Introduction to Software Engineering Methods

Testing

Literature used

- Text book

  Chapter 11

Introduction Content

- Terminology
- Types of errors
- Dealing with errors
- Component Testing
  - Unit testing
  - Integration testing
- System testing
  - Function testing
  - Structure Testing
  - Performance testing
  - Acceptance testing
  - Installation testing
- Summary

What is testing?

- Testing is the process of analyzing a system or components to detect the differences between specified (required) and observed (existing) behavior
- Systematic way to find faults in a planned way in the implemented software

Testing

- Testing often viewed as dirty work (that is completely wrong!)
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Knowledge of the testing techniques
  - Skill to apply these techniques in an effective and efficient manner

Testing

- Testing is done best by independent testers
- Programmer often stick to the data set that makes the program work
- A program often does not work when tried by somebody else

Overview of Terminology

- **Reliability:** The measure of success with which the observed behavior of a system confirms to some specification of its behavior.
- **Fault (Bug):** The mechanical or algorithmic cause of an error (a design or coding mistake that may cause abnormal component behavior)
- **Erroneous state (Error):** manifestation of the fault during the execution of the system. 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.

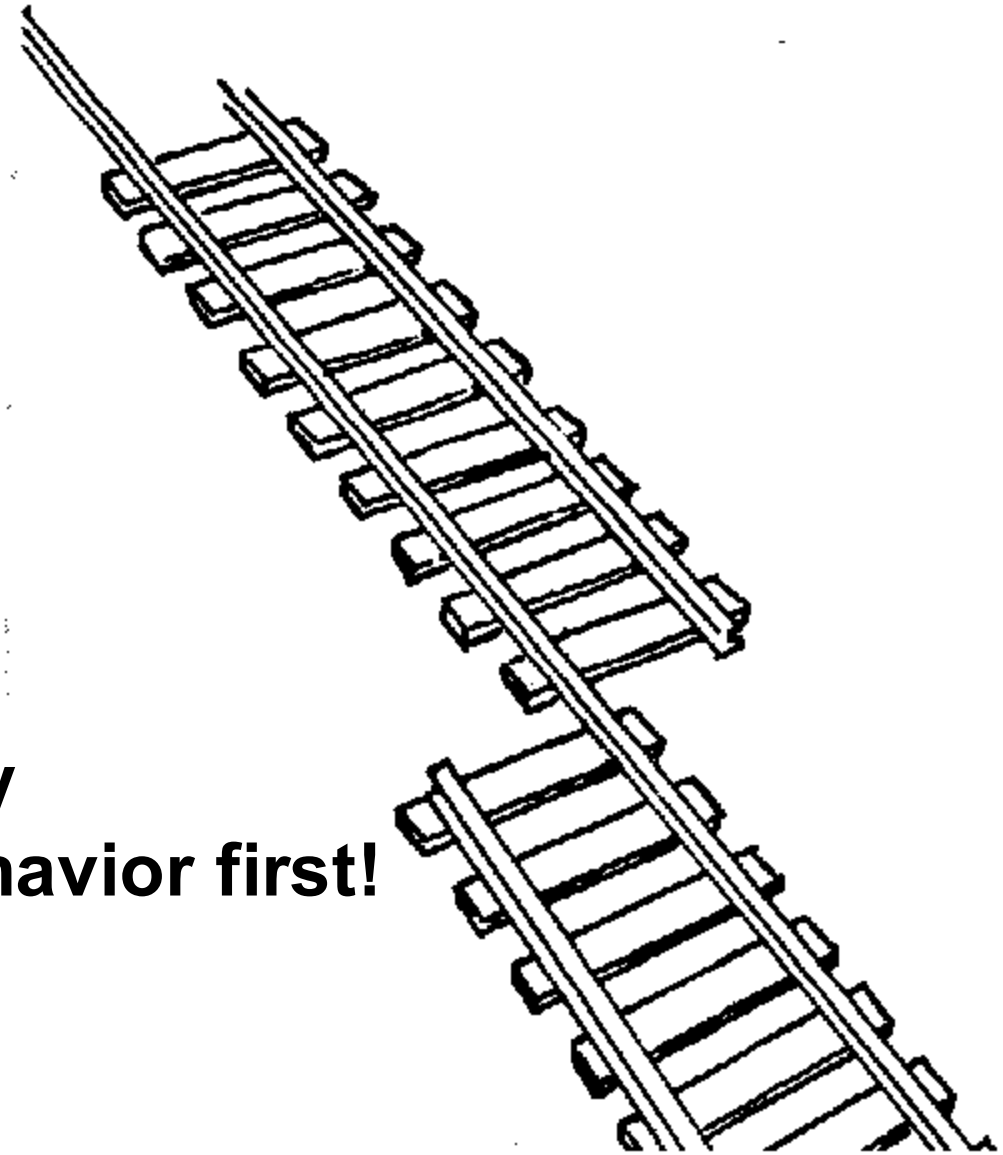
Overview of Terminology continue

- **test case:** a set of inputs and expected results that exercises a component with the purpose of causing failures and detecting faults.

- **test stub:** a partial implementation of components on which the tested component depends.

- **test driver:** a partial implementation of a component that depends on the tested component

What is this?



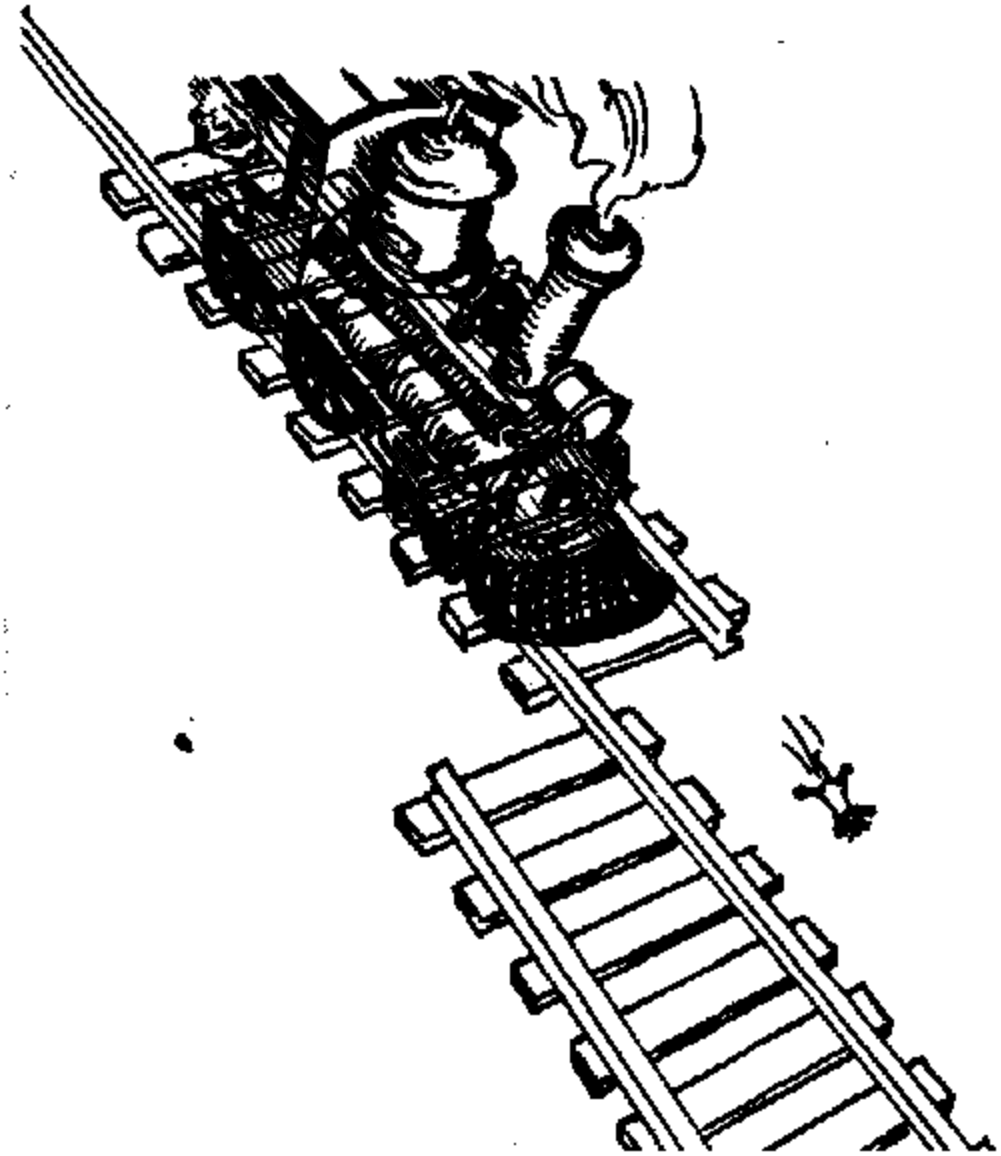
A failure?

An error?

A fault?

Need to specify the desired behavior first!

Erroneous State (“Error”)

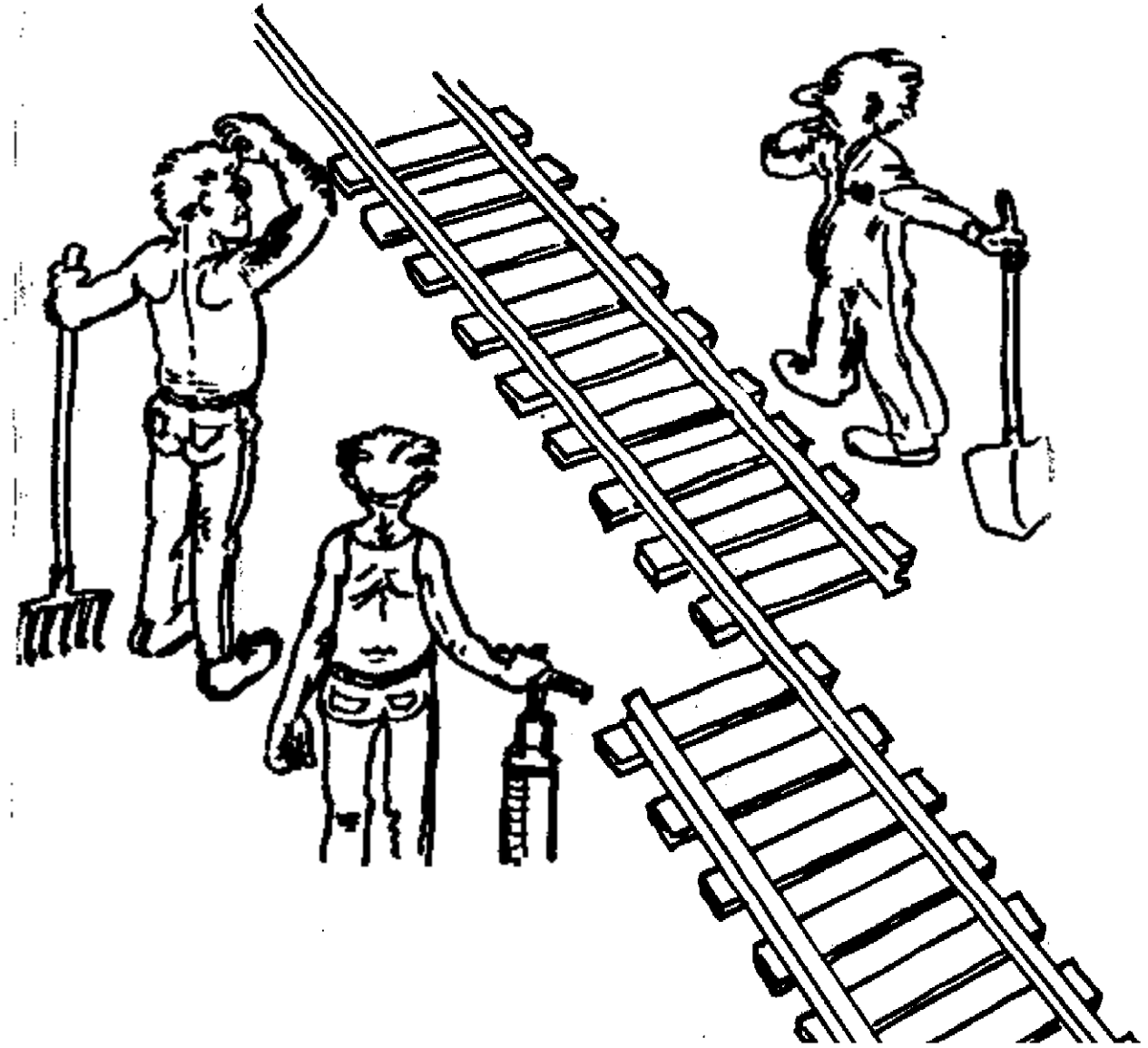


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Types of Faults

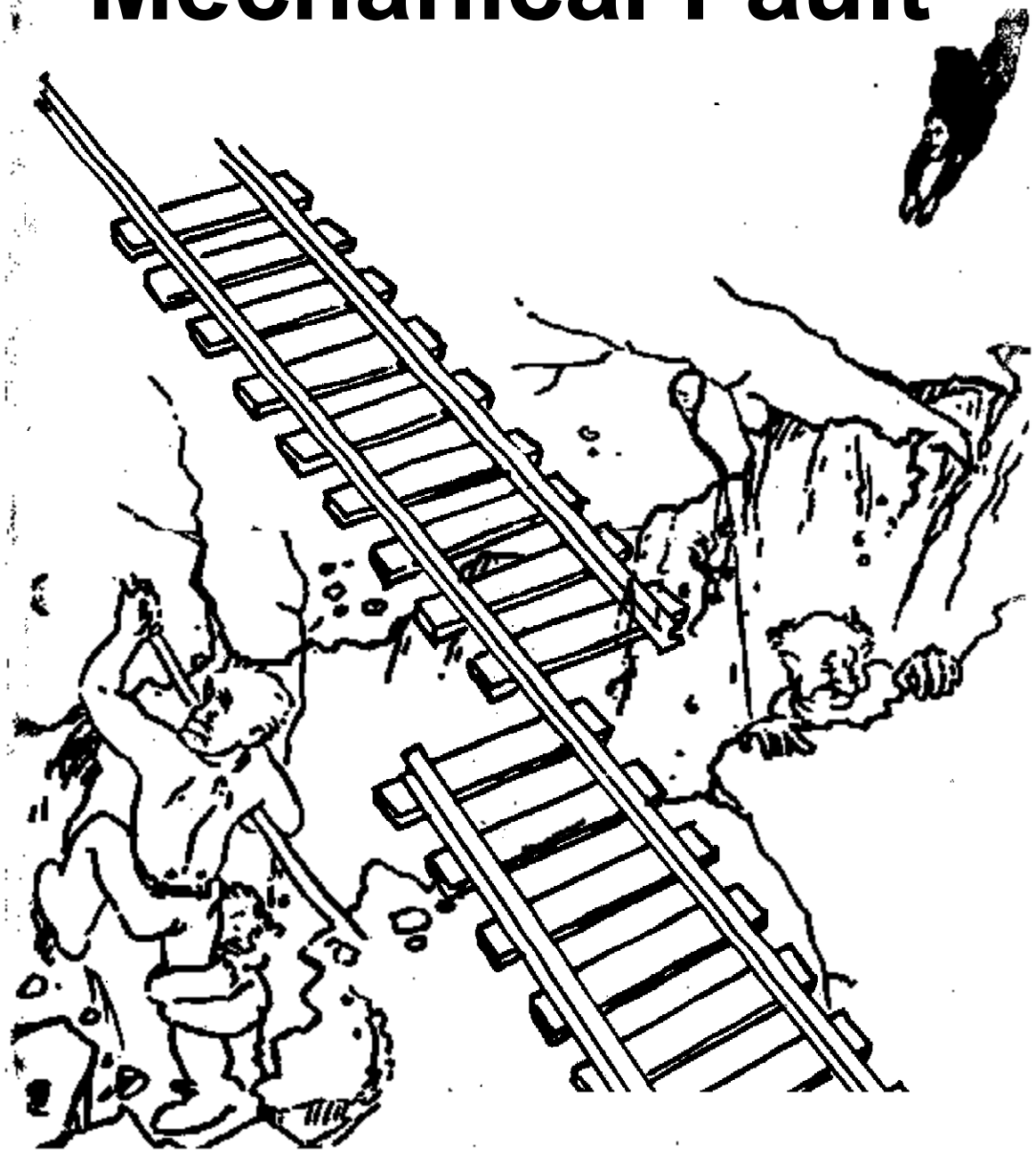
- Faults in the Interface specification
- Algorithmic Faults
- Mechanical Faults (very hard to find)

Algorithmic Fault



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Mechanical Fault



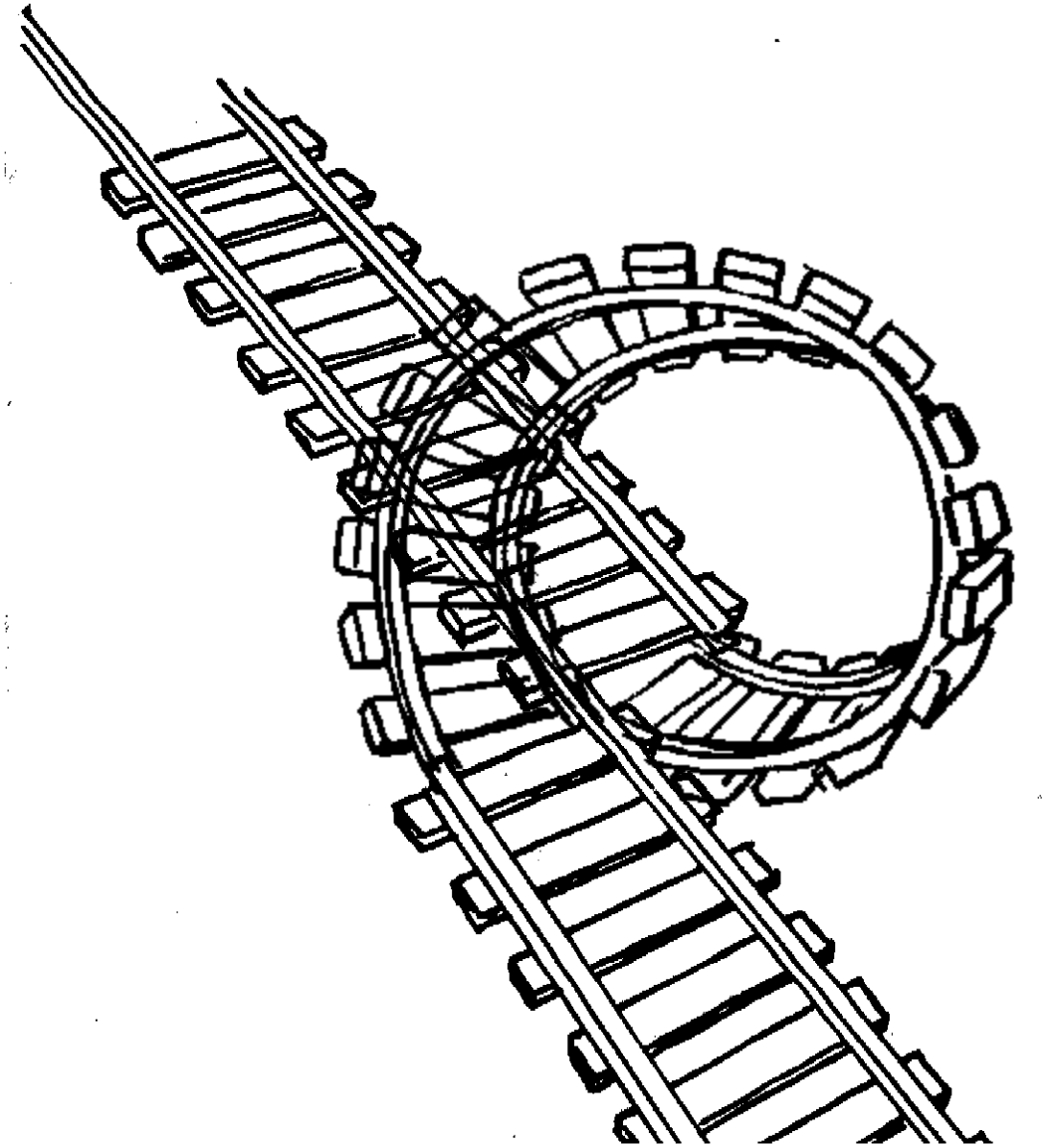
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

How to Deal with Errors

- **Error prevention** (before the system is released):
  - Use good programming methodology to reduce complexity
  - Use version control to prevent inconsistent system
  - Apply verification to prevent algorithmic bugs

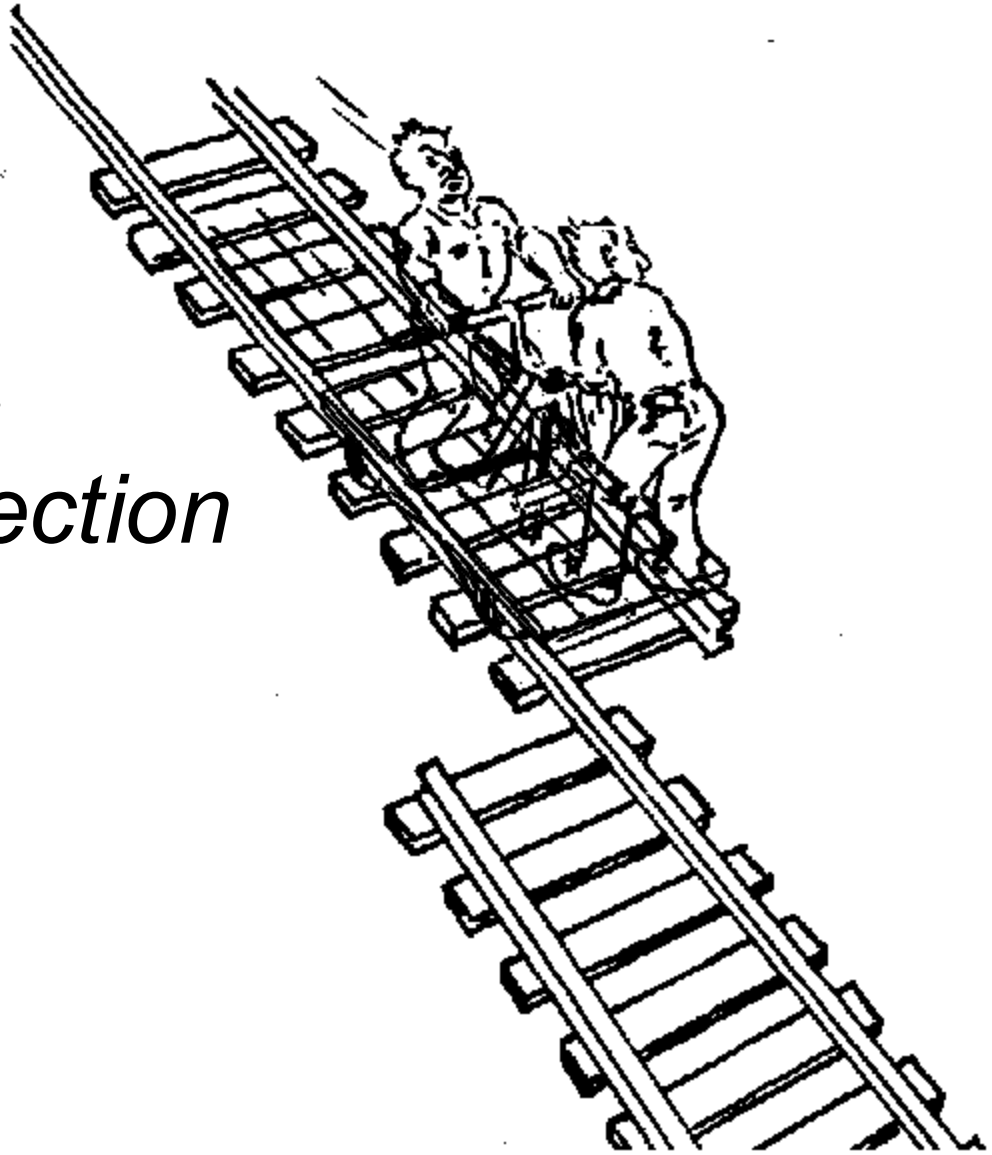
How do we deal with Errors and Faults?

Verification



How do we deal with Errors and Faults?

- *Error detection*



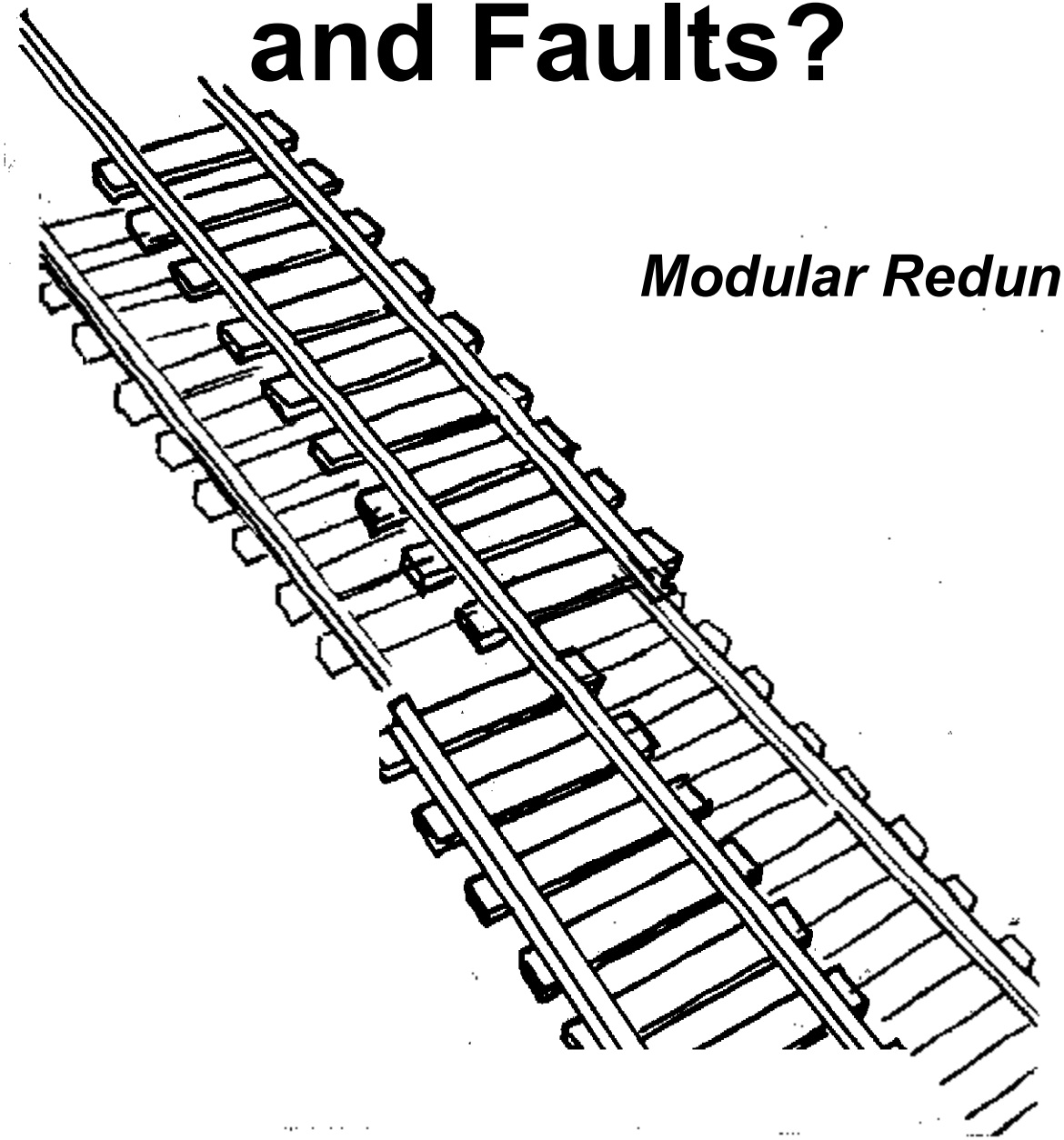
How do we deal with Errors and Faults?

- **Error detection** (while system is running):
  - Testing: Create failures in a planned way
    - Testing can only show the presence of bugs, not their absence (Dijkstra)
  - Debugging: Start with an unplanned failures
  - Monitoring: Deliver information about state. Find performance bugs

Fault tolerance

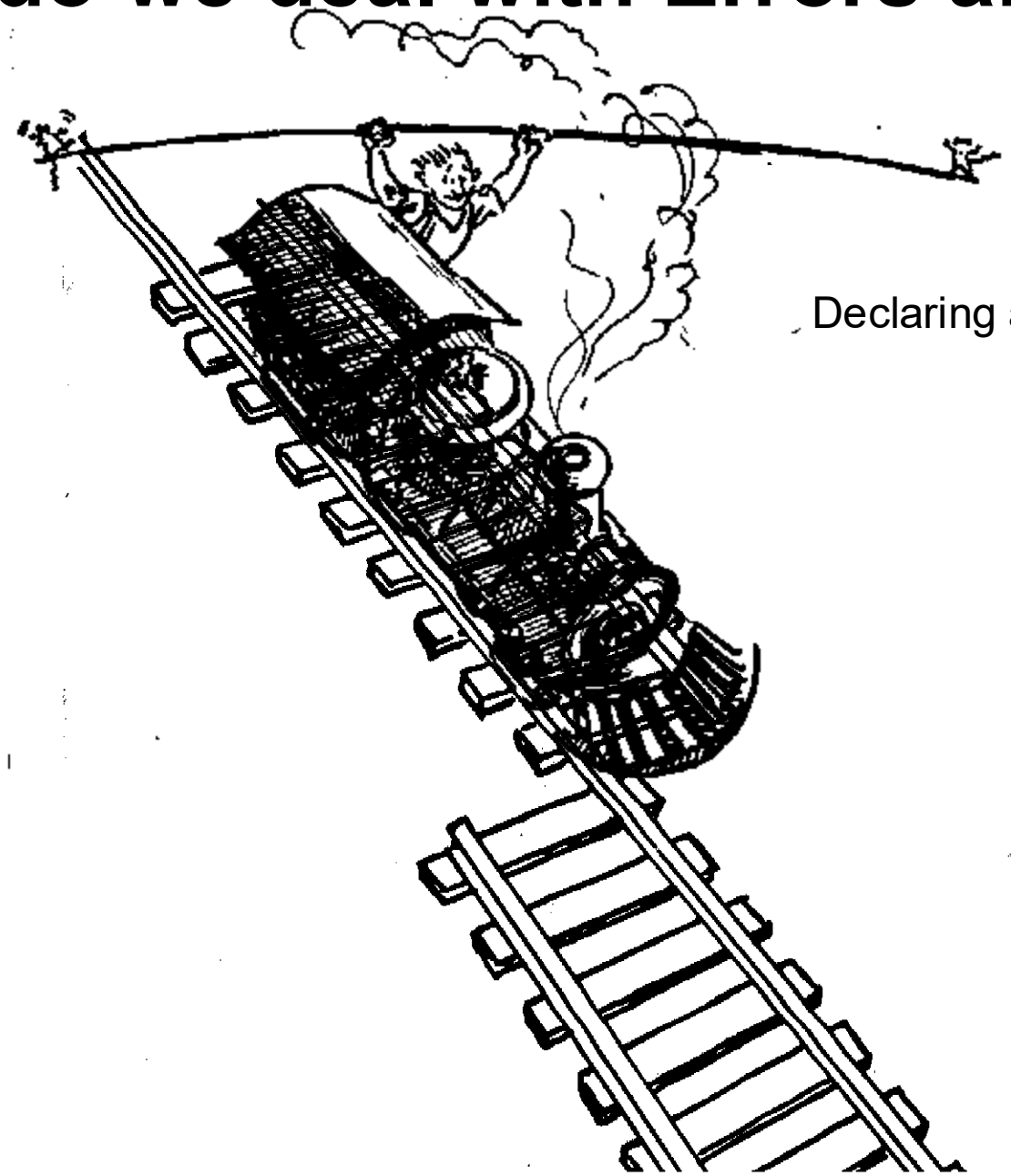
- Assumes that the system can be released with faults and that system failures can be dealt with by recovering from them in runtime

How do we deal with Errors and Faults?



Modular Redundancy?

How do we deal with Errors and Faults?



Declaring a bug to be a “feature”

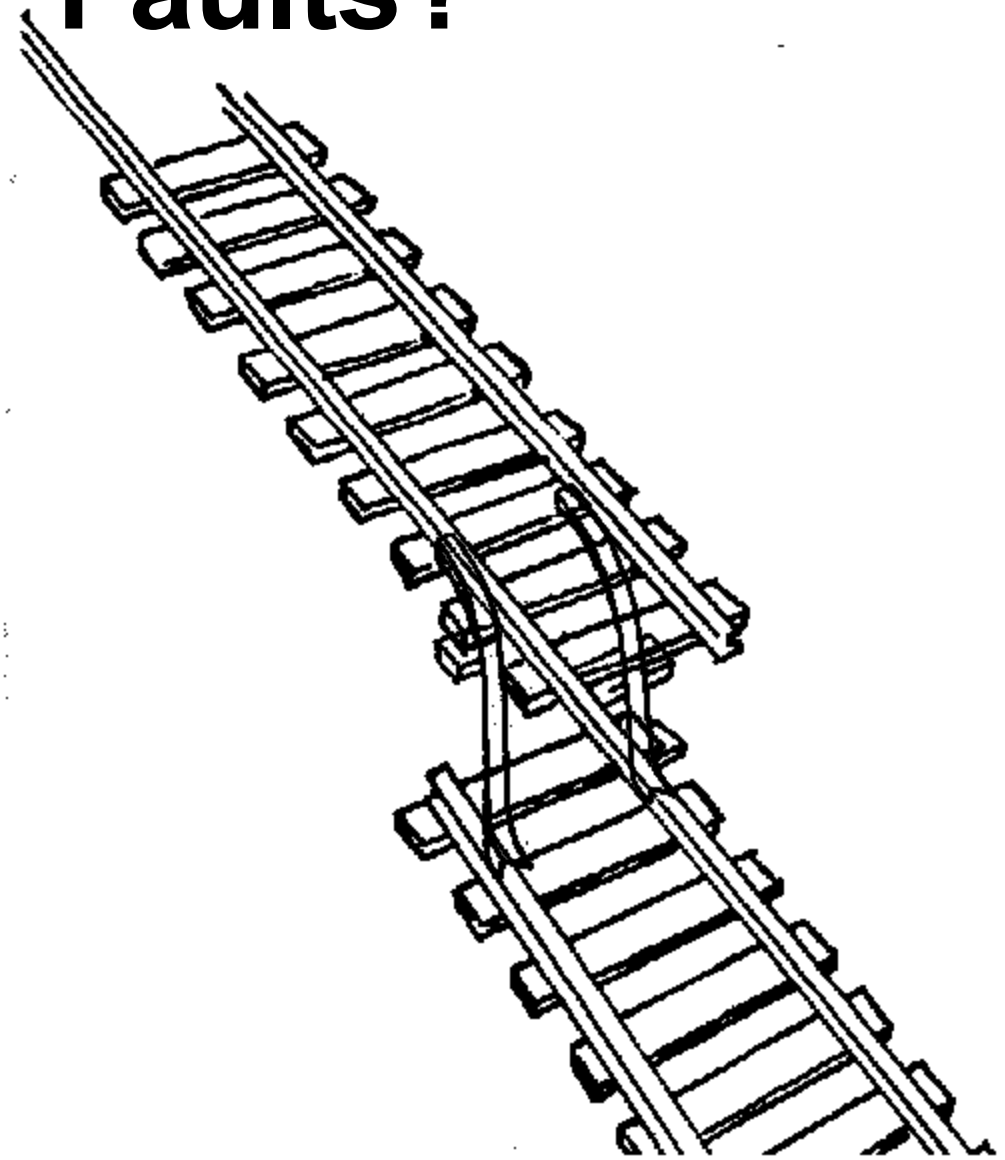
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

How do we deal with Errors and Faults?

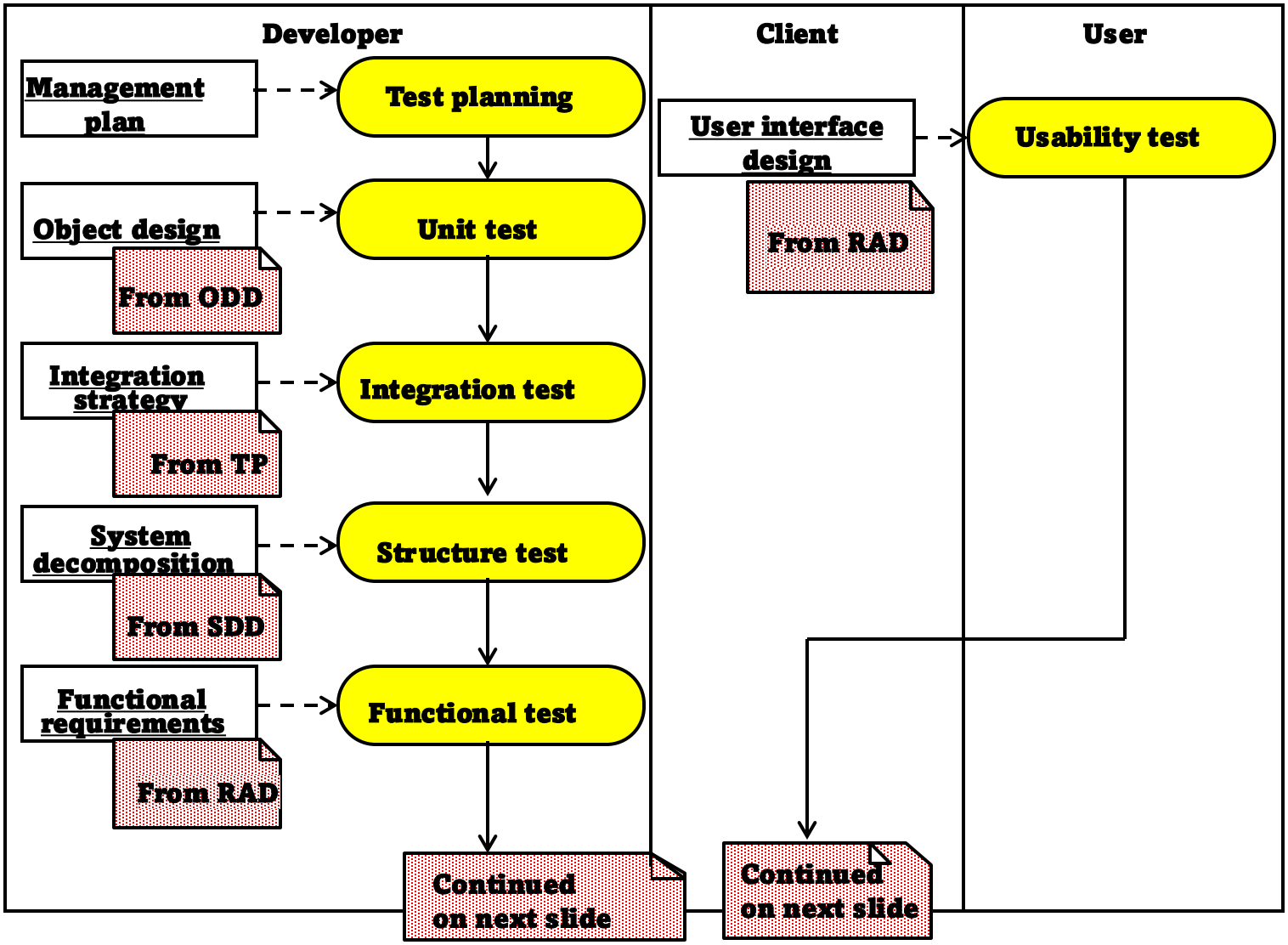
- **Error recovery** (recover from failure once the system is released):
  - Data base systems (atomic transactions)
  - Recovery blocks

How do we deal with Errors and Faults?

Patching?

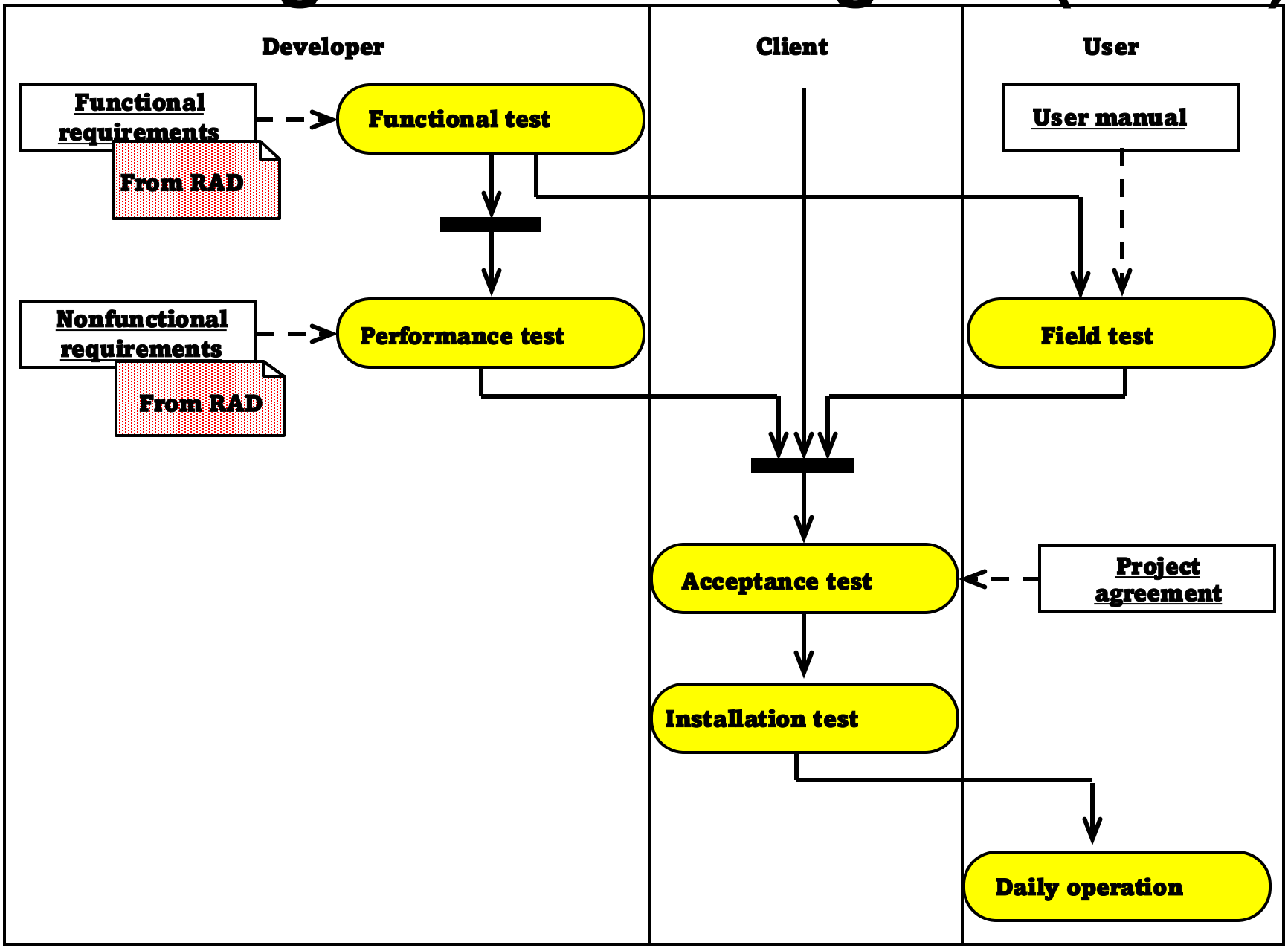


Testing Activities Diagram



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Testing activities Diagram (cntd.)



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Usability testing

- Scenario test
- Prototype test
- Product test

Unit Testing elements

- Static Analysis:
  - Hand execution: Reading the source code
  - 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 object)

Black-box Testing

- Focus: I/O behavior.
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.
- 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

A (faulty) implementation of the `getNumDaysInMonth()` method

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};
public class MyGregorianCalendar {
   public static boolean isLeapYear(int year) {
        boolean leap;
        if (year%4==0) {
                leap = true;
        } else {
                leap = false;
        }
        return leap;
   }
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;
}...
```

Equivalence classes and selected valid inputs

Equivalence classes input	Month input	Year
31 days, non-leap year	January	1901
31 days, leap year	January	1928
30 days, non-leap year	April	1901
30 days, leap year	April	1928
28 or 29 days, non-leap year	February	1901
28 or 29 days, leap year	February	1928

Boundary testing

Equivalence class	Month input	Year input
Nonpositive invalid months	0	35
Positive invalid months	13	3000
Leap years divisible by 400	February	2000
Non-Leap years divisible by 100	February	1900

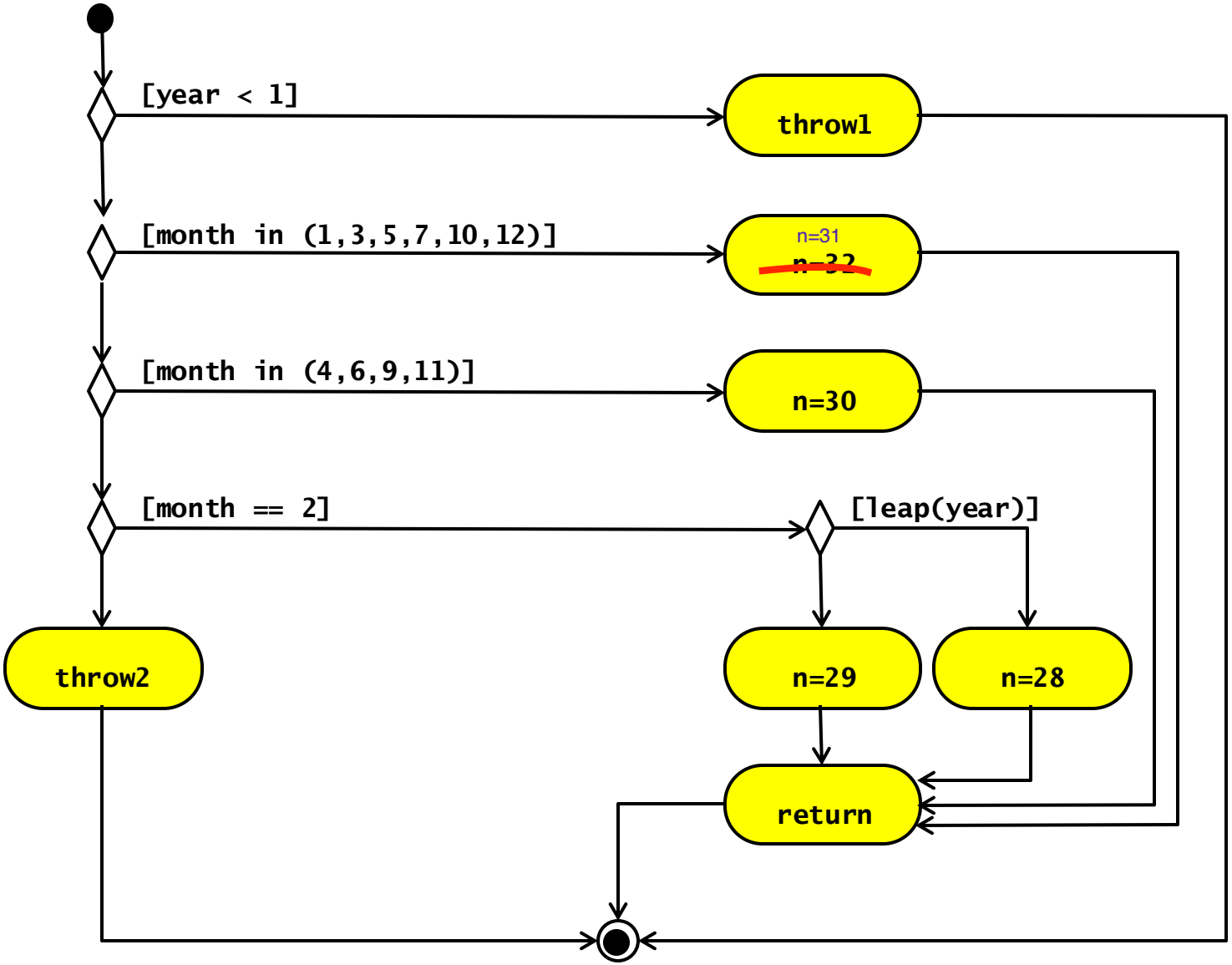
White-box Testing

- Focus: Coverage. Every statement in the component is executed at least once.
- Statement Testing: Test single statements
- Loop Testing:
  - Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
  - Cause execution of the loop to be executed exactly once
  - Cause execution of the loop to be executed more than once

White-box Testing

- Branch Testing (Conditional Testing):
  - Make sure that each possible outcome from a condition is tested at least once
- Path testing:
  - Make sure all paths in the program are executed

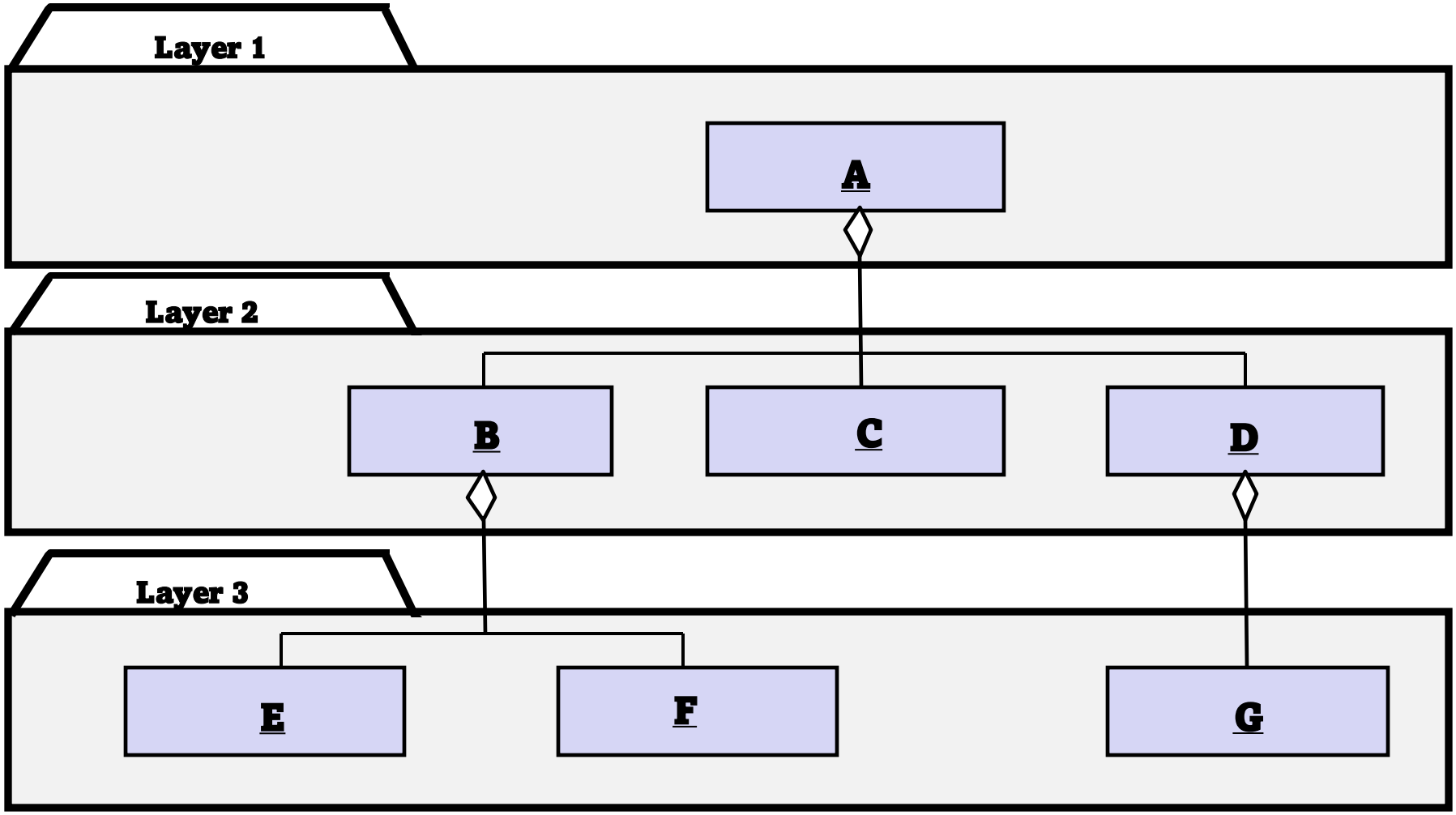
Equivalent flow graph for the `getNumDaysInMonth()` method implementation



Path testing

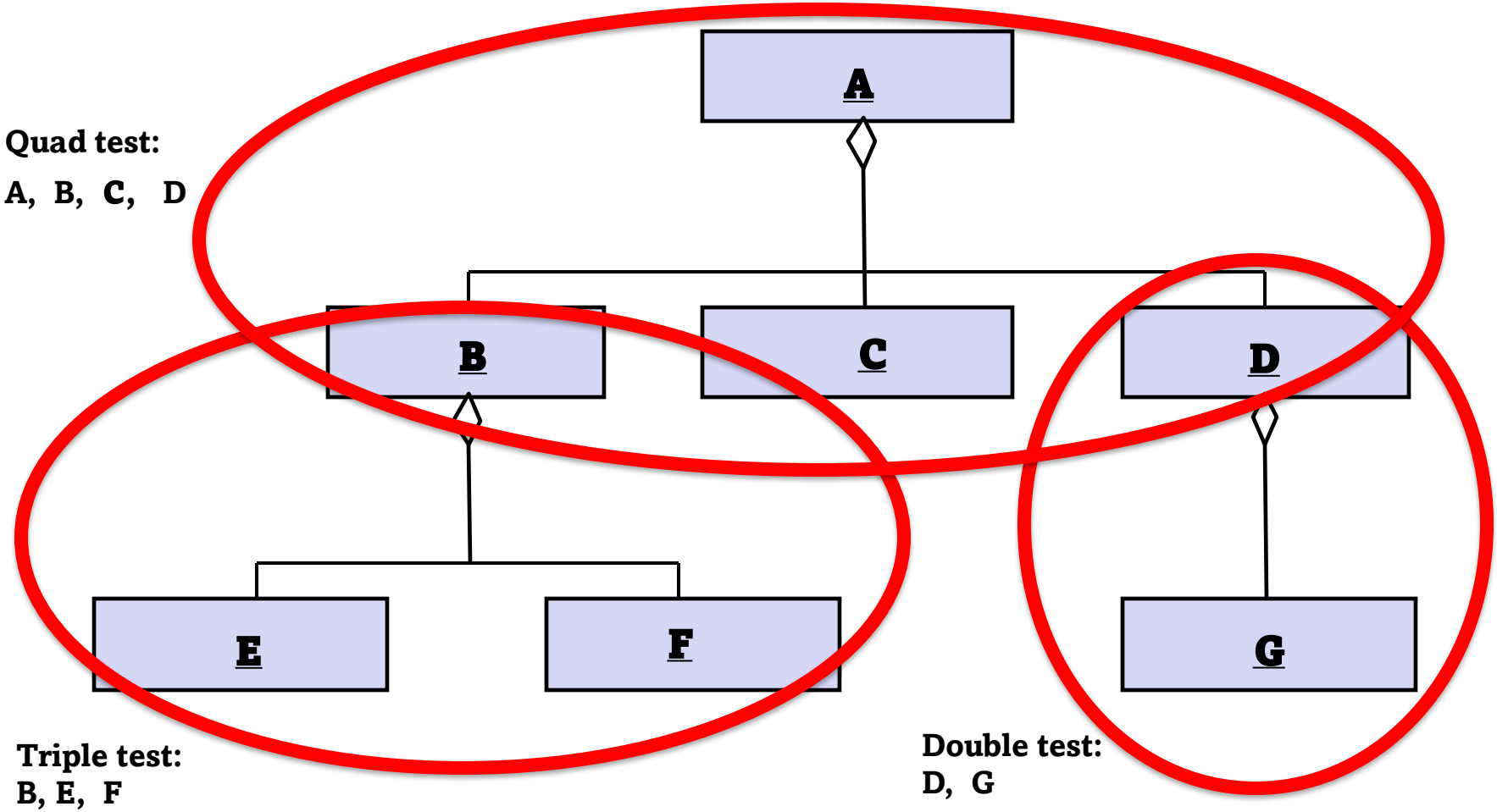
Test case	Path
(year=0, month=1)	{throwErr1}
(year=1905, month=1)	{n=32 return}
(year=1905, month=2)	{n=28 return}
(year=1908, month=2)	{n=29 return}
(year=1905, month=4)	{n=30 return}
(year=1905, month=0)	{throwErr2}

Integration testing strategies



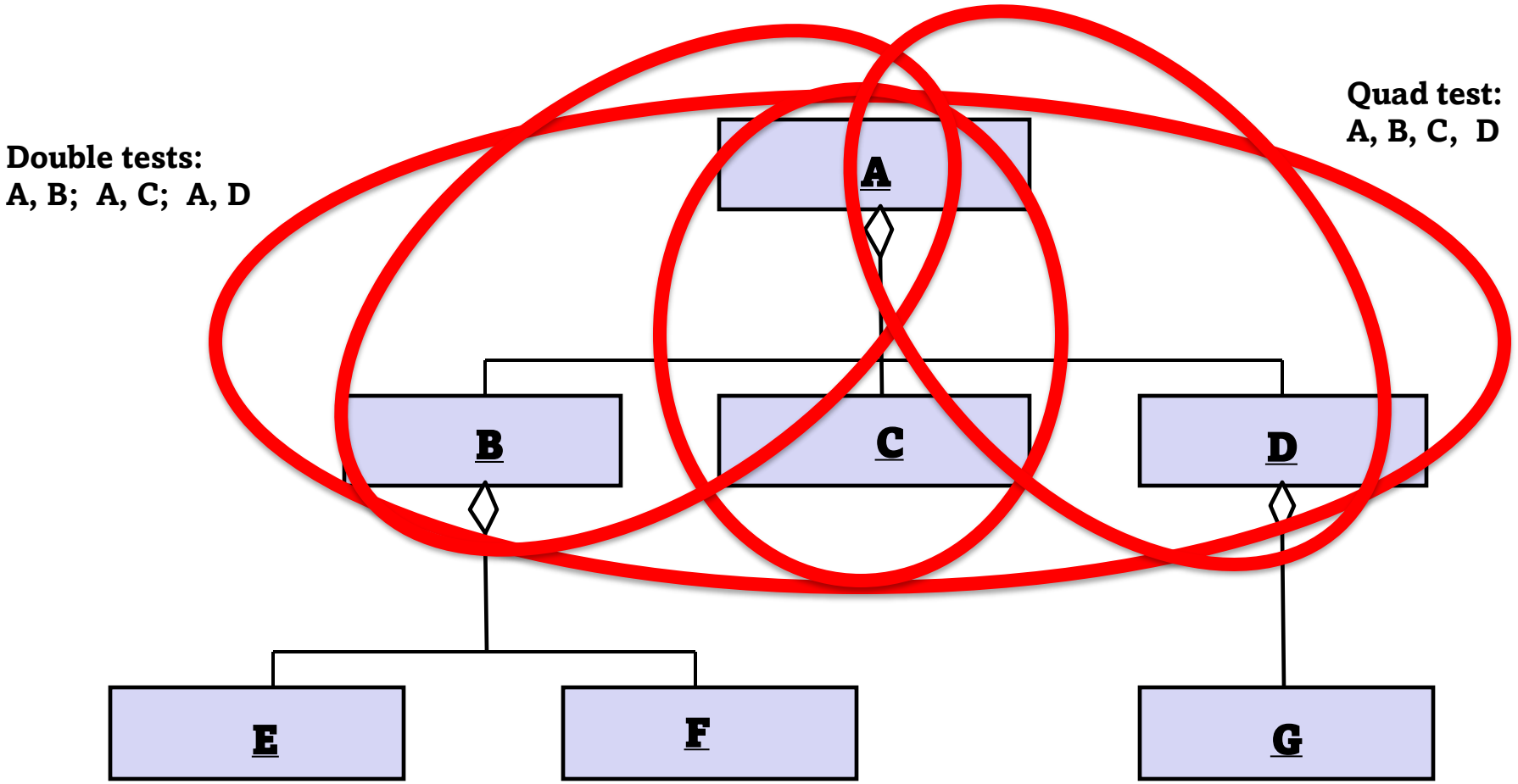
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Integration testing strategies (bottom-up)

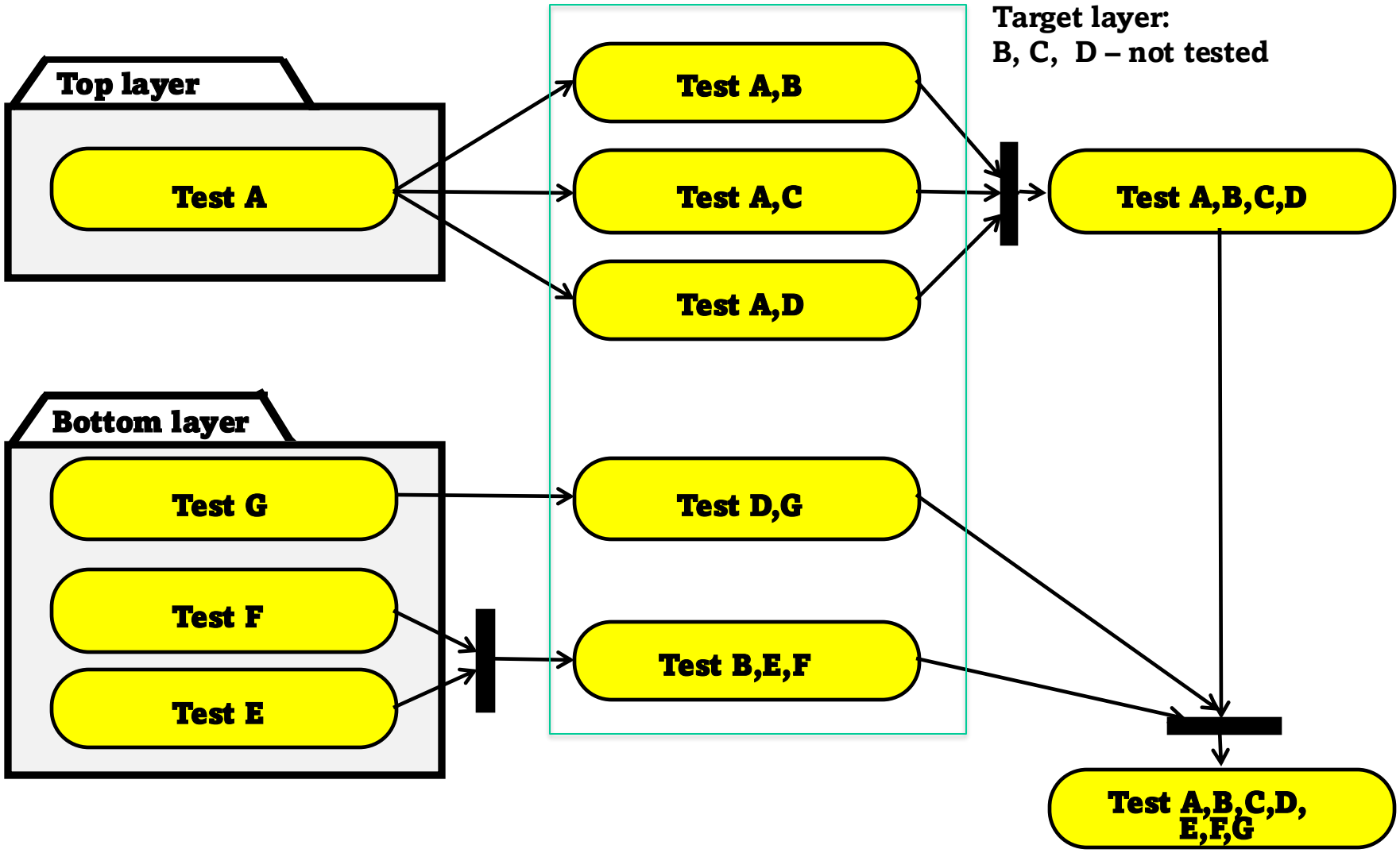


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Integration testing strategies (top-down)

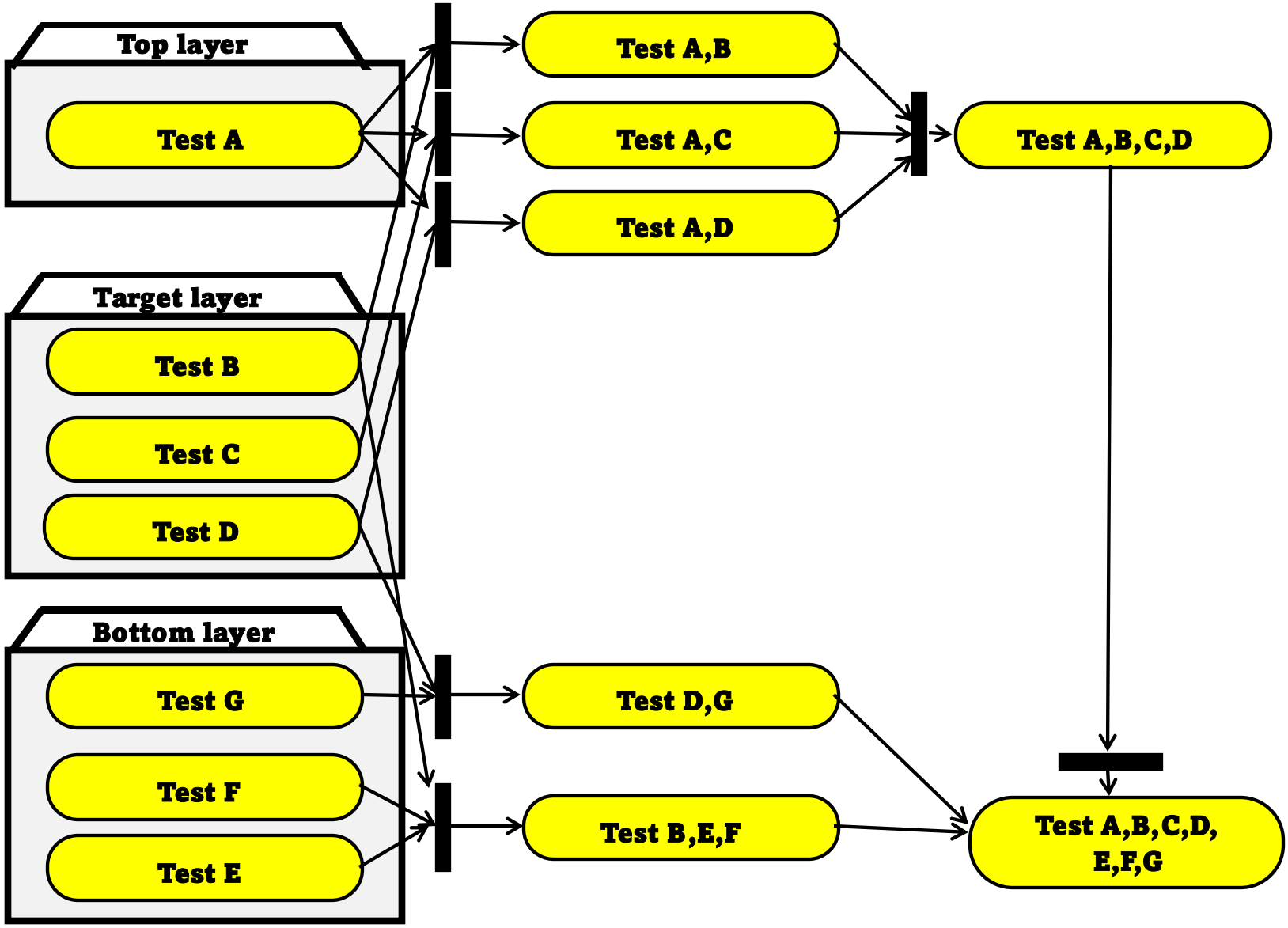


Integration testing strategies (sandwich)



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Integration testing strategies (sandwich)



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

System Testing

- Functional testing
- Performance testing
- Acceptance testing
- Pilot testing
- Installation testing

Functional testing

- Finds differences between the functional requirements and the system
- BlackBox technique
- Test cases are derived from use case model
- Selects tests that are relevant to the user and have high probability of a failure

Performance testing

- Stress testing
- Volume testing
- Security testing
- Timing testing
- Recovery tests

Acceptance testing

- Benchmark test
- Competitor testing
- Shadow testing

Pilot testing (field test)

- The system is installed and used by a selected set of users
- Pilot tests are useful when a system is built without a specific set of requirements or without a specific customer in mind
- An alpha test is a pilot test with users exercising the system in the development environment
- In a beta test, the acceptance test is performed by a limited number of end users in the target environment

Installation testing

- Testing reconfiguration
- Often repeats test cases from previous phases
- Some requirements cannot be executed in the development environment because they require target-specific resources