3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
Testing

- The fact: Testing is the process of executing a program with the intent of finding errors. [Myers 1979]

- Our hope: The more thoroughly a program has been tested, the higher the probability that the program will behave as expected also in the non-tested cases

- Good to know: The correctness of a program can’t be proven by testing (except in trivial cases); this is due to combinatorial explosion of input values to be tested
Expected results must be known

- A crucial prerequisite for testing is knowing the expected results
  - Either from a specification
  - or by comparing the outcome with the results of a successful previous test run (so-called regression testing)
Testing systematics

- “Let’s run it”: A developer “tests” with some ad-hoc created data – the test is passed when the results “look good”

- **Throwaway-Test**: Somebody creates test cases and executes them, but the tests
  - are not documented
  - can’t be repeated
  - don’t have defined criteria when to stop
Testing systematics – 2

- **Systematic test**: Trained testers create, run and document the tests
  - Test is planned
  - Test procedure has been written beforehand
  - Test is executed according to test procedure
  - Expected and observed results are compared; any deviation is recorded
  - Searching and fixing defects are performed separately
  - A failed test is repeated after fixing the defects
  - Test results are documented
  - Test ends, when a pre-defined testing goal has been achieved
Forms of testing

- Artifacts to be tested may be modules, partial systems or systems

- Unit Test (or component test)

- Integration test

- System test
Forms of testing – 2

- **Acceptance test**
  - A special form of testing
  - **Not** about finding errors
  - But: demonstrate that the system satisfies its requirements; i.e., that the acceptance test cases don’t reveal any faults
The process of testing

ница Planning
- Embed testing into the development plan:
  - Which documents to create
  - Deadlines and cost for test preparation, execution and evaluation
- Who will be testing

ница Preparation
- Selection of test cases
- Setting up the test environment / test harness
- Writing the test procedure
The process of testing – 2

❖ Execution
  ● Install test environment / test harness
  ● Run tests according to test procedure; record results
  ● Don’t modify the tested artifact while executing a test
  ● Repeat failed tests after fault fixing

❖ Evaluation
  ● Assemble findings

❖ Fault fixing (no part of the testing process!)
  ● Analyze errors/symptoms found
  ● Find defects (debugging)
  ● Fix defects
Determining test cases

- Black-box (function-oriented)
  - Function coverage
  - Output coverage
  - Exception coverage
  - Attribute coverage
- Glass-box (structure-oriented)
  - Control flow coverage
  - Dataflow analysis
  - Symbolic testing
- Determining test cases
- Equivalence classes
  - Boundary values
  - Cause-effect graphing
  - Statistical testing
  - Error guessing
- Statement coverage
  - Branch coverage
  - Path coverage
- Determining test cases
3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
Branch coverage: create test cases such that all branches of the program are covered.

For this fragment, two test cases achieve 100% coverage:

```
... VAR
  a,b,x: INTEGER;
... BEGIN
  ... IF (a>1) AND (b=0)
    THEN x := x DIV a;
  IF (a=2) OR (x>1)
    THEN x := x+1;
  ...
```

Test cases:
- a=2, b=1, x=1
- a=3, b=0, x=3

```
[Myers 1979]
```

This section extends the discussion on Glass-Box-Testing in Chapter 8 of my 2nd year course on Software Engineering.
Branch coverage has a problem

Classic branch coverage has a problem:
Imagine, the specification states

\[
\begin{align*}
x' &= \begin{cases} 
  x \text{ DIV } a & \text{if } a \geq 1 \text{ & } b = 0 \\
  x & \text{else}
\end{cases}
\end{align*}
\]

○ Our test does not find the defect in the code

Why?
The remedy: term coverage

- Cover not just all branches of a condition, but
- Create test cases such that every individual term makes the condition once true and once false
- In our example: Achieving term coverage for the first if-statement requires three test cases:
  - a=1  b=0  x=1  (first term makes condition false)
  - a=2  b=1  x=1  (second term makes condition false)
  - a=3  b=0  x=3  (both terms make condition true)
  - Achieves term coverage also for second if-statement
In practice: MC/DC

- **MC/DC (Modified condition/decision coverage)** is a term coverage criterion used for safety-critical systems.

- Requires that for every conditional statement, every term in the condition expression has been shown to **determine** the **value** of the condition expression independently:

  Let \( c = t_1 \text{ op } t_2 \text{ op } t_3 \ldots \text{ op } t_{i-1} \text{ op } t_i \text{ op } t_{i+1} \ldots \text{ op } t_n \) be a condition.

  \( c \) needs to become **once true** and **once false** by varying \( t_i \) while keeping all other terms \( t_j \) \( j \neq i \) constant.

- For example, MC/DC is required by the FAA for avionics software.
3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
What is data flow testing?

- A glass-box (structure-oriented) test
- Based on analysis of data flow in a program:
  - Determine the control flow graph
  - Annotate the control flow graph:
    - Where are variables set or modified?
    - Where are variables used in computations?
    - Where are variables used as parts of a condition?
- Various coverage criteria
- Can also be used to assess the quality of a black-box test (in terms of achieved data flow coverage)
Example

A small program in C:

```c
void SortAbs1 (int& z1, int& z2) /* Sorts the set \{|z1|, z2\} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux = z2;
        z2 = z1;
        z1 = aux;
    }
}
```

Annotated control flow graph:

- `def()`: variable set/modified
- `c-use()`: computational use
- `p-use()`: predicative use

- `n_start`: `def(z1)` and `def(z2)`
- `n1`:
  - `p-use(z1)`
  - `p-use(z1)`
  - `c-use(z1)`, `def(z1)`

- `n2`:
  - `c-use(z1)`, `def(z1)`

- `n3`:
  - `p-use(z1)`
  - `p-use(z1)`
  - `p-use(z2)`
  - `c-use(z1)`, `def(z2)`

- `n4`:
  - `c-use(z2)`, `def(aux)`
  - `c-use(z1)`, `def(z2)`
  - `c-use(aux)`, `def(z1)`
  - `c-use(z1)`, `c-use(z2)`

- `n_end`
Variable definitions and uses

- After constructing the control flow graph of a program, we characterize its data flow by annotating the graph:
  - $\text{n.def}(x)$, iff variable $x$ is set or modified in node $n$
  - $\text{n.c-use}(x)$, iff variable $x$ is used in a computation in node $n$
  - $(n,m).\text{p-use}(x)$ iff variable $x$ is used predicatively in a branching condition on edge $(n,m)$

- A path $(n_n, \ldots, n_m)$ in a control flow graph is called definition-clear with respect to variable $x$ iff
  - def$(x)$ in node $n_n$
  - c-use$(x)$ in node $n_m$ or p-use$(x)$ on edge $(n_{m-1}, n_m)$
  - Between the definition of $x$ in $n_n$ and its use in $n_m$ or $(n_{m-1}, n_m)$ there is no other definition of $x$
Test case derivation

Test cases are created such that the program executes definition-clear paths of some coverage class for all variables of the program:

- **all defs-criterion:** For all definitions of x, execute a definition-clear path to at least one use of x
- **all p-uses-criterion:** For all definitions of x, execute a definition-clear path to all predicative uses of x
- **all c-uses-criterion:** For all definitions of x, execute a definition-clear path to all computational uses of x
Test case derivation – Example 1: all-defs

```c
void SortAbs1 (int& z1, int& z2) 
/* Sorts the set {lzl, z2} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux= z2;
        z2 = z1;
        z1 = aux;
    }
}
```

**all-defs:**

- \((n_{\text{start}}, n_1, n_3, n_{\text{end}})\)
- \((n_{\text{start}}, n_1, n_2, n_3, n_4, n_{\text{end}})\)
void SortAbs1 (int& z1, int& z2)  
/* Sorts the set {l|z1|, z2} */  
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux = z2;
        z2 = z1;
        z1 = aux;
    }
}

all-p-uses:
(n<sub>start</sub>, n<sub>1</sub>, n<sub>3</sub>, n<sub>end</sub>)
(n<sub>start</sub>, n<sub>1</sub>, n<sub>3</sub>, n<sub>4</sub>, n<sub>end</sub>)
(n<sub>start</sub>, n<sub>1</sub>, n<sub>2</sub>, n<sub>3</sub>, n<sub>4</sub>, n<sub>end</sub>)
(n<sub>start</sub>, n<sub>1</sub>, n<sub>2</sub>, n<sub>3</sub>, n<sub>end</sub>)

Hint: all-p-uses implies branch coverage – why?
Test case derivation – Example 3: all-c-uses

```c
void SortAbs1 (int& z1, int& z2) {
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux = z2;
        z2 = z1;
        z1 = aux;
    }
}
```

c-use(z1), c-use(z2)
Significance of data flow testing

- In theory very **attractive**
- Derivation of test cases requires considerable **effort**
- Supported by few tools only
- **Low significance** in today’s practice of **testing**
- Data flow analysis is significant as an **automated static analysis technique**
3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
The notion of use-case-based testing

- Defining test cases based on a use case model
- Belongs to the family of black-box (function-oriented) tests
- Goal: Cover all use cases
- Per use case
  - At least one test case for the normal course
  - At least one test case per alternate or exceptional course
- Dependencies between use cases should also be considered
- Suitable particularly for acceptance testing
Exercise: determining test cases

Create test cases for this use case:

**Borrow Book**

**Actor(s):** Library user

**Trigger:** A library user brings one or more books s/he wants to borrow to the check-out station

**Normal course:**
1. Read and validate user’s library card
2. Scan book id and identify corresponding book record in database
3. Record the book to be borrowed and deactivate anti-theft label
4. If library user wants to borrow more than book, repeat steps 2 & 3
5. Print borrow slip for all books just borrowed
6. Hand over books to library user and terminate
Exercise: determining test cases – 2

Alternative courses:
1.1 No library card or scanned card is invalid: cancel transaction
2.1 Book has been reserved for another user: set book aside and proceed with step 4
2.2 Library user has overdue books to be returned: cancel transaction
3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
The problem of combinatorial explosion

- Problem: Programs having numerous options of combining input data values
- Principally, all possible combinations should be tested
- Number of test cases required grows exponentially: not feasible
The pragmatic solution: pairwise testing

- **Empirical observation**: most errors due to input data combination errors can be detected when testing all possible pairs

- The number of test cases required for exhaustive pairwise testing grows logarithmically only

  \[
  n = O(m^2 \log_2 k)
  \]

  \(k\) Number of input fields

  \(m\) Number of possible values per input field

  \(n\) Required number of test cases for pairwise testing

- Testable also for rather large input data sets
Example

- 13 input fields (k=13) with three values each (m=3)
- Testing **all combinations** requires $3^{13} = 1,594,323$ test cases
- For a **full pairwise test**, 15 test cases suffice

Algorithmically computed combination table for full pairwise test (k=13, m=3) [Cohen et al. 1997]

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
<th>P10</th>
<th>P11</th>
<th>P12</th>
<th>P13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Derivation of test cases

❍ There is no simple way of computing the minimum number of test cases manually

❍ Cohen et al. (1997) provide an algorithm

❍ Pairwise testing requires a tool for determining the required combinations of test cases

❍ Commercial testing tools typically include a generator for producing test cases for a full pairwise test automatically

❍ There is also a free Perl script for determining all pairs [Bach 2006]
Example: Testing a credit card payment app

<table>
<thead>
<tr>
<th>Credit card payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card type:*</td>
</tr>
<tr>
<td>MasterCard</td>
</tr>
<tr>
<td>Card number:*</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>valid thru:*</td>
</tr>
<tr>
<td>12  2014 MM/JJJJ</td>
</tr>
<tr>
<td>Name on card: *</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>CVC Code:*</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>What is this?</td>
</tr>
</tbody>
</table>

Determine values to be tested for every input field based on finding equivalence classes on the sets of all potential values
Number of test cases

- Assume three equivalence classes per input field
- We have six fields with three test values each
- Testing all combinations requires $3^6 = 729$ test cases
- Pairwise test requires only 15 test cases
- How sensitive is pairwise testing in this example?

Here’s the code for checking the CVC number:

```html
<TD><P><INPUT TYPE="text" NAME="cardCVC" VALUE="" SIZE=6 MAXLENGTH=3>&nbsp; <FONT SIZE="-1" FACE="Helvetica"><A HREF="http://help/view/pk/en//CVC.shtml" TARGET="_blank" title="">What is this?</A></FONT>
</TD>
```
Analysis of test sensitivity

- Testing all possible combinations finds an error:
  For example, this test case fails:
  \{American Express, 1234432156788765, “John Doe”, 12, 2014, 1234\}
  - Mastercard and Visa use a tree digit CVC code, American Express uses four digits
  - However, entering a four digit CVC number is impossible as the programmer did not know about four-digit codes

- Every test case \{American Express, •, •, •,•, 1234\} finds this error ("•" stands for any input value)

⇒ Pairwise testing suffices to find this error
3.1 The Basics of Testing

3.2 Branch Coverage in Glass-Box Testing

3.3 Data flow Testing

3.4 Use-Case-Based Testing

3.5 Pairwise Testing

3.6 Test Automation
Manual vs. automatic testing

- Creative vs. routine tasks in testing
  - Testing strategy
  - Determining test cases
  - Creating testing procedures
  - Setting up testing environment
  - Running tests
  - Test summary and evaluation

- Routine tasks are **easier to automate** than creative ones
- Automation of repeated tasks is **efficient**
Advantages and limitations of test automation

❖ Advantages
  ● Large number of test cases testable
  ● Unloading routine tasks from human testers
  ● Frequent or even continuous regression testing feasible
  ● Improves testing productivity

❖ Limits
  ● No full replacement for manual testing
  ● Strongly dependent on quality of test oracle
  ● Automation makes testing more efficient, not more effective
  ● Efficiency gain must be balanced with creation effort
  ● No means against insufficient time or inexperienced testers
Automating the selection of test cases

- Generating glass-box (structure-oriented) tests
  - Generating test cases that satisfy some given coverage criteria is possible
  - Problem: from where do we get the expected results?

- Generating user interface tests
  - Test cases for testing formal properties such as dead links or non-editable input fields can be generated
Automating the selection of test cases – 2

- **Generating black-box (function-oriented) tests** including a test oracle
  - Requires a formal specification
  - Practical application rather limited
- **Support for test case selection**, for example, computing the tuples required for pairwise testing
Automating the test procedure

Dependent on type of test:

- Unit test
- System test
- Acceptance test

- We need to automate not only test case execution, but also the comparison of observed and expected results
Automation – Unit and integration testing

- Test procedure written as a program:
  - One test method per test case
  - Comparison of observed and expected results is also part of the program
  - A testing framework
    - simplifies programming test cases
    - serves as test environment
    - visualizes results

- Most widely known unit testing framework:
  - JUnit [Gamma und Beck 2000]
  - Meanwhile also for other languages: CppUnit, PyUnit,...
Automation – System test

Problem: *Actors* in the *system context* must be *simulated*

- Technical devices: Technical *test bed* simulating sensors and actuators
- Neighboring systems: *test harness* with drivers and stubs
- Human interaction: *scripting*
Scripting human interaction

- Test automation with scripting works by
  - writing or recording scripts,
  - in scripting languages such as Apple script, Perl, Python, VBScript, ...
  - which then are executed automatically

- Where to script
  - On the presentation layer
    - physical
    - logical
  - On the function layer
Automation on presentation layer

- **Physical**: keys typed, mouse movement, mouse clicks,...
  - Realistic
  - Scripts rather **low level**: e.g., absolute screen coordinates
  - typically **neither readable nor changeable**
  - **highly sensitive** to minimal, even irrelevant changes
  - Comparison of expected and actual results **difficult**

- **Logical**: Select menu item, set radio button,...
  - Simulation of interaction dialog on a more **abstract** layer
  - Scripts are more **stable, easier to read and easier to modify**
Automation on functional layer

- Accessing system functions over
  - Application programmer interfaces (APIs)
  - Web interfaces or browser interfaces
- Does not test the user interface
- Stable, UI-independent test programs and scripts
- Comparison of observed and expected results easy
- APIs, Web interfaces or browser interfaces must exist
- Caution: potential opportunities for attacking a system
Influence of software architecture

The software architecture has a strong influence on the testability of a system on the function layer

- Layered, acyclic system structure (metaphor of layered virtual machines)
- Models and logic, presentation, and control clearly separated (Model-View-Controller pattern)
Automation: acceptance test

- Creating acceptance test cases from requirements
  - Formal specifications allow generation of test cases
  - Semi-formal models allow generating test case frameworks

- Generating acceptance test cases from examples
  For example: Fit [Cunnigham 2002]
  - User describes expected behavior in spreadsheet-like tables
  - Tester writes a “Fixture”, which maps the table to program code
  - Fit executes the test automatically and visualizes the results
Automation example: Fit

User specifies sample cases:

<table>
<thead>
<tr>
<th>Payroll Fixtures, Weekly Compensation</th>
<th>HolidayHours</th>
<th>Wage</th>
<th>Pay()</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardHours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>20</td>
<td>$800</td>
</tr>
<tr>
<td>45</td>
<td>0</td>
<td>20</td>
<td>$950</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>20</td>
<td>$1360</td>
</tr>
</tbody>
</table>

Programmer writes “Fixture”:

```java
public class WeeklyCompensation : ColumnFixture
{
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;

    public Currency Pay()
    {
        WeeklyTimesheet timesheet = new WeeklyTimesheet(StandardHours, HolidayHours);
        return timesheet.CalculatePay(Wage);
    }
}
```

Fit executes tests and visualizes the results for the user:

<table>
<thead>
<tr>
<th>Payroll Fixtures, Weekly Compensation</th>
<th>HolidayHours</th>
<th>Wage</th>
<th>Pay()</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardHours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>20</td>
<td>$800</td>
</tr>
<tr>
<td>45</td>
<td>0</td>
<td>20</td>
<td>$950</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>20</td>
<td>$1360</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>expected</th>
<th>$1360 expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual</td>
<td>$ 1040 actual</td>
</tr>
</tbody>
</table>
Automation of result evaluation: test oracles

- For every automatically executed test case, expected and observed results must be compared. Options:
  - Comparison during program execution
  - Comparison after program execution

- An automated mechanism which compares expected and observed results is called a test oracle

- Challenges
  - Writing a test oracle can be very demanding and difficult, in particular when human behavior is involved
  - Faults in the oracle yield false positive test results
  - Oracles can’t distinguish between significant and accidental discrepancies: leads to false-negative test results
Automation of result evaluation – 2

- **Executable test procedures** required, including **test oracle**
  - Programmed test procedures
  - Testing scripts

- Set-up, execution and evaluation of a test are automatable to a large extent
  - Example: *Cruisecontrol* is a tool for automated unit and integration testing
References


