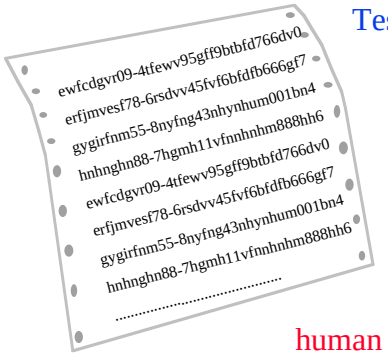
Test Oracles

What is an oracle?

An "Inspector" of executions:
do test executions produce acceptable results?

An oracle can be:
human being
machine
a former version of the same program
another program
.....

What is a good oracle?



Testing large, complex applications may require millions of test runs

The size of the outputs to be inspected exceed the capabilities of human eyes

human eyes are slow and unreliable examiners even of small number of outputs

So ... automated oracles are essential

Can oracles be automatically produced?

Oracles can be easily derived from a "golden version" of the program

BUT...

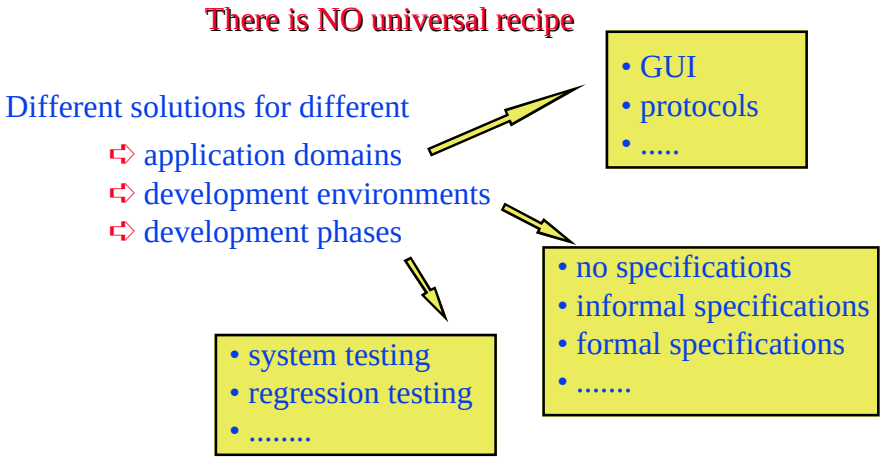
how often do we have a "golden version?"

almost never ....

what to do then?

let's try to find an acceptable approximation...

How can we build acceptable oracles?

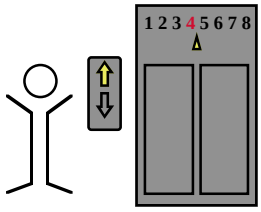


Oracles from System Specifications

- An essential part of requirements specification: Making specified properties *checkable*
- System oracles are designed early
  - NOT after system design
- Subsystem oracles are a part of architectural design and system build plan
  - "design for test"

Narrowing for Checkability

•**Example:** Elevator response



*Uncheckable requirements can often be **narrowed** to checkable properties (often sufficient but not necessary conditions)*

- **Objective:** Passengers are not frustrated by waiting
- **Specified property:** Elevator responds within 60 seconds, 99% of the time
- **Excluded solution:** Install a mirror in the waiting area

Oracles from Design

Example: UML design notations

- Message sequence charts
  - A UML message sequence chart indicates a *test case* and *expected outcome,* which can be interpreted by a *driver* and *oracle*
  - *Typical of "scenario-based" oracles*
    - scenarios combine test case with special oracle
- StateChart (finite state acceptor)
  - A UML finite state machine describes *all* permissible behaviors of a module
    - oracle can be used with large numbers of automatically generated test cases

Oracles from Code Documentation

Parnas' tabular annotations precisely describe the functional behavior of the unit. The table can be evaluated with respect to the produced outputs to check for their correctness .

DISPLAY 1 *Display 1 Specification* *

Find(x,A,j,present)

$R_0(,) = ((1 \le n)$ **and** $[$**for all** $(1 \le i \le n) \Rightarrow$ ™A[ι] ″∏A[ι+1]]) ⇒ ∃ ι[(1″ι″ ν) αvδ (™A[ι]=™ξ)] =

	true	*false*	
j' |	'A[j]='x	*true*	
present'=	true	false	and NC(x,A)

Display 1 Program

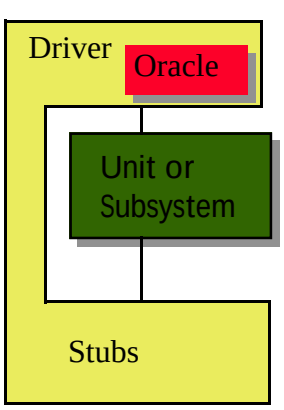
procedure find (...)
.....
end {find}

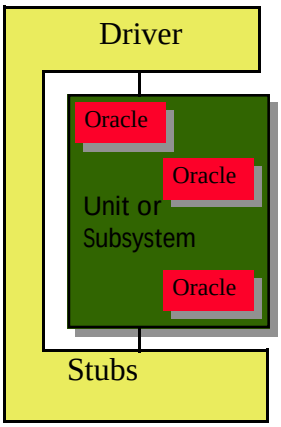
Display 1 Specifications of the Invoked Programs

.....

* from: Parnas, Madey, Iglewski, *Precise Documentation of Well-Structured Programs*, IEEE-TSE Vol.. 20 N. 12 Dec 1994

Harness vs. Embedded Assertions





Embedded assertions act as oracles within the unit under test

Assertions as Oracles

```
/*
 * Alphabetic sort of an array of strings
 */
void  sort( char *words[ ],  int nwords )
{
    . . .
    assert( is_sorted(words, nwords) );
    return;
}
```


Oracle Design Exercise 1

Devise an oracle for this function:

void escape_html_specials(char *in, char * out);

- Input: String of ASCII text
- Output: Same string EXCEPT each special html character (&, #, <, etc.) is replaced by entity name or numeric code
- Example: "<Analysis & Test>" becomes "<Analysis & Test>"

Oracle Exercise 2: Shortest Path

- Devise an oracle for a module or program that finds the shortest weighted path in a graph
  - or, more realistically: find train route from city X to city Y minimizing changes and total time
- Input: source city, destination city, set of (A,B,d) where A and B are cities, d is a positive integer
- Output: sequence (source,x1,d0), (x1,x2,d1), ... , (xn,dest,dn) minimizing d0+d1+...+dn

Automation

- Capture/Playback/Compare
  - for interaction scenarios
  - especially for regression testing
  - typical problem: low-level compare
    - ex: bitmap compare fails because of changed font
- Scripting with expected output checks
    - similar limitations as playback/compare
- Embedded assertions & self-checks
  - simple support (e.g., C assert macro) in most languages, typically with compile-time disabling
    - support for quantification, “before” values, etc. is currently rare (available in some research tools; also Gnu “nana”)