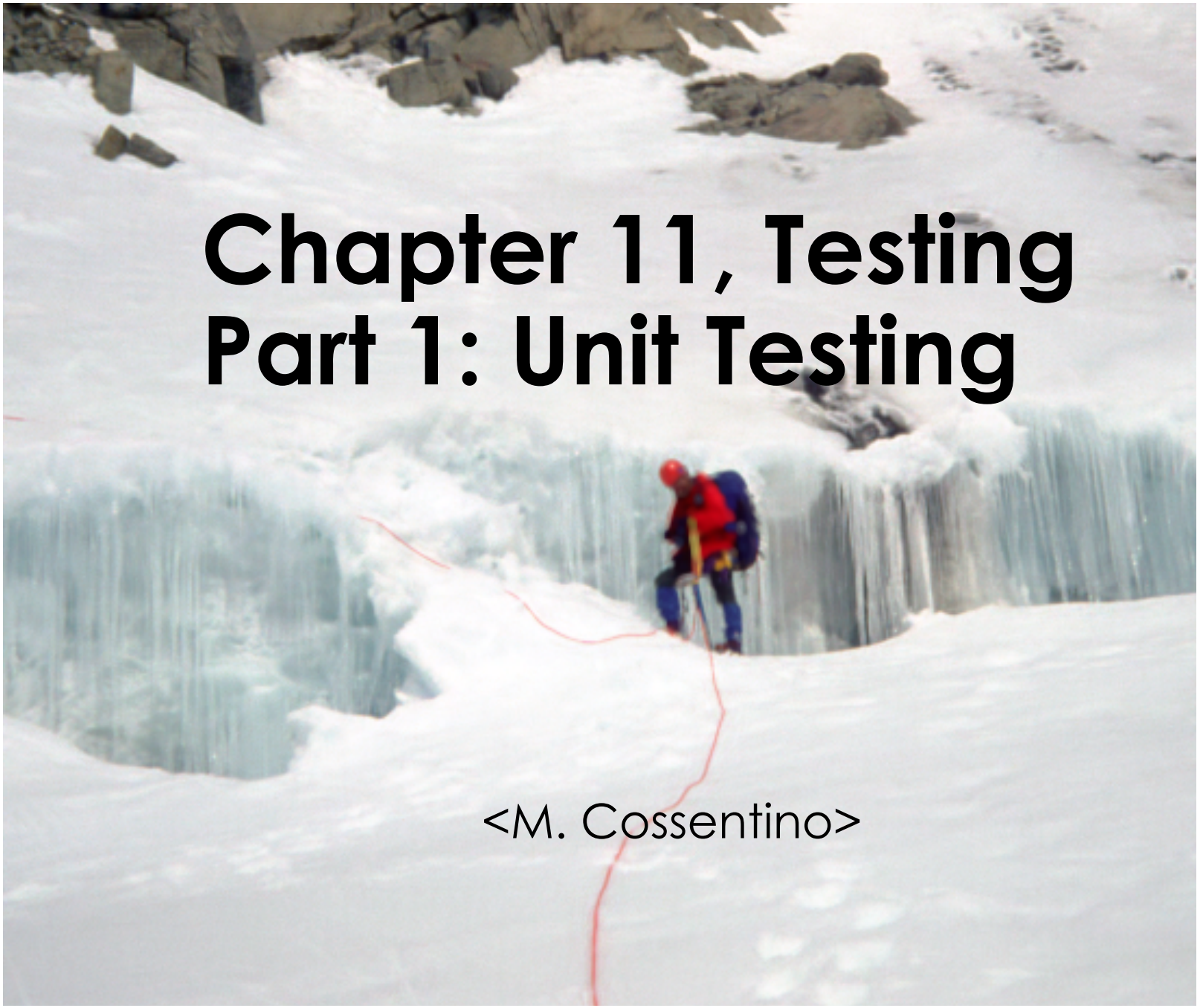


Chapter 11, Testing
Part 1: Unit Testing

<M. Cossentino>

Outline of the Lectures on Testing

- Terminology
  - Failure, Error, Fault
- Test Model
- Model-based testing
- Model-driven testing
- Mock object pattern
- Testing activities
  - Unit testing
  - Integration testing
  - System testing

Famous Problems

- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software from a rocket

- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem
- The Therac-25 accidents (1985-1987), quite possibly the most serious non-military computer-related failure ever in terms of human life (at least five died)
  - Reason: Bad event handling in the GUI,

The Therac-25

- The Therac-25 was a medical linear accelerator
- Linear accelerators create energy beams to destroy tumors
  - Used to give radiation treatments to cancer patients
  - Most of the patients had undergone surgery to remove a tumor and were receiving radiation to remove any leftover growth
- For shallow tissue penetration, electron beams are used
- To reach deeper tissue, the beam is converted into x-rays
- The Therac-25 had two main types of operation, a low energy mode and a high energy mode:
  - In low energy mode, an electronic beam of low radiation (200 rads) is generated
  - In high energy mode the machine generates 25000 rads with 25 million electron volts
- Therac-25 was developed by two companies, AECL from Canada and CGR from France
  - Newest version(reusing code from Therac-6 and Therac-20).

A Therac-25 Accident

- In 1986, a patient went into the clinic to receive his usual low radiation treatment for his shoulder
- The technician typed „X“ (x-ray beam), realizing the error, quickly changed „X" into „E" (electron beam), and hit "enter“:
  - `X <Delete char>  E <enter>`
  - This input sequence in a short time frame (about 8 sec) was never tested
- Therac-25 signaled "beam ready“ and it also showed the technician that it was in low energy mode
- The technician typed „B" to deliver the beam to the patient
  - The beam that actually came from the machine was a blast of 25 000 rads with 25 million electron volts, more than 125 times the regular dose
  - The machine responded with error message “Malfunction 54”, which was not explained in the user manual. Machine showed under dosage.
  - Operator hit “P” to continue for more treatment. Again, the same error message
- The patient felt sharp pains in his back, much different from his usual treatment. He died 3 months later.

Reasons for the Therac-25 Failure

- Failure to properly reuse the old software from Therac-6 and Therac-20 when using it for new machine
- Cryptic warning messages
- End users did not understand the recurring problem (5 patients died)
- Lack of communication between hospital and manufacturer
- The manufacturer did not believe that the machine could fail
- No proper hardware to catch safety glitches.

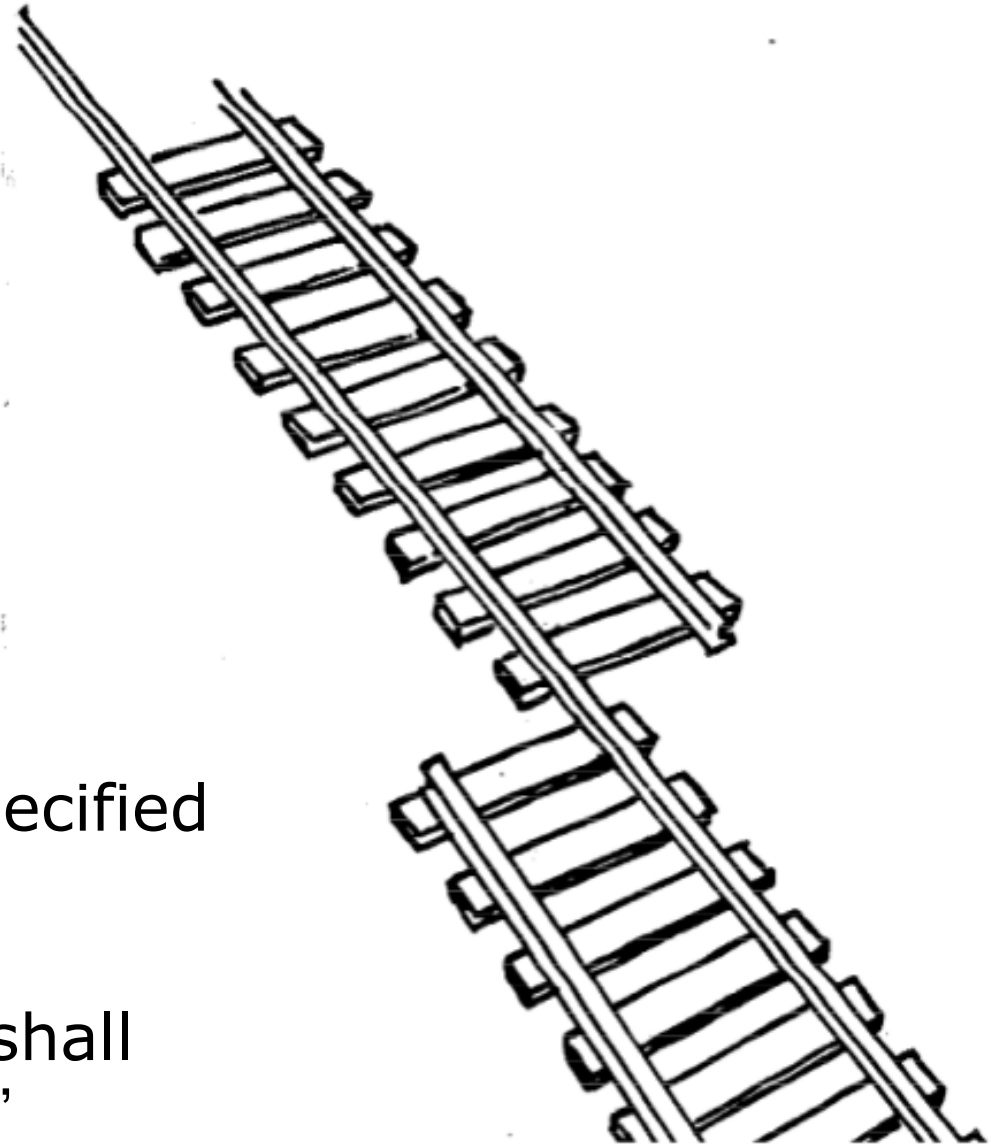
How the Problem was solved

- On February 10, 1987, the Health Protection Branch of the Canadian government along with the FDA (United States Food and Drug Administration) announced the Therac-25 dangerous to use
- On July 21, 1987 recommendations were given by the AECL company on how to repair the Therac-25. Some of these recommendations were
  - Operators cannot restart the machine without re-entering the input command
  - The dose administered to the patient must be clearly shown to the operator
  - Limiting the input modalities to prevent any accidental typos
  - Error messages must be made clearer
  - All manuals must be rewritten to reflect new changes.

Terminology

- Failure: Any deviation of the observed behavior from the specified behavior
- Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure
- Fault: The mechanical or algorithmic cause of an error ("bug")
- Validation: Activity of checking for deviations between the observed behavior of a system and its specification.

What is this?



A failure?

An error?

A fault?

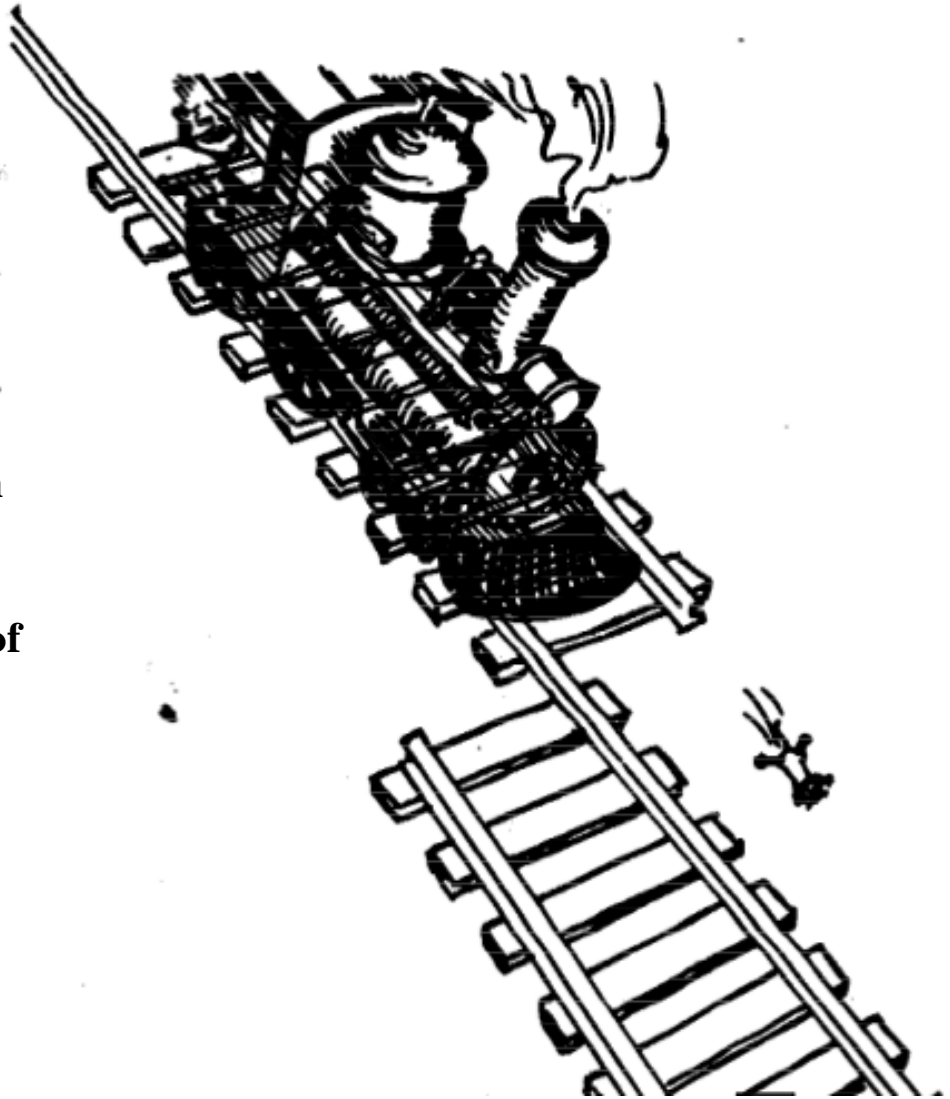
We need to describe specified behavior first!

Specification: “A track shall support a moving train”

Erroneous State ("Error")

Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure

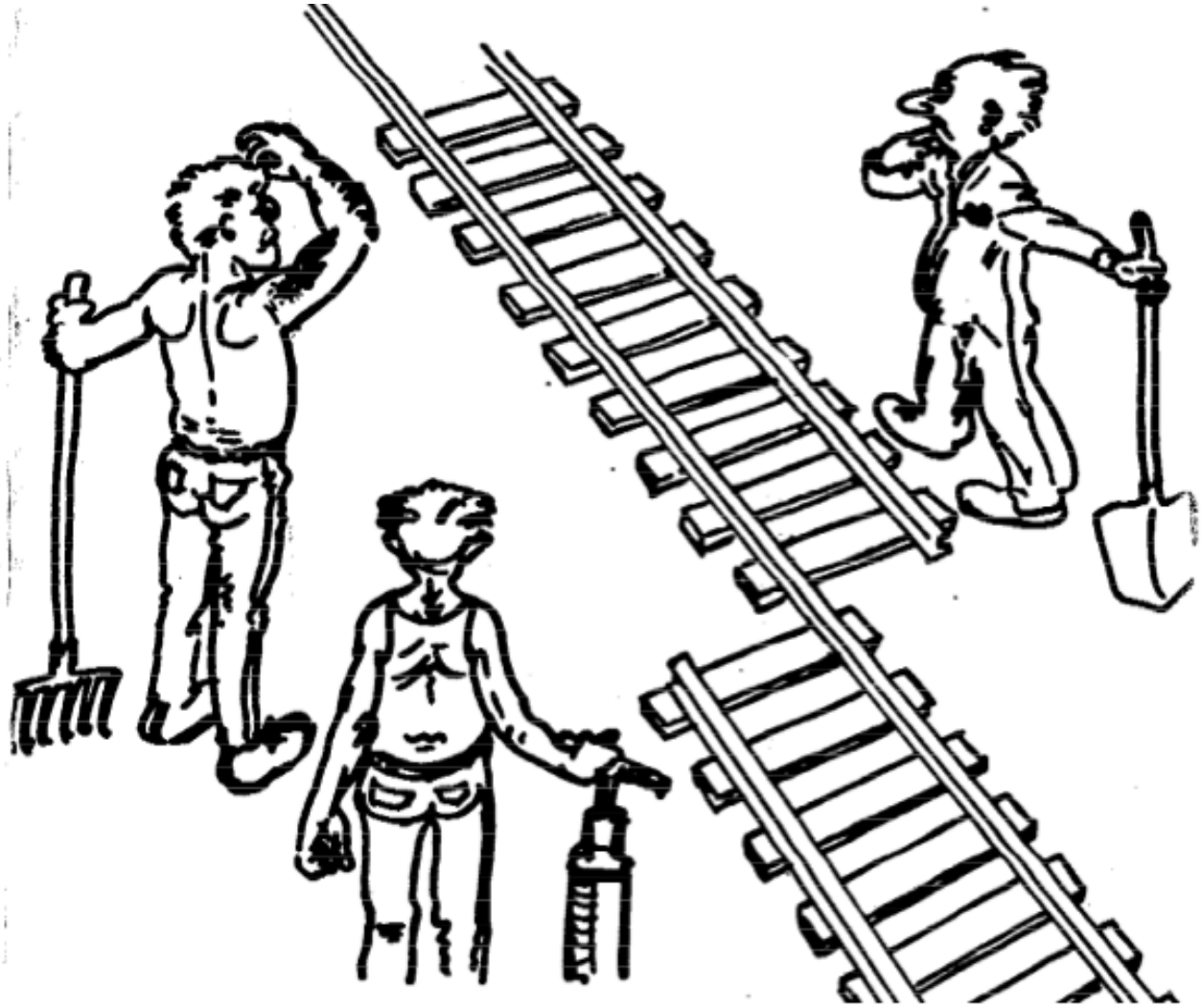
Failure: Any deviation of the observed behavior from the specified behavior



Fault

Possible algorithmic fault: Compass shows wrong reading

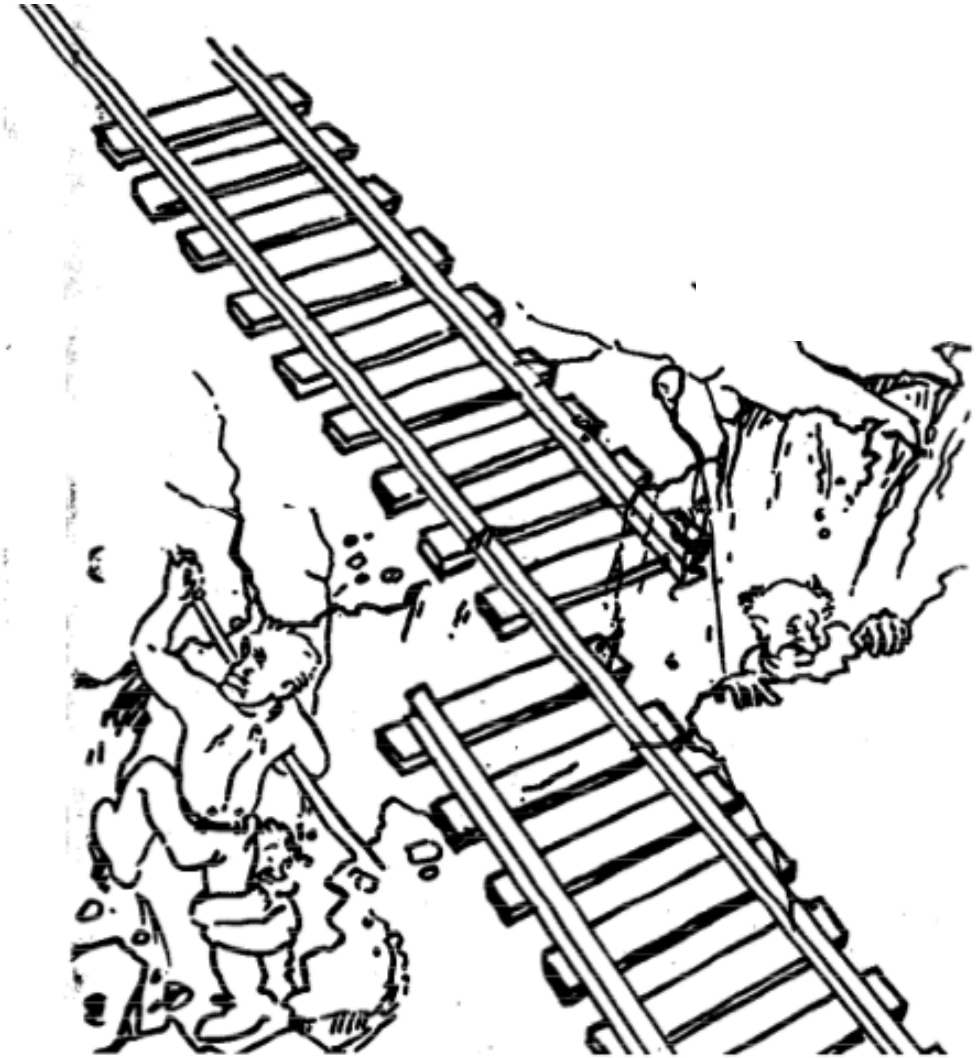
Fault: The mechanical or algorithmic cause of an error ("bug")



Or: Wrong usage of compass

Another possible fault: Communication problems between teams

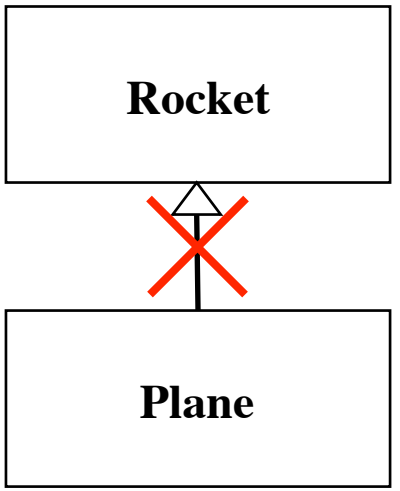
Mechanical Fault



F-16 Bug



- Where is the failure?
- Where is the error?
- What is the fault?
  - Bad use of implementation inheritance
  - A Plane is **not** a rocket.



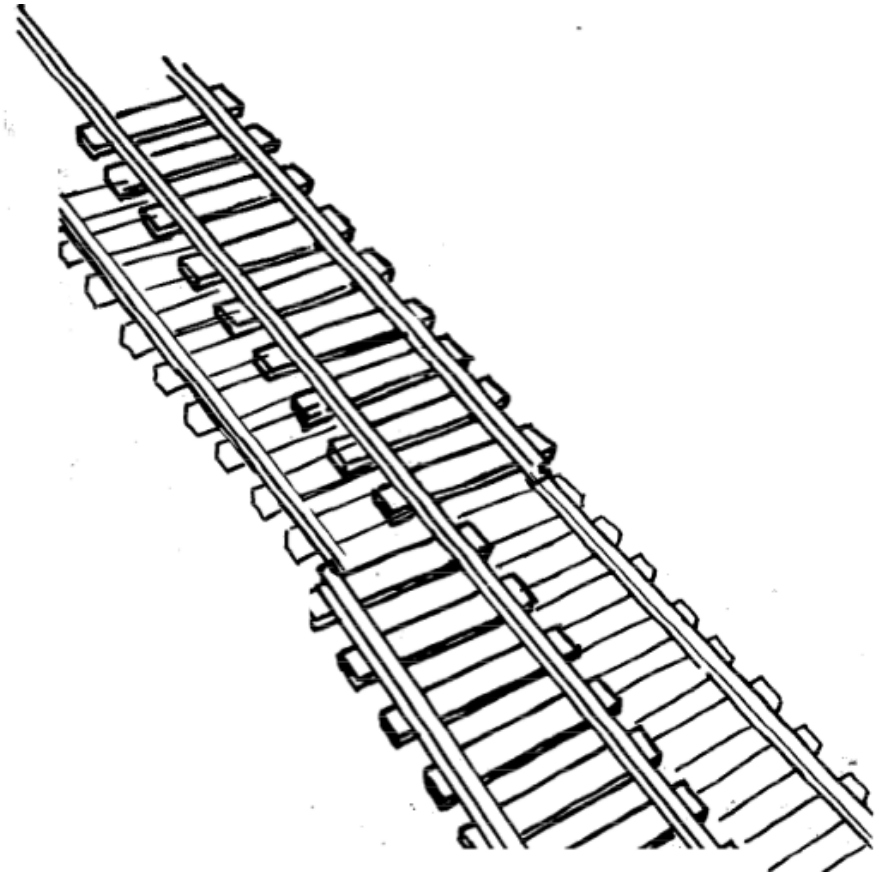
Examples of Faults and Errors

- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null

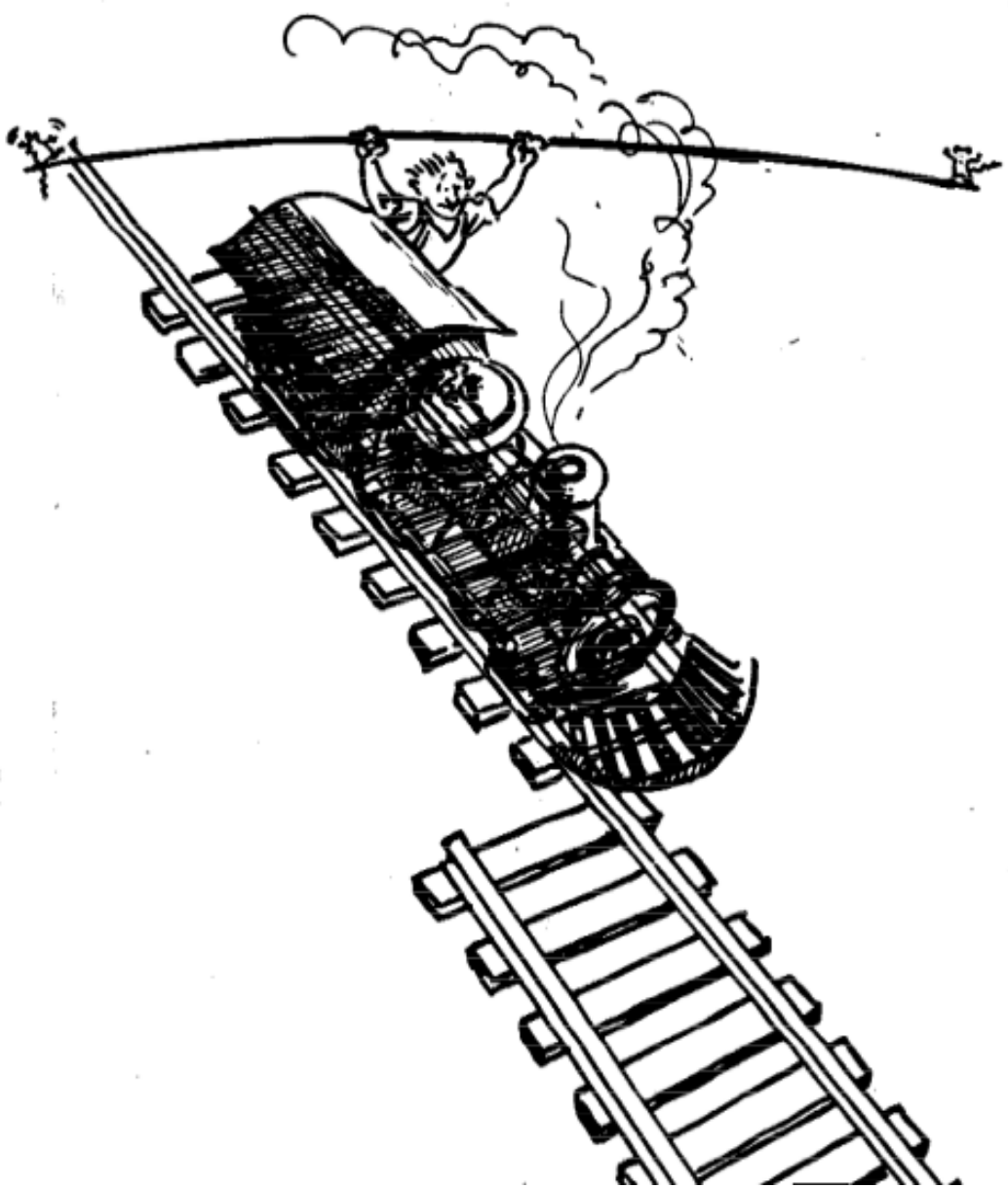
- Mechanical Faults *(very hard to find)*
  - Operating temperature outside of equipment specification
- Errors
  - Wrong user input
  - Null reference errors
  - Concurrency errors
  - Exceptions.

How do we deal with Errors, Failures and Faults?

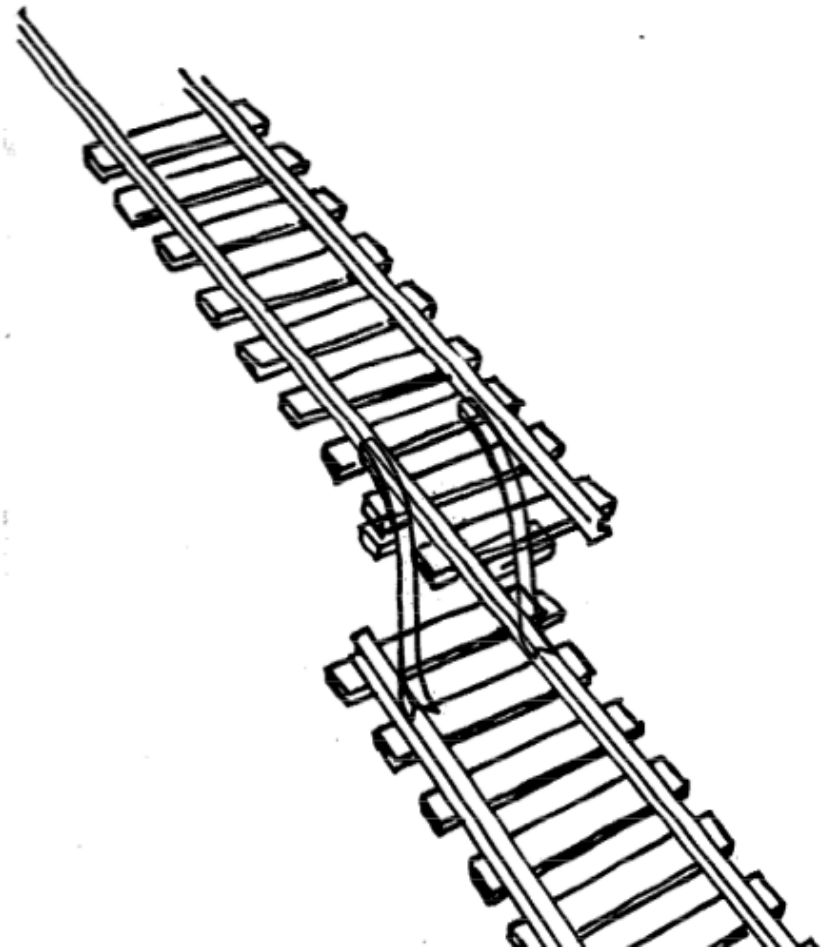
Modular Redundancy



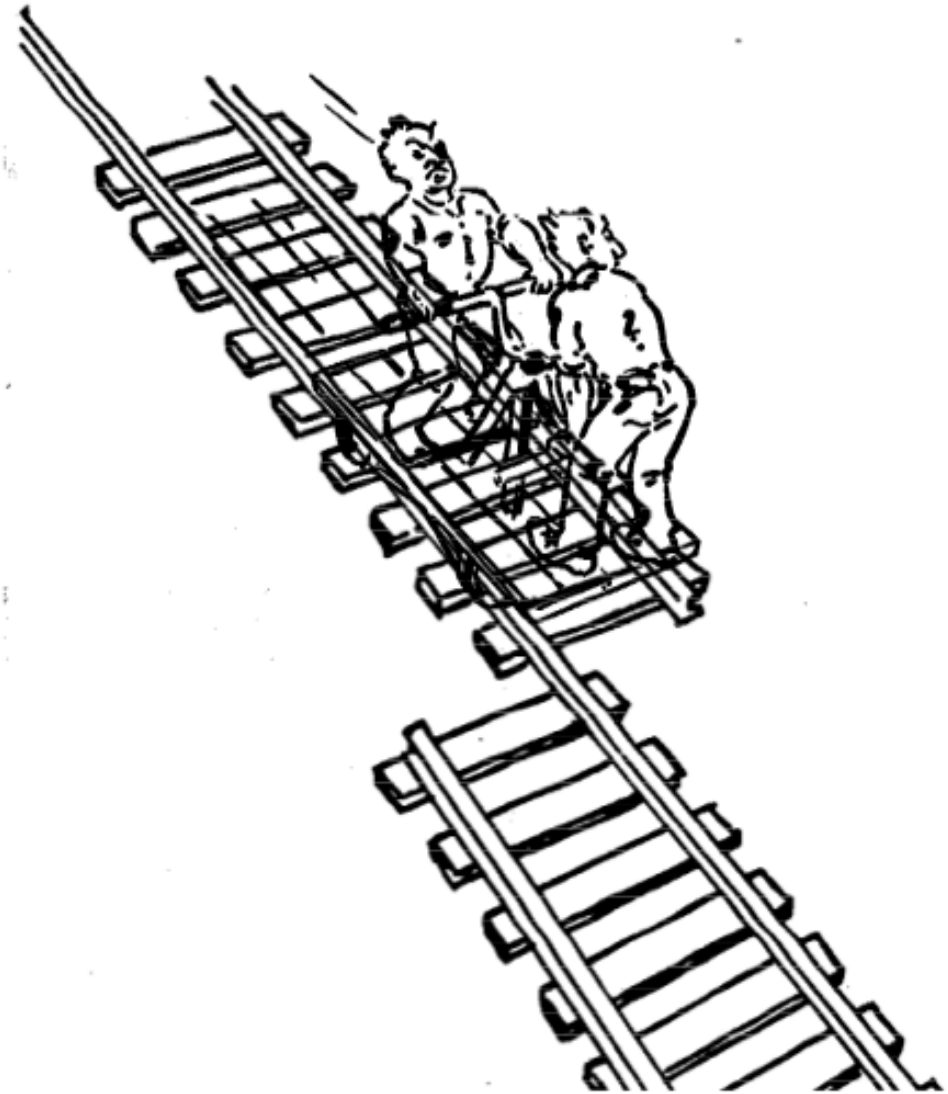
Declaring the Bug as a Feature



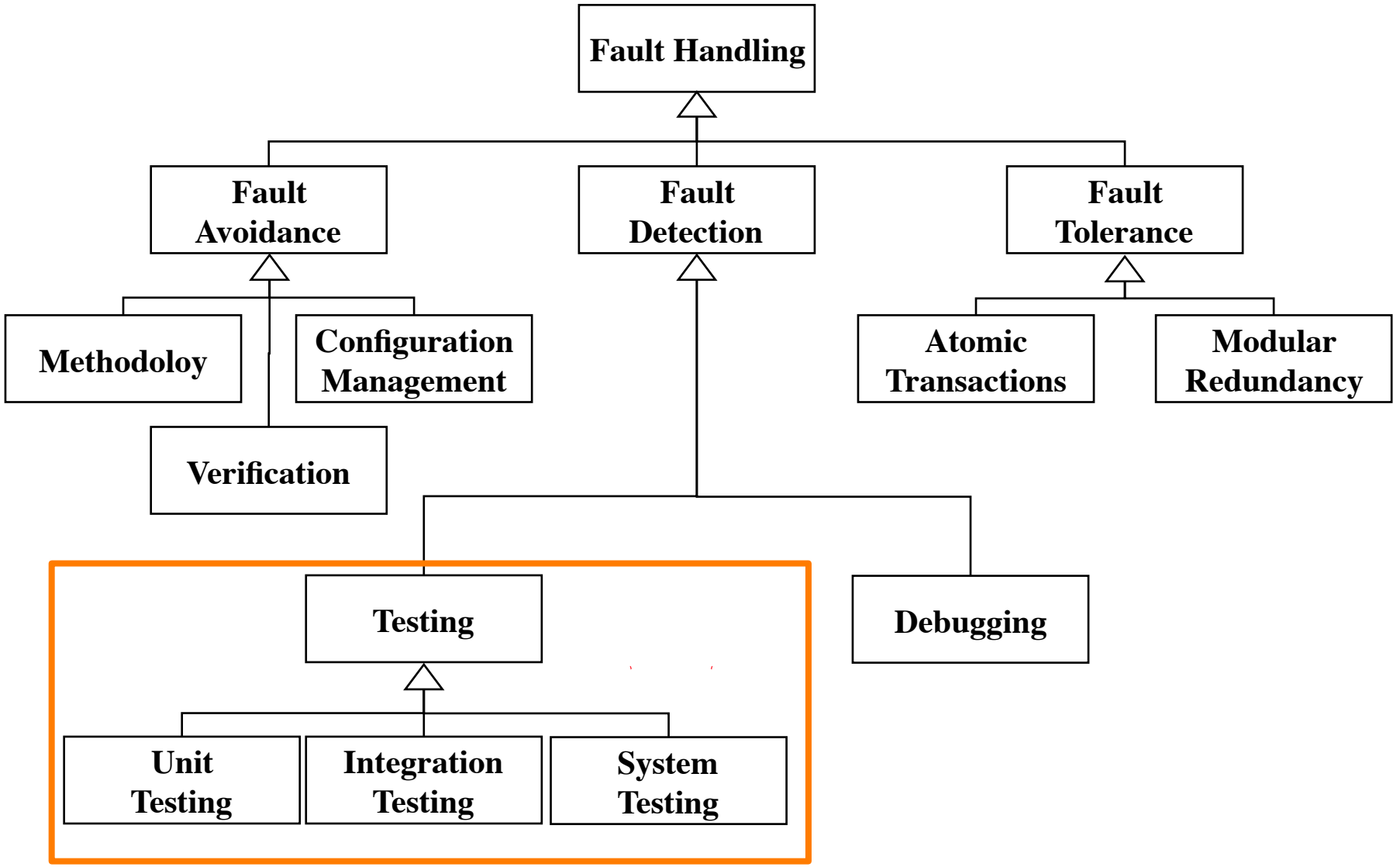
Patching



Testing



Taxonomy for Fault Handling Techniques



Another View on How to Deal with Faults

- Fault avoidance (before the system is released):
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use reviews to identify faults already in the design
- Fault detection (while system is running):
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (fault) of an observed failure
  - Monitoring: Deliver information about state and behavior => Used during debugging
- Fault tolerance (recover from failure once the system is released):
  - Exception handling
  - Modular redundancy.

Observations

- **It is impossible to completely test any nontrivial module or system**
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- **"Testing can only show the presence of bugs, not their absence"** (Dijkstra).
- Testing is not for free

=> Define your goals and priorities

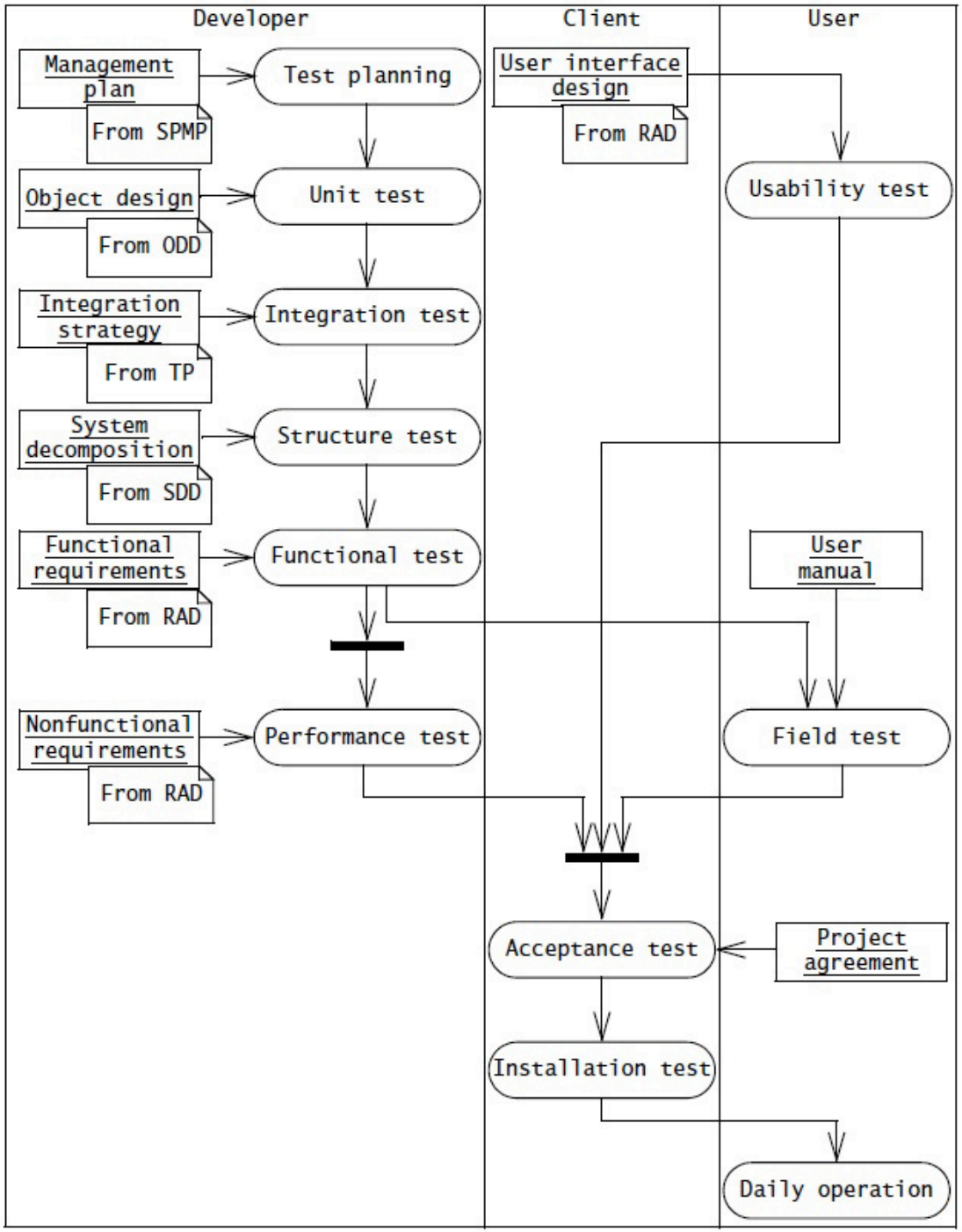
Edsger W. Dijkstra (1930-2002)
- First Algol 60 Compiler
- 1968:
  - T.H.E.
  - Go To considered Harmful, CACM
- Since 1970 Focus on Verification and Foundations of Computer Science
- 1972 A. M. Turing Award

Testing takes creativity

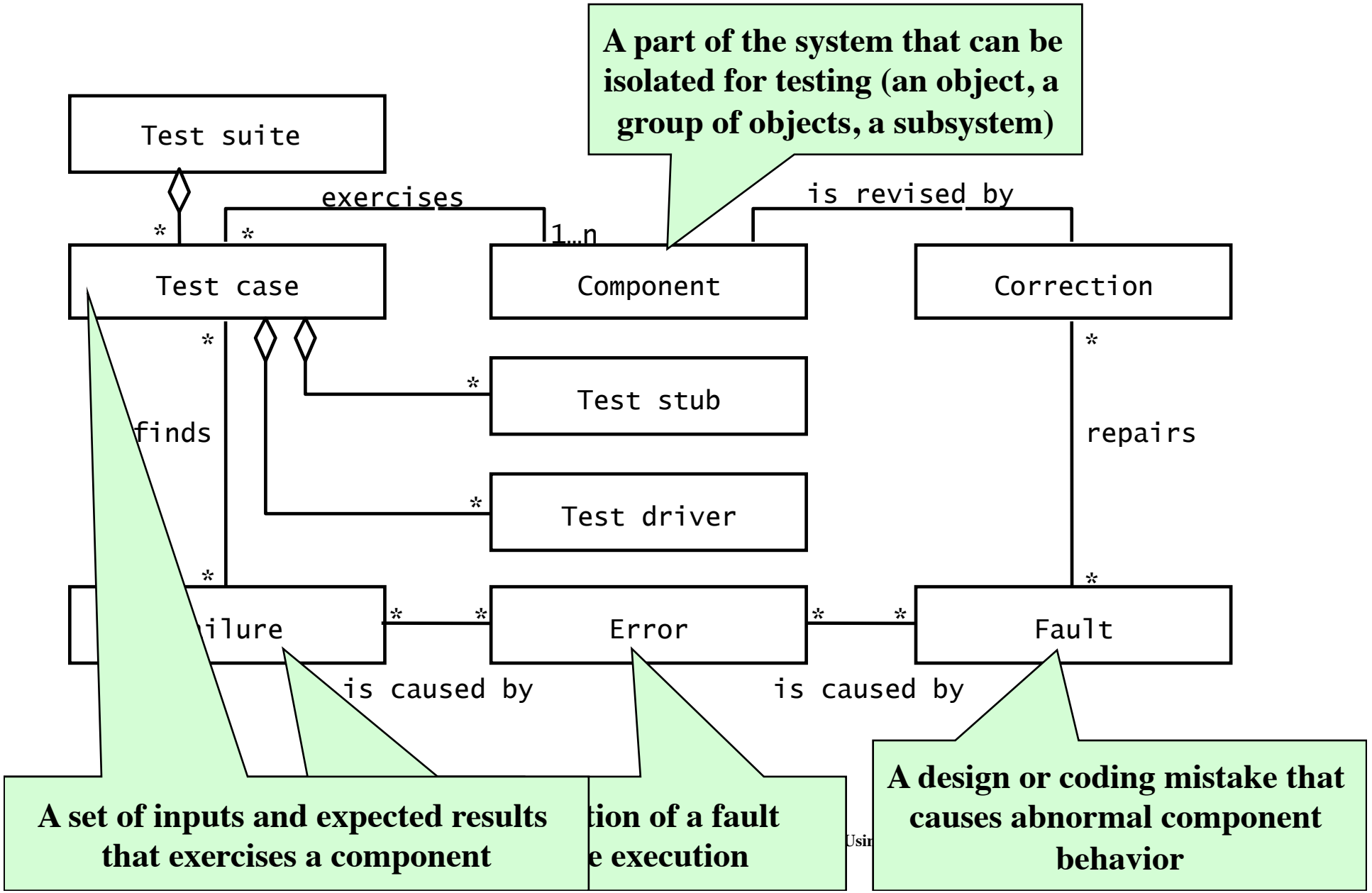
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

A quick overview

Testing activities

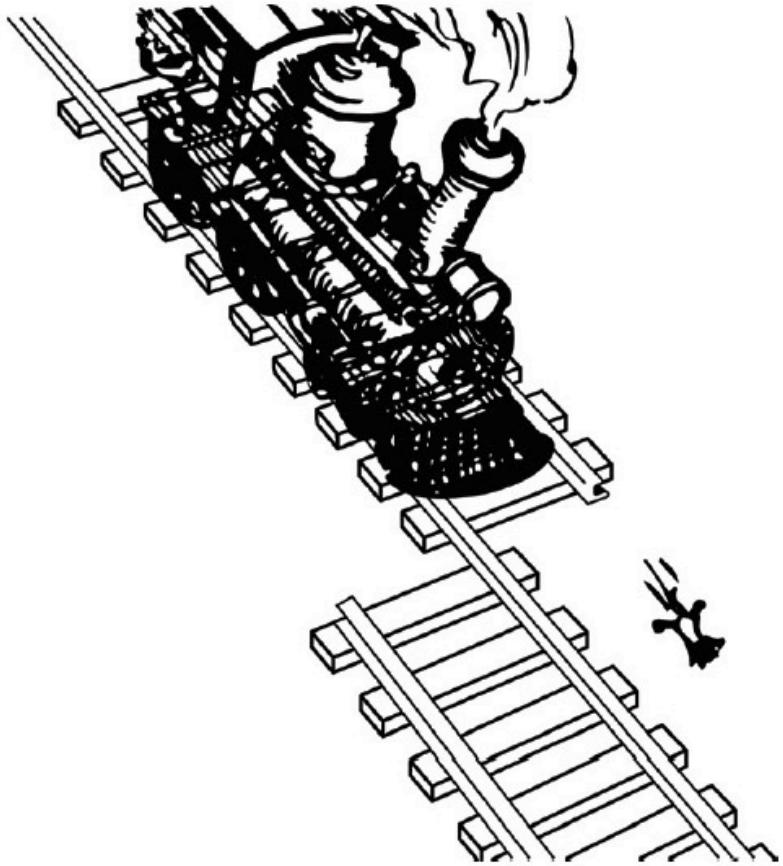


Model elements used during testing



Test Example

- Let’s test the previous presented system



System designed behavior

Use case name	`DriveTrain`
Participating actor	`TrainOperator`
Entry condition	`TrainOperator` pushes the "StartTrain" button at the control panel.
Flow of events	1. The train starts moving on track 1. 2. The train transitions to track 2.
Exit condition	The train is running on track 2.

Figure 11-4 Use case `DriveTrain` specifying the expected behavior of the train.

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

An example of Test Case

Test-case identifier	`DriveTrain`
Test location	`http://www12.in.tum.de/TrainSystem/test-cases/test1`
Feature to be tested	Continuous operation of engine for 5 seconds
Feature Pass/Fail Criteria	The test passes if the train drives for 5 seconds and covers the length of at least two tracks.
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 a input file `http://www12.in.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.

Figure 11-5 Test case `DriveTrain` for the use case described in Figure 11-4.

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

The test case

It is a set of input data and expected results that exercises a component with the purpose of causing failures and detecting faults.

(Most relevant) Attributes of the test case:

- Name
  - it allows the designer to distinguish different test cases
- Location
  - where the test case is located; it could address the pathname or the URL of the executable and input data
- Input
  - the set of input data
- Oracle
  - the expected behavior of the component (the set of output data/ commands that the system should provide)
- Log
  - a set of time-stamped correlations of the *observed* and *expected* behavior (for various test runs)

Model-based testing

Test Model

- The Test Model consolidates all test related decisions and components into one package (sometimes also test package or test requirements)
- The test model contains tests, test driver, input data, oracle and the test harness
  - A test driver (the program executing the test)
  - The input data needed for the tests
  - The oracle comparing the expected output with the actual test output obtained from the test
  - The test harness
    - A framework or software components that allow to run the tests under varying conditions and monitor the behavior and outputs of the system under test (SUT)
    - Test harnesses are necessary for automated testing.

Model-Based Testing

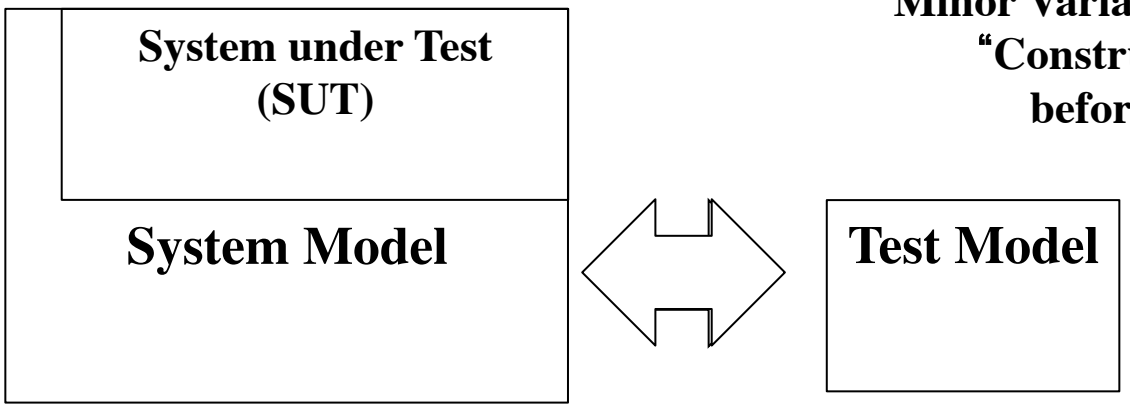
Definition: Model Based Testing

- The system model is used for the generation of the test model

Definition: System under test (SUT)

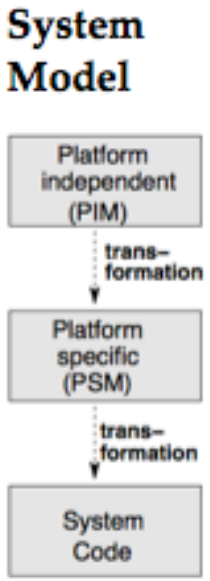
- (Part of) the system model which is being tested

- Advantages of model-based testing:
  - Increased effectiveness of testing
  - Decreased costs, better maintenance
  - Reuse of artifacts such as analysis and design models
  - Traceability of requirements



Minor Variant: Extreme Programming
"Construct the test model first, before the system model"

Model-Driven Testing (MDT)



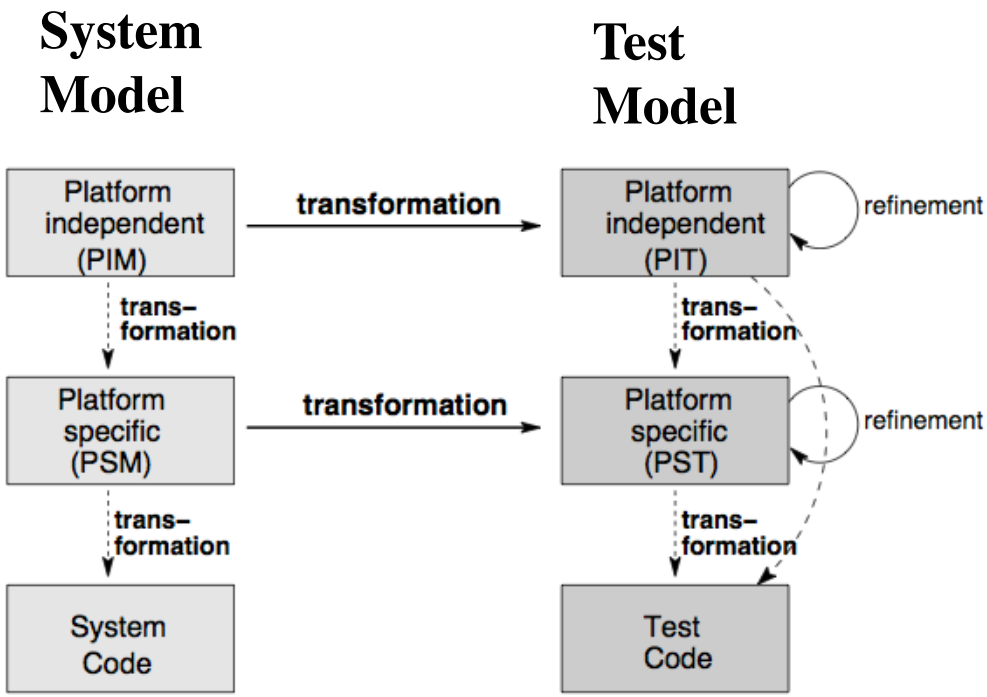
Remember: Model-Driven Architecture (MDA)

- The system model can be separated into a platform independent system model (PIM) and a platform specific system model (PSM)
  - The PIM describes the system independently from the platform that may be used to realize and execute the system
  - The PIM can be transformed into a PSM. PSMs contain information on the underlying platform
  - In another transformation step, the system code is derived from the PSM
- The completeness of the system code depends on the completeness of the system model

- Model-driven testing has its roots in the idea of MDA
- Model-driven testing distinguishes between:
  - Platform independent test models (PIT)
  - Platform specific test models (PST)
  - Test code is generated from these models.

Model-Driven Testing

- System models are transformed into test models
  - When the system model is defined at the PIM level, the platform-independent test model (PIT) can be derived
  - When PSM level is defined, the platform-specific test model (PST) can be derived
    - The PST can also be derived by transforming the PIT model
  - Executable test code is then derived from the PST and PIT models
- After each transformation, the test model may have to be enriched with test specific properties. Examples:
    - If PIT and PST models must cover unexpected system behavior, special exception handling code must be added to the test code
    - Test control and deployment information is usually added at the PST level



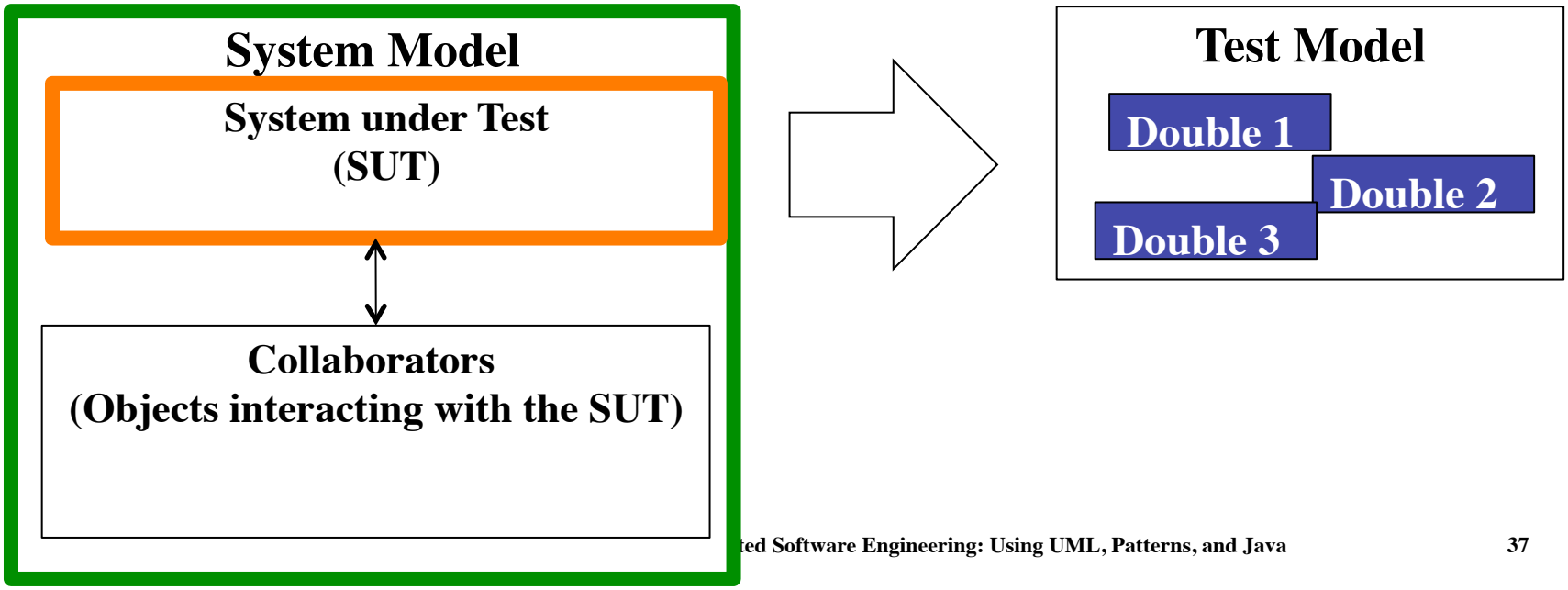
- Model-driven testing enables the early integration of testing into the system development process.

Automated Testing

- There are two ways to generate the test model
  - Manually: The developers set up the test data, run the test and examine the results themselves. Success and/or failure of the test is determined through observation by the developers
  - Automatically: *Automated generation* of test data and test cases. Running the test is also done automatically, and finally the comparison of the result with the oracle is also done automatically
- Definition Automated Testing
  - All the test cases are *automatically executed* with a test harness
- Advantage of automated testing:
  - Less boring for the developer
  - Better test thoroughness
  - Reduces the cost of test execution
  - Indispensible for regression testing.

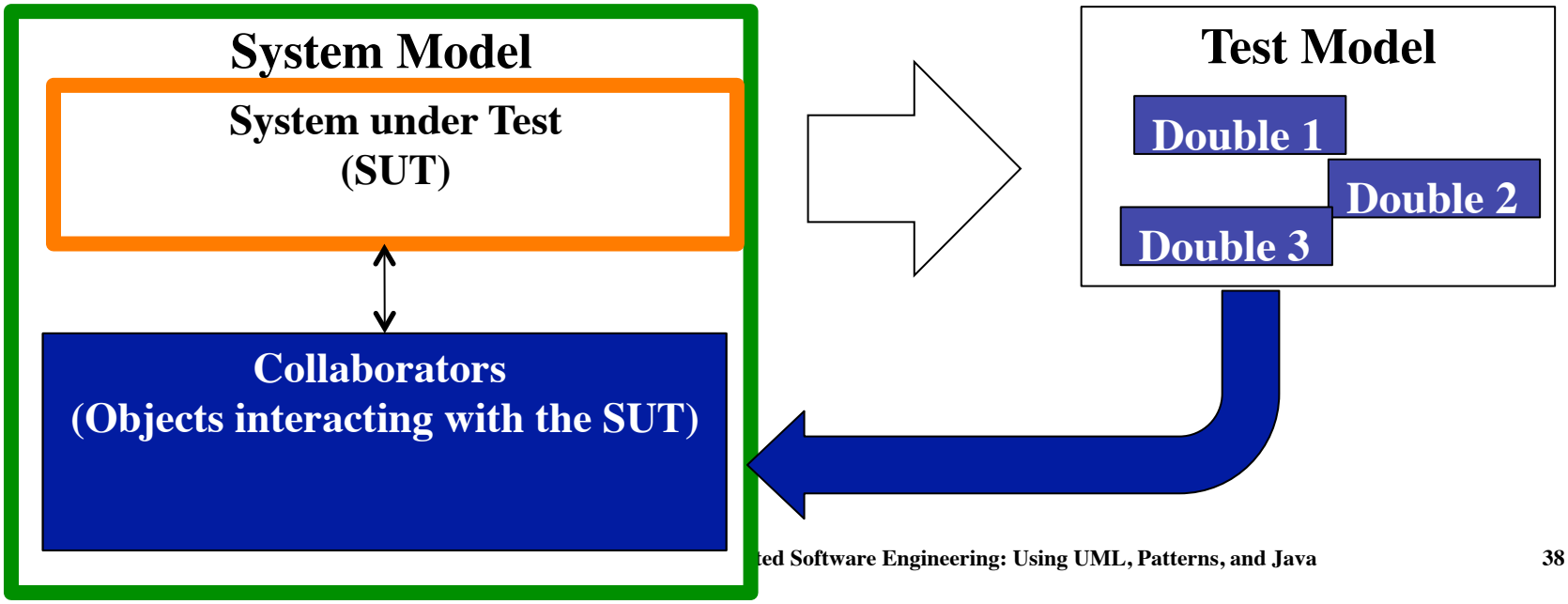
Object-Oriented Test Modeling

- We start with the system model
- The system contains the SUT (the unit we want to test)
- The SUT does not exist in isolation, it collaborates with other objects in the system model
- The test model is derived from the SUT
- To be able to interact with collaborators, we add *objects to the test model*
- These objects are called test doubles



Object-Oriented Test Modeling

- We start with the system model
- The system contains the SUT (the unit we want to test)
- The SUT does not exist in isolation, it collaborates with other objects in the system model
- The test model is derived from the SUT
- To be able to interact with collaborators, we add *objects to the test model*
- These objects are called test doubles
- These doubles are substitutes for the Collaborators during testing



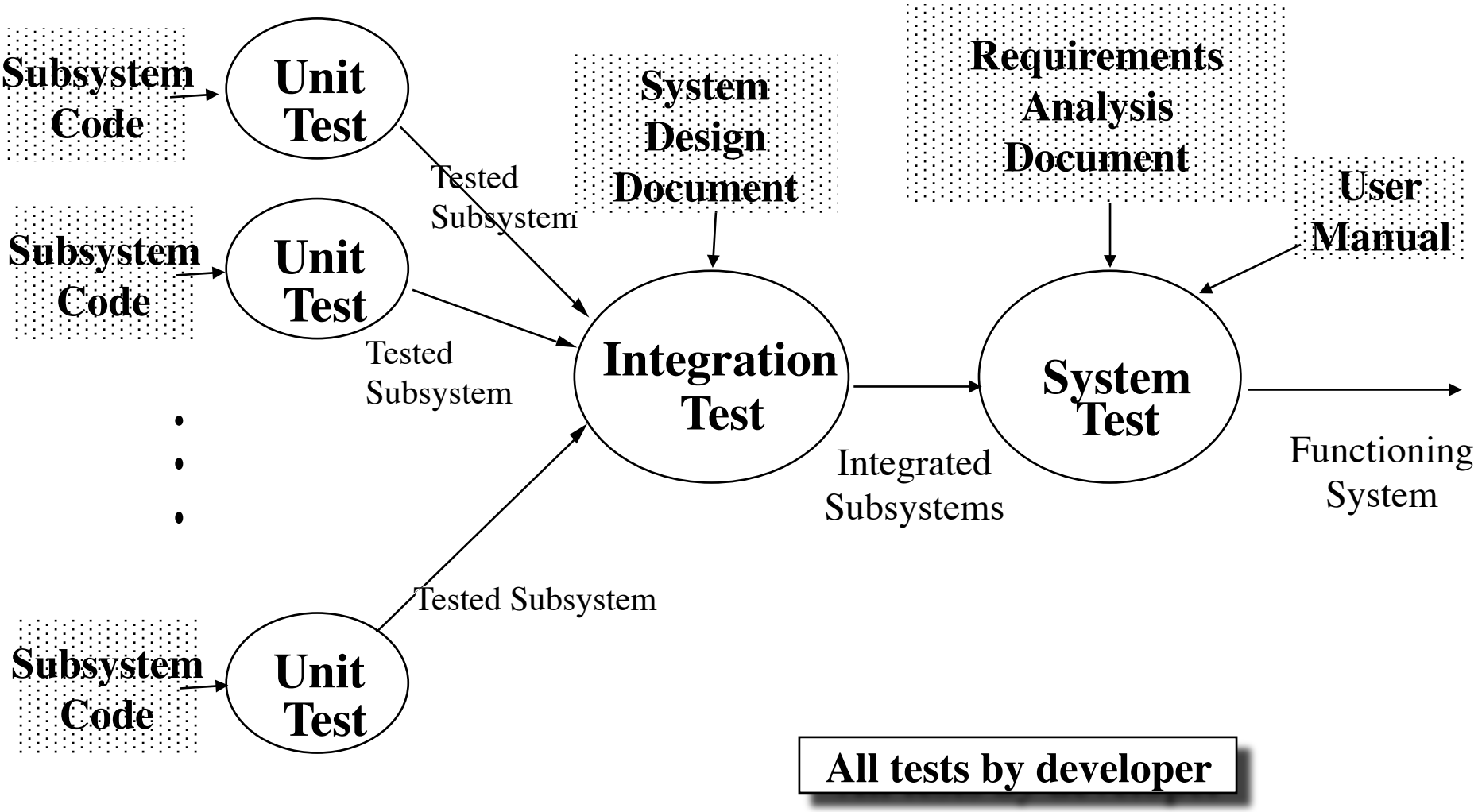
Test Doubles

- A test double is like a double in the movies („stunt double“) replacing the movie actor, whenever it becomes dangerous
- A test double is used if the collaborator in the system model is awkward to work with
- There are 4 types of test doubles. All doubles try to make the SUT believe it is talking with its real collaborators:
  - Dummy object: Passed around but never actually used. Dummy objects are usually used to fill parameter lists
  - Fake object: A fake object is a working implementation, but usually contains some type of “shortcut” which makes it not suitable for production code (Example: A database stored in memory instead of a real database)
  - Stub: Provides canned answers to calls made during the test, but is not able to respond to anything outside what it is programmed for
  - Mock object: Mocks are able to mimic the behavior of the real object. They know how to deal with sequence of calls they are expected to receive.

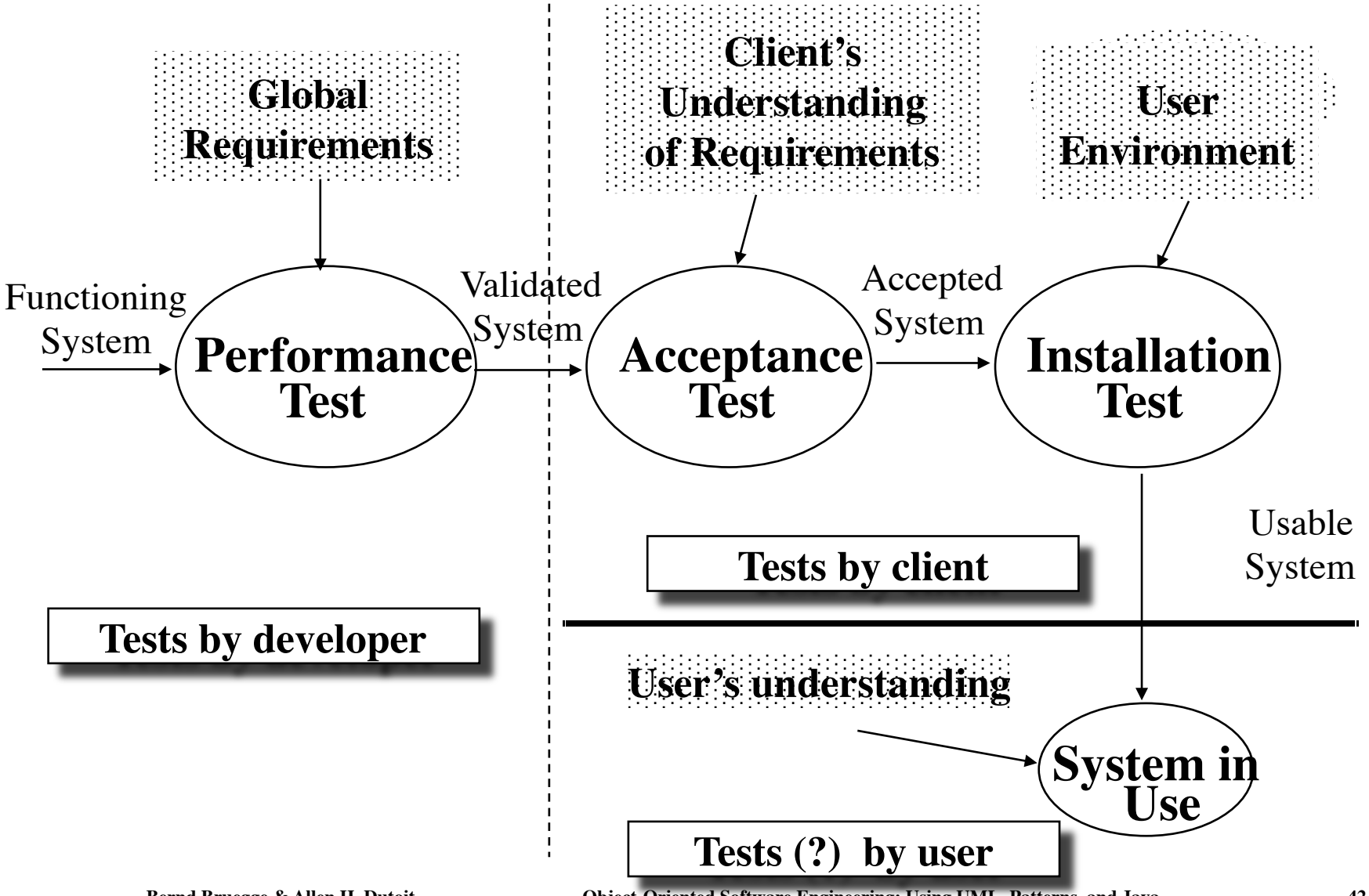
Outline of the Lectures on Testing

- ✓Terminology
  - ✓Failure, Error, Fault
- ✓Test Model
- ✓Model-based testing
- ✓Model-driven testing
- ✓Testing activities
- ✓Mock object pattern
- ➢Testing activities
- Unit testing
- Integration testing
  - Testing strategy
  - Design patterns & testing
- System testing
  - Function testing
  - Acceptance testing.

Testing Activities



Testing Activities ctd



Types of Testing

- Unit Testing
  - Individual components (class or subsystem) are tested
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system are tested
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.
- System Testing
  - The entire system is tested
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

Static Analysis vs Dynamic Analysis

- Static Analysis
  - Hand execution: Reading the source code
  - Walk-Through (informal presentation to others)
  - Code Inspection (formal presentation to others)
  - Automated Tools checking for
    - syntactic and semantic errors
    - departure from coding standards
- Dynamic Analysis
  - Black-box testing (Test the input/output behavior)
  - White-box testing (Test the internal logic of the subsystem or class)
  - Data-structure based testing (Data types determine test cases)

Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the unit passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide inputs into equivalence classes
  - Choose test cases for each equivalence class
    - Example: If an object is supposed to accept a negative number, testing one negative number is enough.

Black box testing: An example

```
public class MyCalendar {

  public int getNumDaysInMonth(int month, int year)
      throws InvalidMonthException
  { … }
}
```

Assume the following representations:

```
Month: (1,2,3,4,5,6,7,8,9,10,11,12)
```
where 1= Jan, 2 = Feb, …, 12 = Dec

```
Year: (1904,…,1999,2000,…,2010)
```

How many test cases do we need to do a full black box unit test of `getNumDaysInMonth()`?

Black box testing: An example

- Depends on calendar. We assume the Gregorian calendar
- Equivalence classes for the `month` parameter
  - Months with 30 days, Months with 31 days, February, Illegal months: 0, 13, -1
- Equivalence classes for the `Year` parameter
  - A normal year
  - Leap years
    - Dividable by /4
    - Dividable by /100
    - Dividable by /400
  - Illegal years: Before 1904, After 2010

How many test cases do we need to do a full black box unit test of `getNumDaysInMonth()`? **12 test cases**

Black-box Testing (Continued)

- Selection of equivalence classes (No rules, only guidelines):
  - Input is valid across range of values. Select test cases from 3 equivalence classes:
    - Below the range
    - Within the range
    - Above the range
  - Input is valid if it is from a discrete set. Select test cases from 2 equivalence classes:
    - Valid discrete value
    - Invalid discrete value
- Another solution to select only a limited amount of test cases:
  - Get knowledge about the inner workings of the unit being tested => white-box testing

White-box Testing

- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once
- Fivetypes of white-box testing
  - Statement Testing
  - Loop Testing
  - Path Testing
  - Branch Testing
  - State-based testing

Unit testing: White-box Testing

- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once.
- Methods of white-box testing
  - Path Testing *(all paths in the program are executed , see next slides)*
  - Statement Testing *(Tests single statements)*
  - Loop Testing *(Focuses on loops: skip, execute once, execute more than once)*
  - Branch Testing *(Each possible outcome from a condition is tested at least once)*
  - State-based testing *(Derives test cases from the state-chart of the class)*

An implementation of `getNumDaysInMonth()` method

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
  public static boolean isLeapYear(int year) {
        boolean leap;
        if (year%4) {
                leap = true;
        } else {
                leap = false;
        }
        return leap;
  }

/* … continued on next slide */
```

1/2

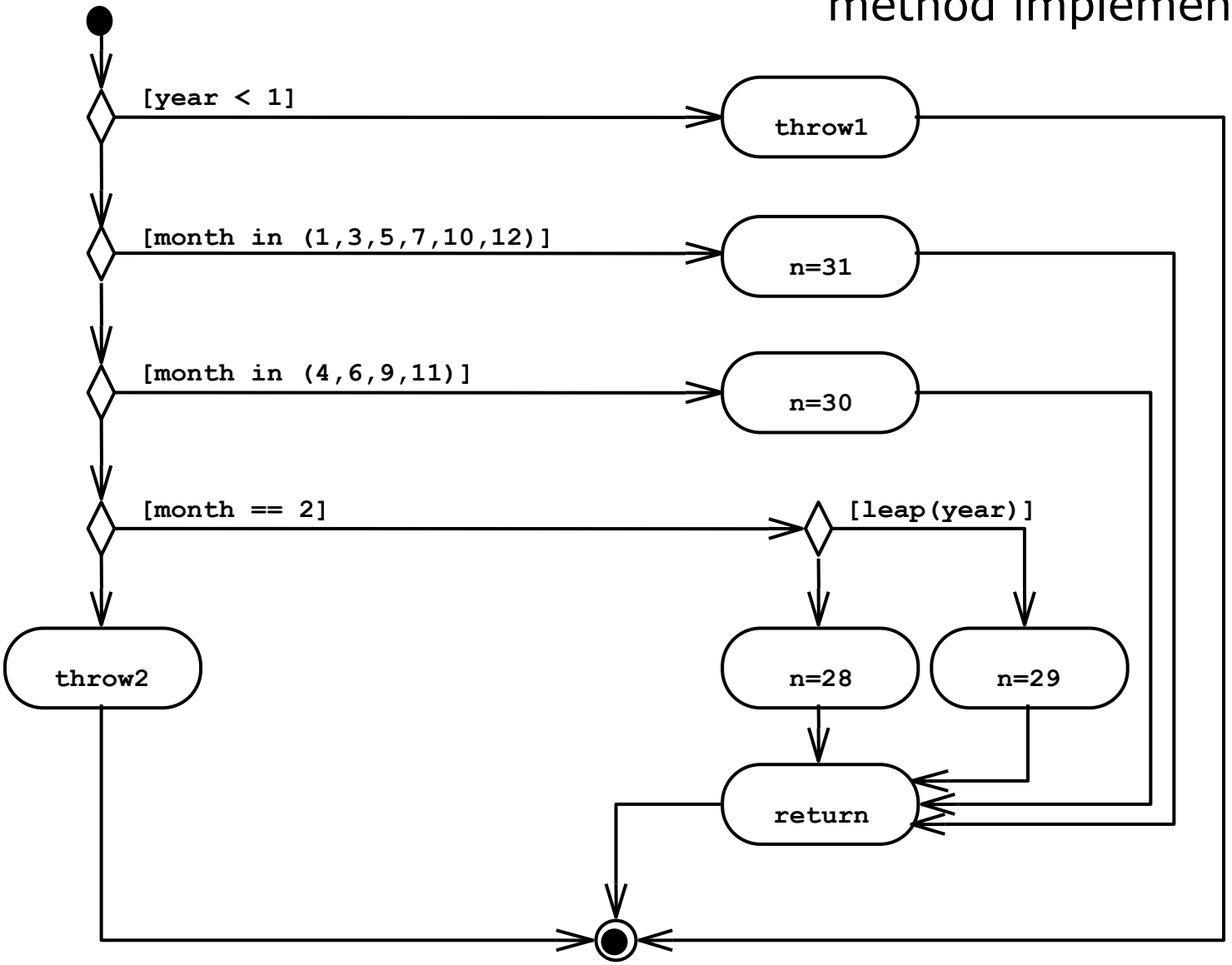
An implementation of `getNumDaysInMonth()` method

```
/* … continued from previous slide */
public static int getNumDaysInMonth(int month, int year)
        throws MonthOutOfBounds, YearOutOfBounds {
  int numDays;
  if (year < 1) {
        throw new YearOutOfBounds(year);
  }
  if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
        numDays = 32;
  } 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 MonthOutOfBounds(month);
  }
  return numDays;
}
```

2/2

Path testing

Equivalent flow graph for the `getNumDaysInMonth()` method implementation



Test cases for the previous flow graph

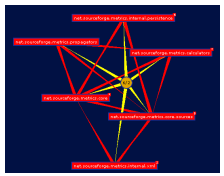
Test Case	Path	Oracle
(year = 0, month = 0)		"Anno fuori range" O "Mese fuori range"
(year = 0, month = 1)	{throw1}	"Anno fuori range"
(year = 1901, month = 1)	{n=32 return}	"Anno fuori range"
(year = 1901, month = 2)	{n=28 return}	"Anno fuori range"
(year = 1904, month = 2)	{n=29 return}	29
(year = 1901, month = 4)	{n=30 return}	"Anno fuori range"
(year = 1904, month = 0)	{throw2}	"Mese fuori range"
(year = 1905, month = 2)		N=28
(year = 1983, month = 5)		N=31
(year = 1983, month = 4)		N=30

Unit testing: white box testing: State based testing

- Introduced for OO programs
- It looks at the state machine of each class
  - The aim is comparing the actual state of the class with the expected one
  - Test cases are derived from the UML statechart of the class
  - For each state a representative set of stimuli is derived for each transition (like in the equivalence testing). Then the variables of the class are observed to verify that the class has reached the specified state

Static Analysis Tools in Eclipse

- Compiler Warnings and Errors
  - *Possibly uninitialized variable*
  - *Undocumented empty block*
  - *Assignment with no effect*
  - *Missing semicolon, …*
- Checkstyle
  - Checks for code guideline violations
  - http://checkstyle.sourceforge.net
- Metrics
  - Checks for structural anomalies
  - http://metrics.sourceforge.net
- FindBugs
  - Uses static analysis to look for bugs in Java code
  - http://findbugs.sourceforge.net
    - Correctness bug: Probable bug - an apparent coding mist ng in code that was probably not what the developer intended. We s low false positive rate.
    - Bad Practice: Violations of recommended and essential coding practice.
    - Dodgy: Code that is confusing, anomalous, or written in a way that leads itself to errors.



FindBugs

- FindBugs is an open source static analysis tool, developed at the University of Maryland
  - Looks for bug patterns, inspired by real problems in real code
- Example: FindBugs is used by Google at socalled „engineering fixit" meetings
- Example from an engineering fixit at May 13-14, 2007
  - Scope: All the Google software written in Java
    - 700 engineers participated by running FindBugs
    - 250 provided 8,000 reviews of 4,000 issues
      - More than 75% of the reviews contained issues that were marked „should fix" or „must fix", „I will fix"
  - Engineers filed more than 1700 bug reports
  - Source: http://findbugs.sourceforge.net/

Observation about Static Analysis

- Static analysis typically finds mistakes but some mistakes don't matter
- Not a magic bullet but if used effectively, static analysis is cheaper than other techniques for catching the same bugs
- Static analysis, at best, catches 5-10% of software quality problems
  - Source: William Pugh, Mistakes that Matter, JavaOne Conference
    - http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf

Comparison of White & Black-box Testing

- White-box Testing
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing
  - Potential combinatorical explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases (unexpected "features")
- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

Unit Testing Heuristics

1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
2. Develop the test cases
   - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Sometimes reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
   - Automate this if possible.

The test case (reminder)

It is a set of input data and expected results that exercises a component with the purpose of causing failures and detecting faults.

Attributes of the test case:

- Name
  - it allows the designer to distinguish different test cases
- Location
  - where the test case is located; it could address the pathname or the URL of the executable and input data
- Input
  - the set of input data
- Oracle
  - the expected behavior of the component (the set of output data/ commands that the system should provide)
- Log
  - a set of time-stamped correlations of the *observed* and *expected* behavior (for various test runs)

When should you write a test?

- Traditionally after the source code is written
- In XP before the source code is written

- Test-Driven Development Cycle
  - Add a new test to the test model
  - Run the automated tests
    => the new test will find a failure
  - Write code to deal with the failure
  - Run the automated tests
    => see them succeed
  - Refactor code.