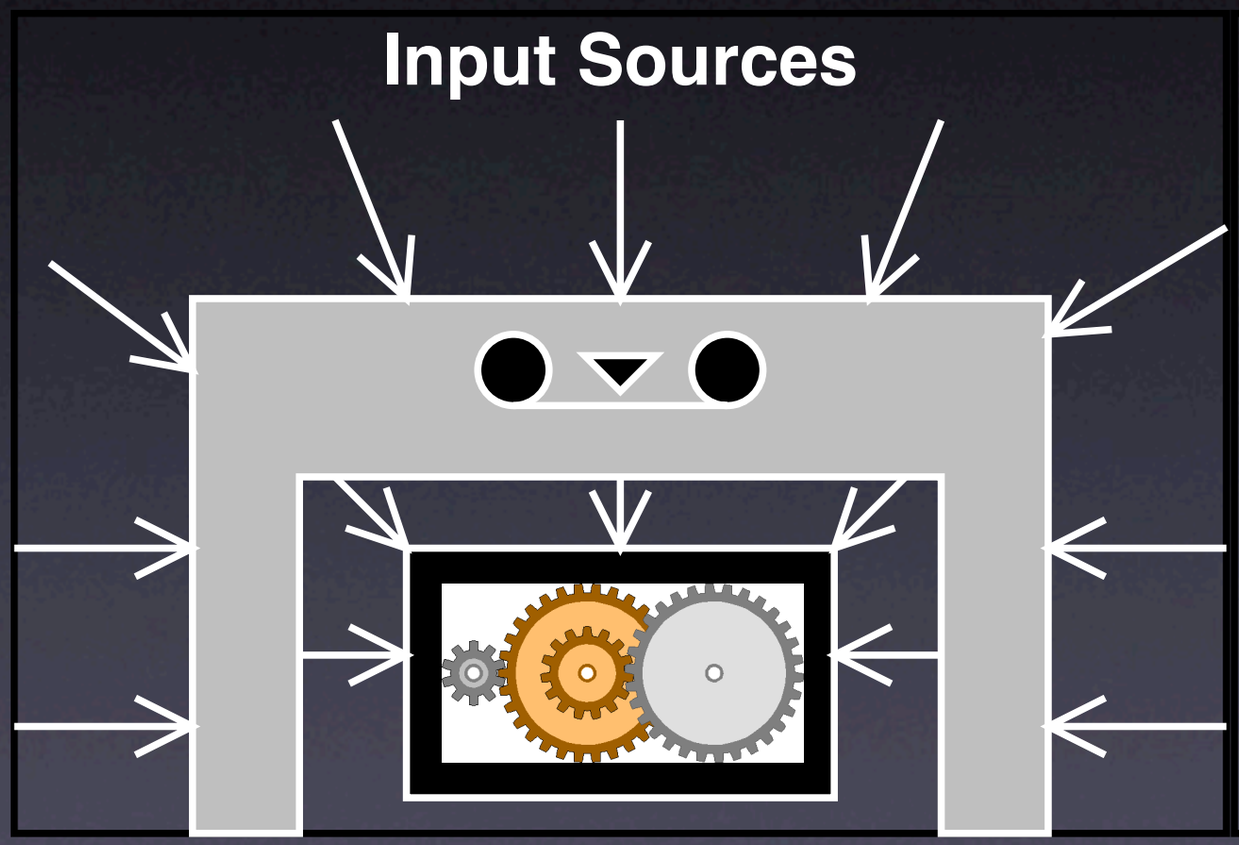
- Easy to transfer and replicate
- Caveat #1: *Get all the data you need*
- Caveat #2: *Get only the data you need*
- Caveat #3: Privacy issues

Program Inputs

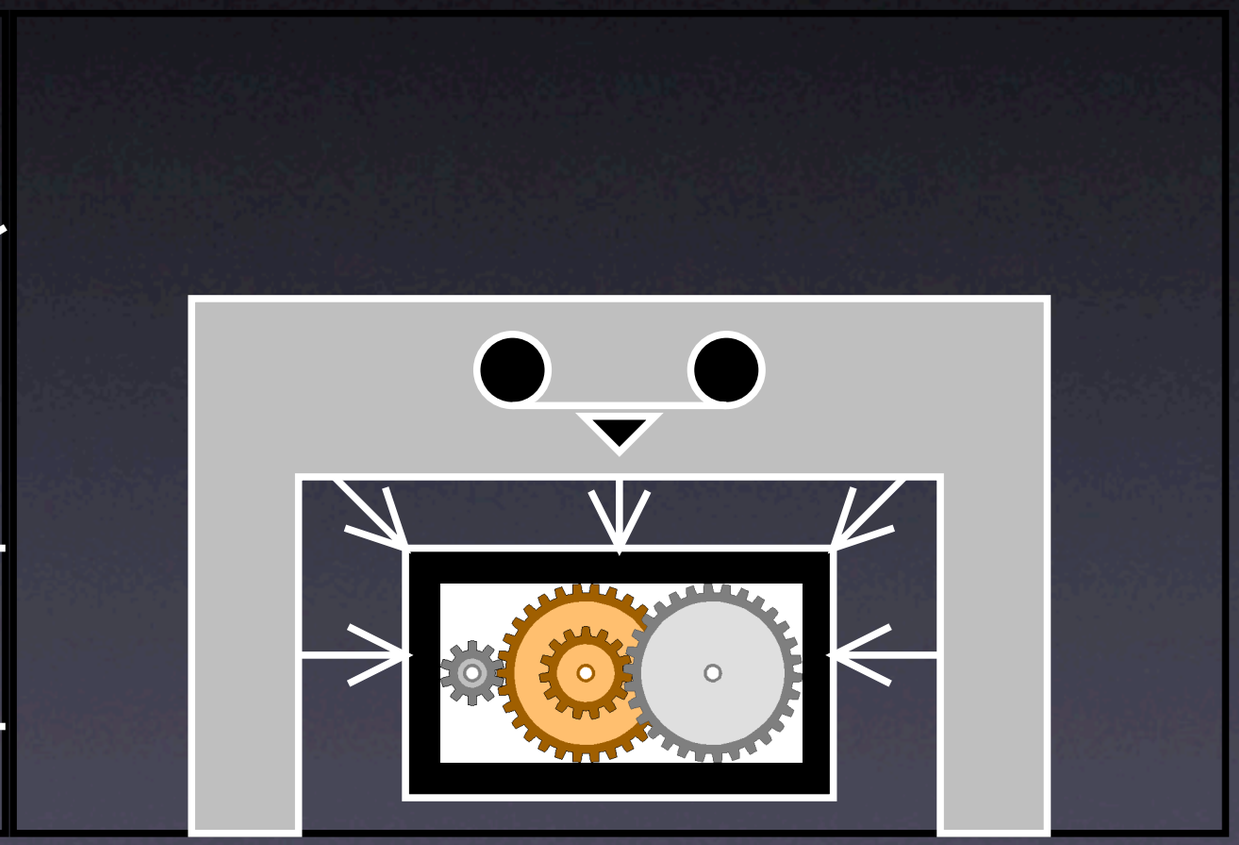
User Interaction

Data

User Interaction



Record

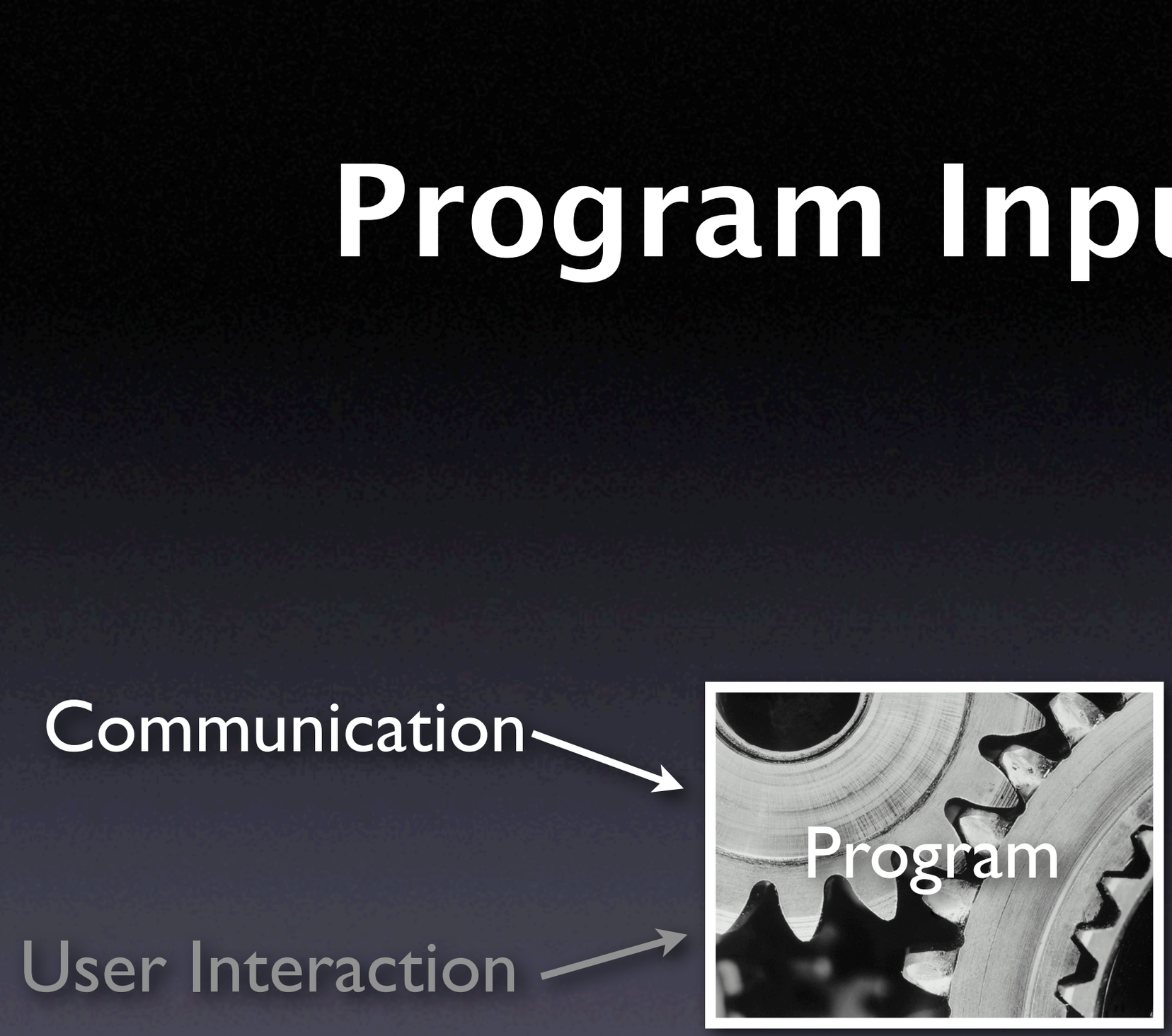


Replay

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
```

Program Inputs



Communication

- General idea: Record and replay like user interaction
- Bad impact on performance
- Alternative #1: Only record since last *checkpoint* (= reproducible state)
- Alternative #2: Only record “last” transaction

Program Inputs

Randomness

Communication

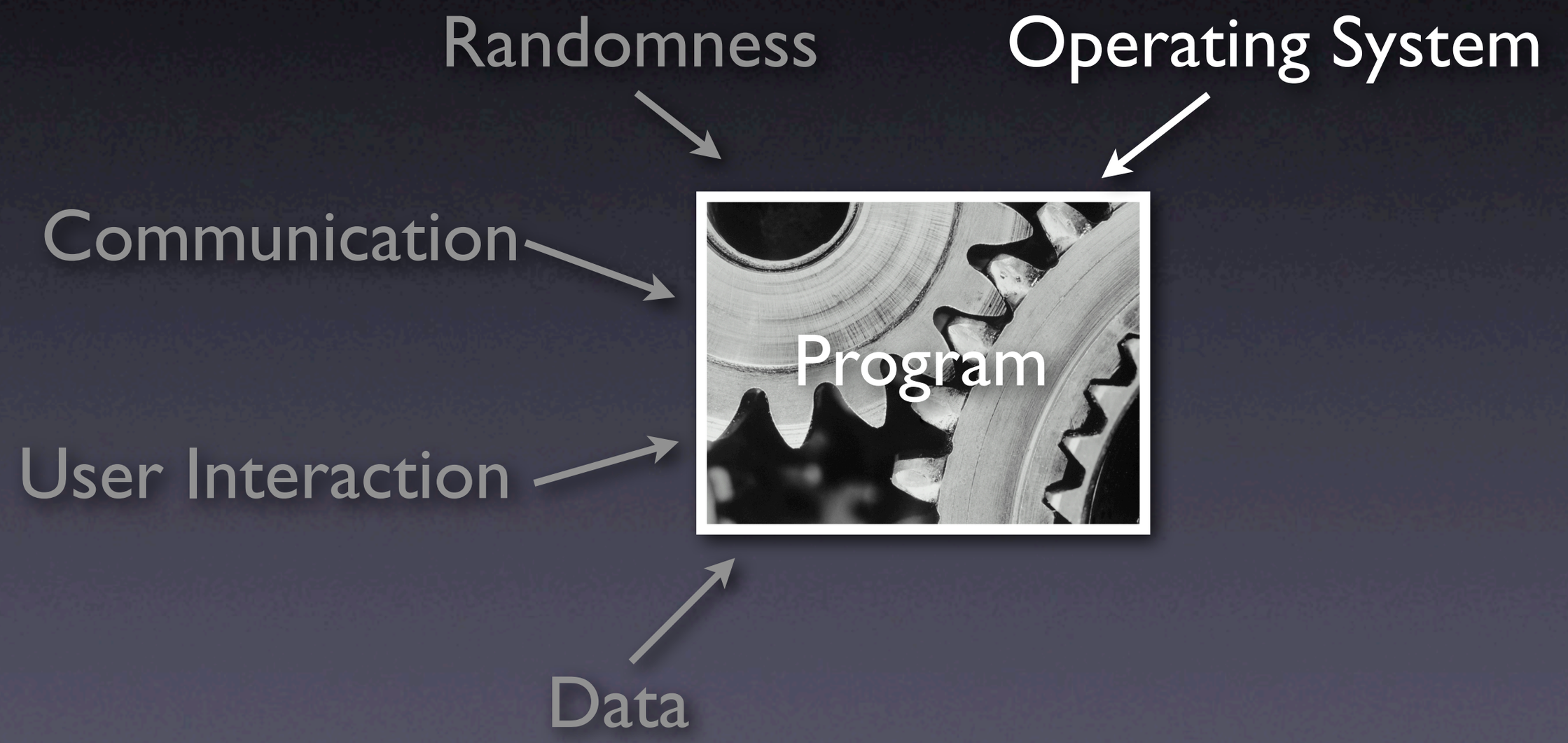
User Interaction

Data

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

Program Inputs



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

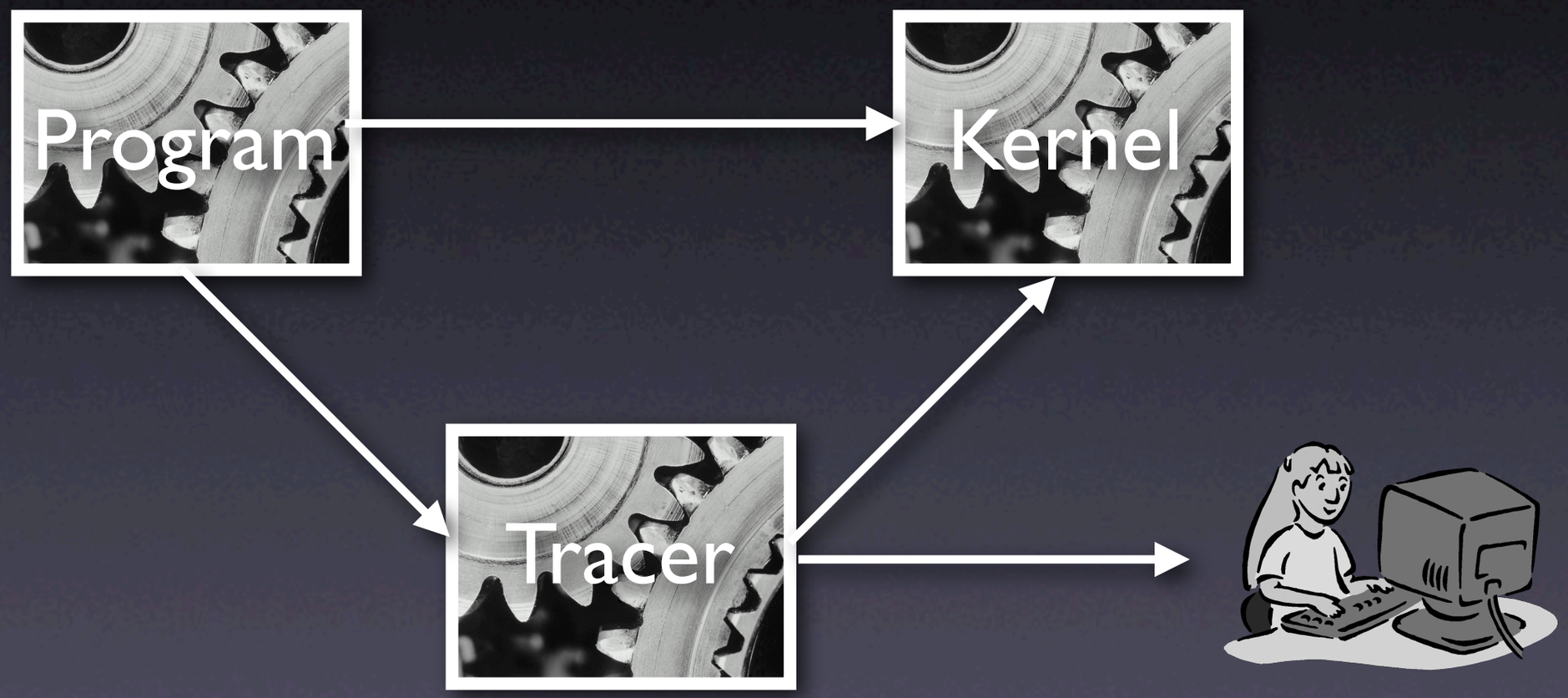
```
#include <string>
#include <iostream>
using namespace std;

string secret_password = "secret";

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;
}
```

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

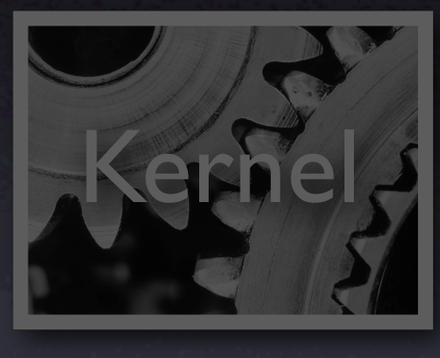
Traced Interaction

```
$ 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!

Program Inputs

Randomness

Operating System

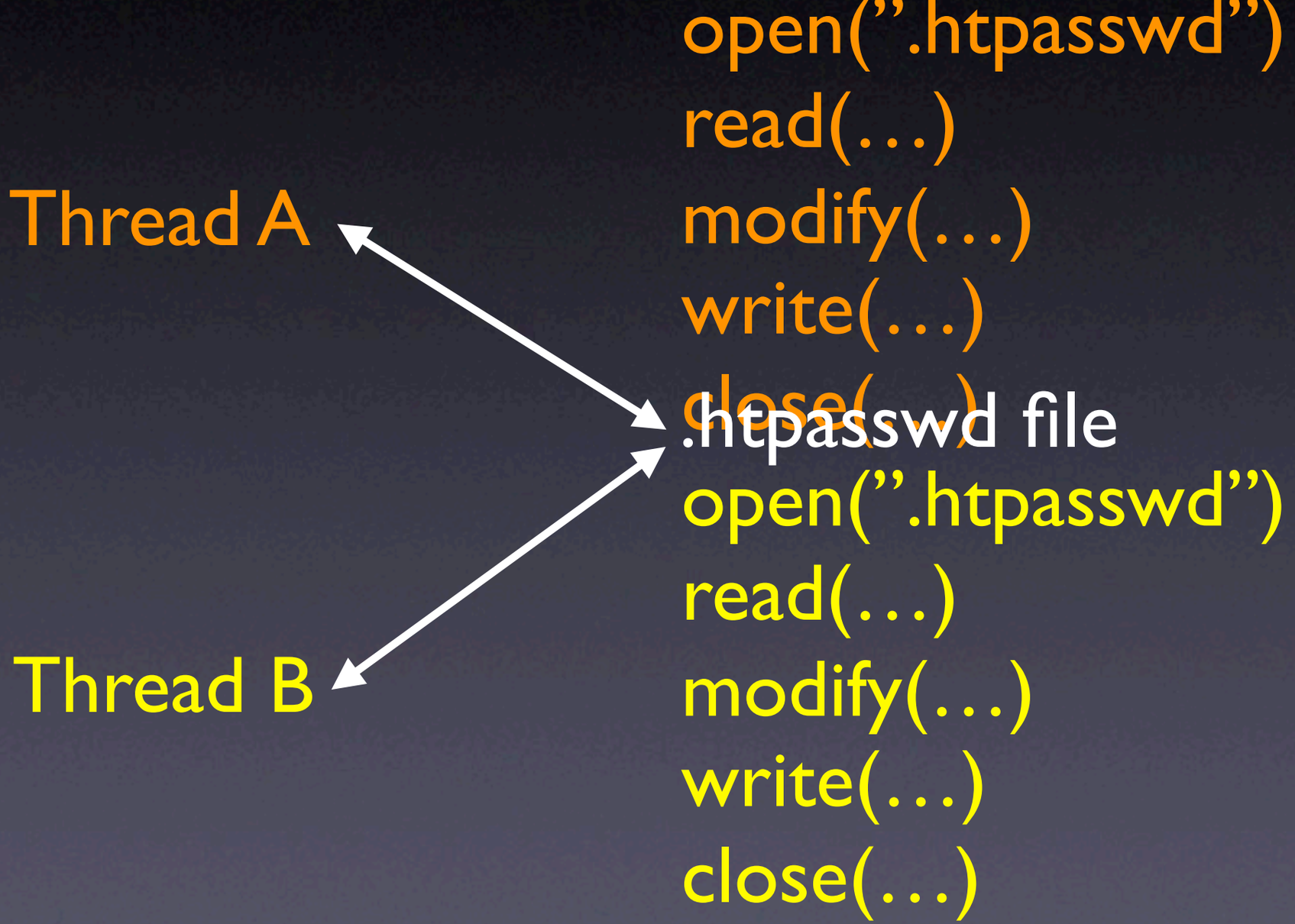
Communication

Schedules

User Interaction

Data

Accessing Passwords



Lost Update

Thread A

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

Thread B

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

A's updates get lost!

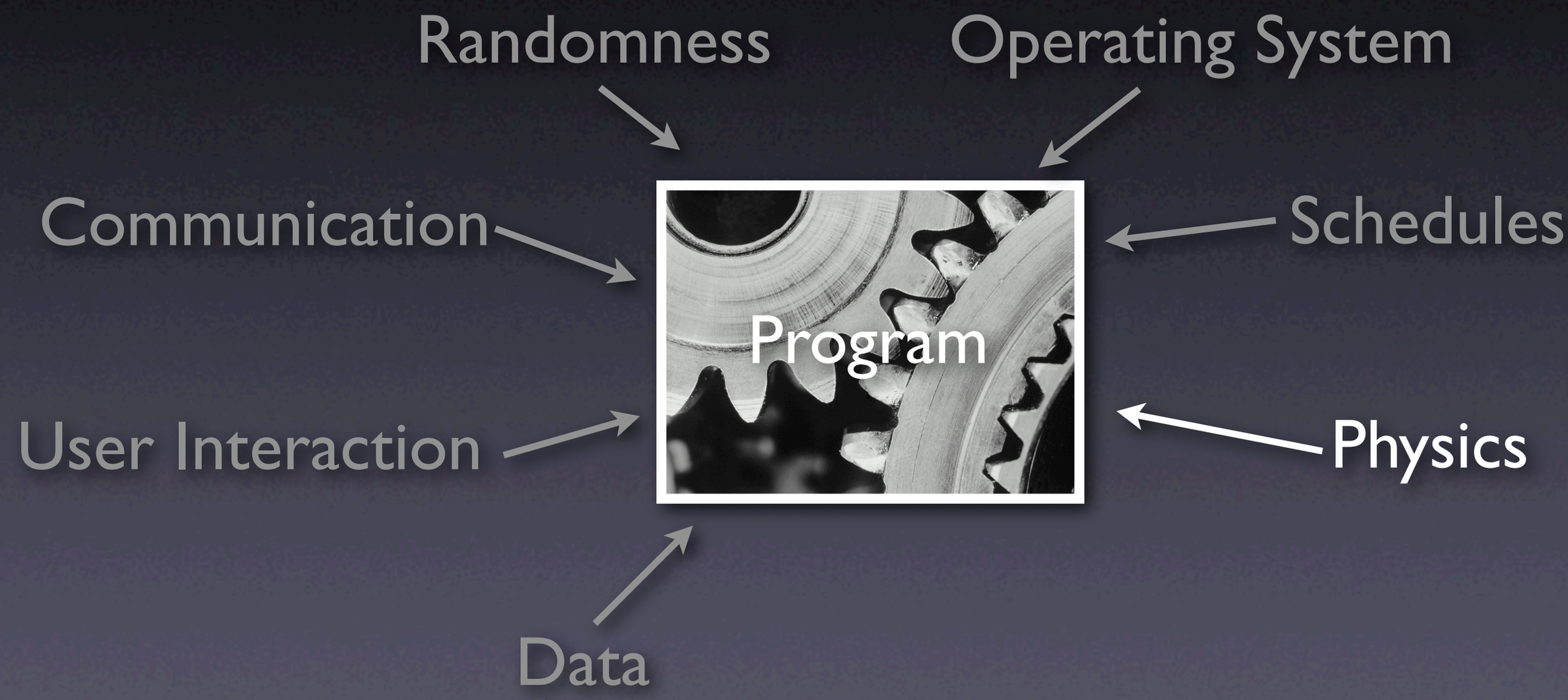
Reproducing Schedules

- 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*

Program Inputs

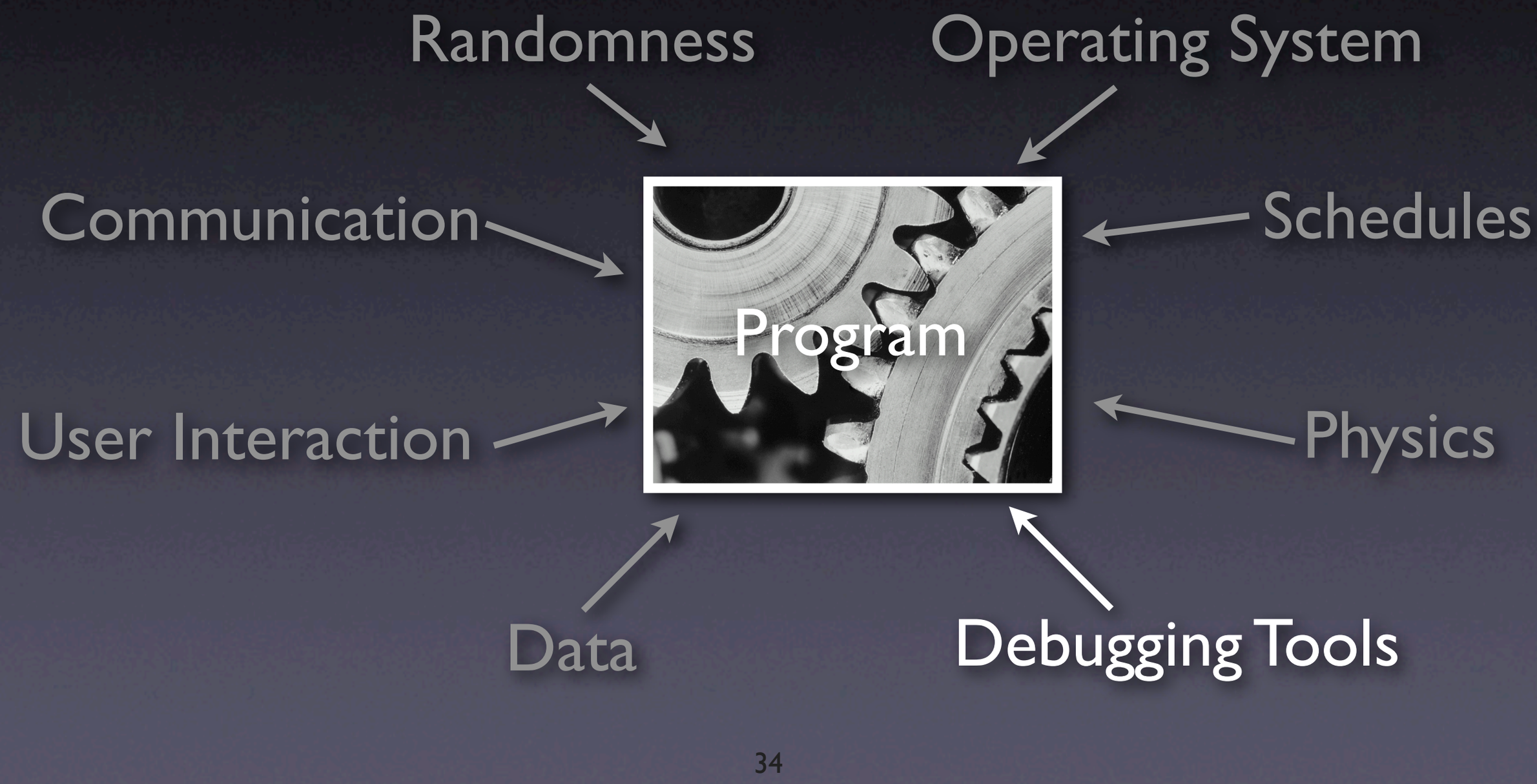


Physical Influences

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

Rare and hard to reproduce

Program Inputs



A Heisenbug

- Code fails outside debugger only

```
int f() {
    int i;
    return i;
}
```

In program:
returns random value

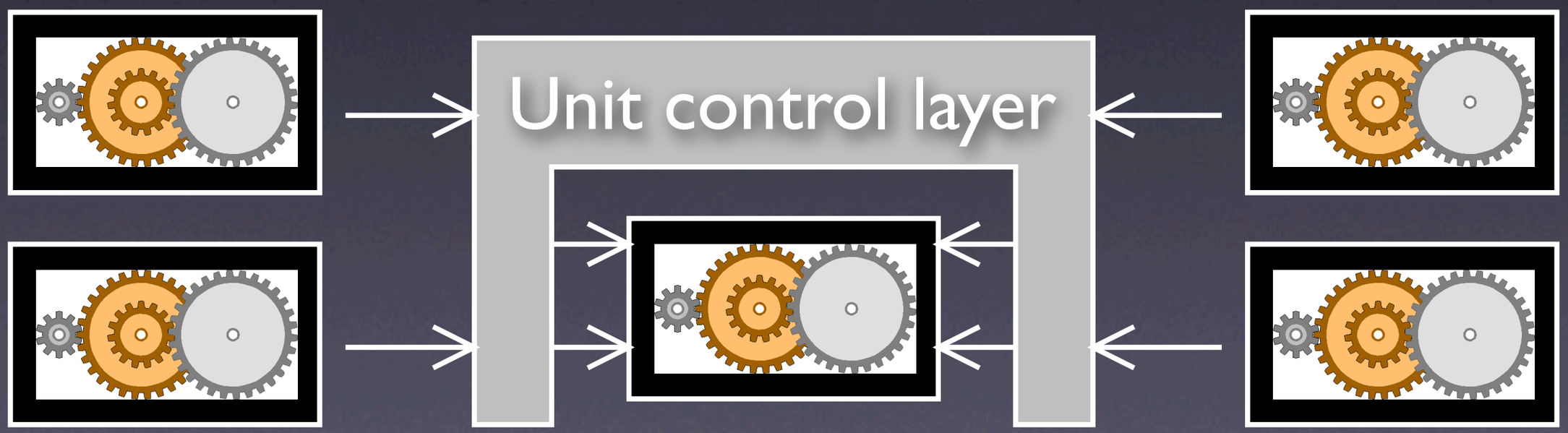
In debugger:
returns 0

More Bugs

- Heisenbug
  - A bug that disappears when you attempt to find it
- Bohr Bug
  - Repeats reliably and consistently; opposite of Heisenbug
- Mandelbug
  - Bug that's difficult to find/fix because of complexity/unpredictability
- Schrödinbug
  - The bug manifests only after someone reads the source and realizes the code never should have worked in the first place. Then the code stops working until fixed.

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.

Logging via Proxies

```
interface Map {
  void add(string key, int value);
  void del(string key);
  int lookup(string key);
};
```

```
// create a log that looks like Map calls!
class MapLogger implements Map {
  Map theMap;
  MapLogger(Map m) { theMap =  m; }
  void add(string key, int value) {
    System.out.println(“map.add(“ + key + “, “  +
     value  + “);”; //
 }
  // etc
}
```

More Interaction

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

Mock Objects

- A *Mock Object simulates* an original object
- Its implementation tells how to react on specific calls (i.e. returning other mock objects)
- Can be combined with *recording*, too!

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.