Recall the V Model of SW Engineering

- We looked at acceptance tests earlier
- Now we need to look at unit tests and integration testing.
- But first a little review . . .
Testing Concepts

- **Testing** is a systematic approach to find and identify faults. It is NOT intended to show that there are no faults. (A successful test finds faults.)
- A **fault** is a coding error that may lead to an erroneous state.
- An **erroneous state** is a run-time state as a result of a fault, that could lead to failure.
- **Failure** occurs when the system fails to operate as required.

Three Approaches to Dealing with Faults

1. **Fault Avoidance** – Very careful development methods to reduce the number of faults introduced in the first place.
2. **Fault Detection** – Debugging, testing, inspections, and other tools to find and remove faults before the product ships.
3. **Fault Tolerance** – Developing SW that continues to operate normally in the face of faults.
More Testing Concepts

TestSuite

TestCase

TestComponent

Correction

TestStub

TestDriver

Failure

ErroneousState

Fault

is revised by

exercises

finds

repairs

is caused by

is caused by

Different Kinds of Tests, and When They are Performed

Test planning occurs early in the process

Unit and integration tests are planned last and performed first. Usability tests are run in parallel.

System tests test the completed system
## A Typical Test Case Specification

<table>
<thead>
<tr>
<th>Test-case identifier</th>
<th>DriveTrain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test location</td>
<td><a href="http://wwwu2.iie.tum.de/TrainSystem/test-cases/test1">http://wwwu2.iie.tum.de/TrainSystem/test-cases/test1</a></td>
</tr>
<tr>
<td>Feature to be tested</td>
<td>Continuous operation of engine for 5 seconds</td>
</tr>
<tr>
<td>Feature Pass/Fail Criteria</td>
<td>The test passes if the train drives for 5 seconds and covers the length of at least two tracks.</td>
</tr>
</tbody>
</table>

### Means of control
1. The `StartTrain()` method is called via a test driver `StartTrain` (contained in the same directory as the `DriveTrain` test).

### Data
2. Direction of trip and duration are read from an input file [http://wwwu2.iie.tum.de/TrainSystem/test-cases/input](http://wwwu2.iie.tum.de/TrainSystem/test-cases/input).
3. If debug is set to TRUE, then the test case will output the system messages “Enter Track n, Exit Track n” for each n, where n is the number of the current track.

### Test Procedure
The test is started by double-clicking the test case at the specified location. The test will run without further intervention until completion. The test should take no more than 7 seconds.

### Special requirements
The test stub `Engine` is needed for the test execution.

*Oracle: A means of determining what the correct results of a test should be.*

---

## Component Inspections

- Careful detailed review of a component by a team of qualified reviewers. (Usually 3 – 5)
- “Components” may include code, documents, diagrams, procedures, or just about anything.
- Usually conducted with the aid of checklists pre-approved by all involved.
- There are several different published approaches to conducting inspections.
Typical Approach to (Code) Inspections I

• Review team may include (experienced) programmers and/or relevant experts.
• Components to be reviewed may be distributed prior to the first meeting.
• First meeting – Author presents and explains the component(s) to be inspected.
• Reviewers separate, and inspect the component(s) individually with aid of checklist(s).

Typical Approach to (Code) Inspections II

• Second meeting – Reviewers present their findings to the author.
  – Author not allowed to speak unless asked a question.
  – No suggestions for corrections or improvement given at this meeting. (Reviewer(s) may meet with author separately for that purpose later.)
• It is important to maintain an attitude of helpfulness, not criticism.
• Optional follow-up meeting may review changes.
Some benefits of inspections

• Can be applied to components that can’t be tested – documents, rarely encountered code.
• Code can be tested in the absence of necessary supporting HW and SW.
• More than one error can be found at once.
• Style and good practices can be inspected.
• Inspections typically find more faults faster.
• However inspections do not replace testing.

The following 3 slides (and some later material) are from Pezze & Young:
Typical Checklists for Java Code

Individual Items Have Detailed Explanations

Ref. Checklist (D1A, page 17).

FEATURE: CLASS DECLARATION: Are the following requirements satisfied?

ITEM: The visibility of the Class is consistent with the design document

Detailed checklist item references:

Description: The visibility and methods exported by a class must correspond to those in the specification, which may be in the form of a UML diagram. If the class specializes another class, methods must be overridden or overridden.

Motivation: Class correspondence of elements of the implementation to elements of the specification facilitates maintenance and reduces integration faults. If significant deviations are needed, e.g., renaming a class or omitting or changing a public method signature, these are design decisions that should be discussed and reflected in the specification document.

Examples: The code implementing the following UML specification of class CompositeFigure should export fields and methods corresponding to the fields of the specification of class CompositeFigure and its ancestor class LineShape. Implementations that use different names for some fields or methods or that do not include methods getArea() and getPerimeter in class CompositeFigure are acceptable if properly documented. Similarly, implementations that export an additional method compare that specializes the default method equals to aid the oracle generation is acceptable.

Lightweight

- getHeight: int
- getWidth: int

CompositeFigure

- getArea: () int
- getPerimeter: () int
Black Box vs. White Box Testing

- Black box tests are developed without knowledge of the internal workings of the component being tested.
  - Acceptance tests are typically black box.
- White box tests are developed based on knowledge of the internal workings of the component being tested. (a.k.a. clear box.)
  - Unit tests may be black box or white box.
- (Gray box tests use knowledge of the algorithms / data structures in use but not the specific code.)
Equivalence Classes / Testing

• It is rarely possible in practice to test all possible values of all input variables.

• An **Equivalence Class** is a range of input values for which the results are expected to be equivalent. (The same errors are expected.)

• **Equivalence Testing** strives to include at least one test case involving every (combination of) equivalence class(es).

---

Example: What equivalence classes can we identify for this method?

class MyGregorianCalendar {
    ...
    public static int getNumDaysInMonth(int month, int year) {...}
    ...}

<table>
<thead>
<tr>
<th>Equivalence class</th>
<th>Value for month input</th>
<th>Value for year input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Months with 31 days, non-leap years</td>
<td>7 (July)</td>
<td>1901</td>
</tr>
<tr>
<td>Months with 31 days, leap years</td>
<td>7 (July)</td>
<td>1904</td>
</tr>
<tr>
<td>Months with 30 days, non-leap years</td>
<td>6 (June)</td>
<td>1901</td>
</tr>
<tr>
<td>Month with 30 days, leap year</td>
<td>6 (June)</td>
<td>1904</td>
</tr>
<tr>
<td>Month with 28 or 29 days, non-leap year</td>
<td>2 (February)</td>
<td>1901</td>
</tr>
<tr>
<td>Month with 28 or 29 days, leap year</td>
<td>2 (February)</td>
<td>1904</td>
</tr>
</tbody>
</table>
Boundary Testing is Related to Equivalence Classes

- **Boundary Testing** recognizes that errors most commonly occur at the boundary of equivalence classes.
- This is particularly true at the boundary between valid and invalid inputs. (Off by 1 errors.)
- Consider array[0], array[1], array[N-1], array[N].
- Also consider extremely large, small, or negative values, & uninitialized or missing values.

Example: What boundary cases can we identify for this method?

```java
class MyGregorianCalendar {
    ...
    public static int getNumDaysInMonth(int month, int year) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>Equivalence class</th>
<th>Value for month input</th>
<th>Value for year input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leap years divisible by 400</td>
<td>2 (February)</td>
<td>2000</td>
</tr>
<tr>
<td>Non-leap years divisible by 100</td>
<td>2 (February)</td>
<td>1900</td>
</tr>
<tr>
<td>Nonpositive invalid months</td>
<td>0</td>
<td>1291</td>
</tr>
<tr>
<td>Positive invalid months</td>
<td>13</td>
<td>1315</td>
</tr>
</tbody>
</table>
Concept: Percent Coverage

- Based on some criteria, the % coverage of a test (suite) can be defined as the amount of that criteria covered / the total amount of that criteria present * 100%. For example:

\[
\text{\% statement coverage} = \frac{\# \text{ lines tested}}{\text{total \# lines}} \times 100
\]

Some criteria are more rigorous than others. For example:

- **Statement testing** – Strives to test every executable line of code.
- **Branch testing** – Strives to test every branch at every decision point, even if some involve no executable statements. (e.g. empty elses.)
- **Condition testing** – Strives to test every condition in both true and false cases. (More rigorous than branch testing for compound logic.)
- **(Path testing)** tests all paths through the flowchart.)
Flowcharts are useful for branch testing.

```java
public static int getNumDaysInMonth(int month, int year) throws MonthOutOf Bounds, YearOutOf Bounds {
    int numDays;
    if (year < 1) {
        throw new YearOutOf Bounds(year);
    }
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
        month == 8 || month == 10 || month == 12) {
        numDays = 31;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        numDays = 30;
    } else if (month == 2) {
        if (isLeapYear(year)) {
            numDays = 29;
        } else {
            numDays = 28;
        }
    } else {
        throw new MonthOutOf Bounds(month);
    }
    return numDays;
}
```
State-based testing strives to test every state transition in a FSM

<table>
<thead>
<tr>
<th>Stimuli</th>
<th>Transition tested</th>
<th>Predicted resulting state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty set</td>
<td>1. Initial transition</td>
<td>MeasureTime</td>
</tr>
<tr>
<td>Press left button</td>
<td>2.</td>
<td>MeasureTime</td>
</tr>
<tr>
<td>Press both buttons simultaneously</td>
<td>3.</td>
<td>SetTime</td>
</tr>
<tr>
<td>Wait 2 minutes</td>
<td>4. Timer</td>
<td>MeasureTime</td>
</tr>
<tr>
<td>Press both buttons simultaneously</td>
<td>3. Put the system in the SetTime state to test the next transition.</td>
<td>SetTime</td>
</tr>
<tr>
<td>Press both buttons simultaneously</td>
<td>5.</td>
<td>SetTime - MeasureTime</td>
</tr>
<tr>
<td>Press both buttons simultaneously</td>
<td>3. Put the system in the SetTime state to test the next transition.</td>
<td>SetTime</td>
</tr>
<tr>
<td>Press left button</td>
<td>6. Loop back into MeasureTime</td>
<td>MeasureTime</td>
</tr>
</tbody>
</table>

Polymorphism complicates the testing of OO Software

(Strategy design pattern)
Code Equivalent for Strategy Example

Original Code:
```java
public class NetworkConnection {
    private NetworkInterface nif;
    void send(Data msg[]) { 
        queue.concat(msg);
    } 
    if (nif.isReady()) {
        nif.send(queue); 
    }
}
```

Equivalent Code, with polymorphism expanded:
```java
public class NetworkConnection {
    private NetworkInterface nif;
    void send(Data msg[]) { 
        queue.concat(msg);
    } 
    boolean ready = false;
    if (nif instanceof Ethernet) {
        Ethernet nif = (Ethernet)nif;
        ready = nif.isReady();
    } else if (nif instanceof WaveLAN) {
        WaveLAN nif = (WaveLAN)nif;
        ready = nif.isReady();
    } else if (nif instanceof UMTS) {
        UMTS nif = (UMTS)nif;
        ready = nif.isReady();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            Ethernet nif = (Ethernet)nif;
            nif.send(queue); 
        } else if (nif instanceof WaveLAN) {
            WaveLAN nif = (WaveLAN)nif;
            nif.send(queue); 
        } else if (nif instanceof UMTS) {
            UMTS nif = (UMTS)nif;
            nif.send(queue); 
        }
        queue.setLength(0);
    }
}
```

Flowchart Equivalent for Strategy Example:
Integration testing requires a methodical approach

• Errors are most likely to occur at interfaces.
• Horizontal integration approaches:
  – Top-down – Requires stubs for lower layers.
  – Bottom-up – Requires drivers for upper layers.
  – Sandwich – Top and bottom, then the middle.
• Vertical integration – Test units together needed for a particular use-case, one by one.
• Big Bang – Test everything at once. (Not good.)

System testing tests the complete system.

• **Functional testing** – Tests functional requirements.
  – May be scenario / use-case based.
• **Usability testing** – Tests the user interface.
• **Performance testing** – Push (past) the limits.
• **Pilot testing** – Done in production environment.
  – Alpha testing is conducted in-house.
  – Beta testing is conducted by external evaluators.
• **Security testing** – May involve “tiger teams”.
And some more testing issues

- **Regression Testing** involves re-testing things that used to work, after other changes are made.
  - Usually a subset of the original tests of the item.
- **Automating testing** supports frequent regression testing.
  - Requires scaffolding to run automated tests, input files, oracles, reports on results, and possibly roll-backs.
  - Often run overnight, over the weekend, or when (major) changes are made to the project repository.

Test planning may require the consideration of dependencies
One Possible Outline of a Test Plan
Report / Test Specifications

**Test Plan**
1. Introduction
2. Relationship to other documents
3. System overview
4. Features to be tested/not to be tested
5. Pass/Fail criteria
6. Approach
7. Suspension and resumption
8. Testing materials (hardware/software requirements)
9. Test cases
10. Testing schedule

**Test Case Specification**
1. Test case specification identifier
2. Test items
3. Input specifications
4. Output specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

---

**Figure 11-27** Outline of a Test Plan.

**Figure 11-28** Outline of a Test Specification.