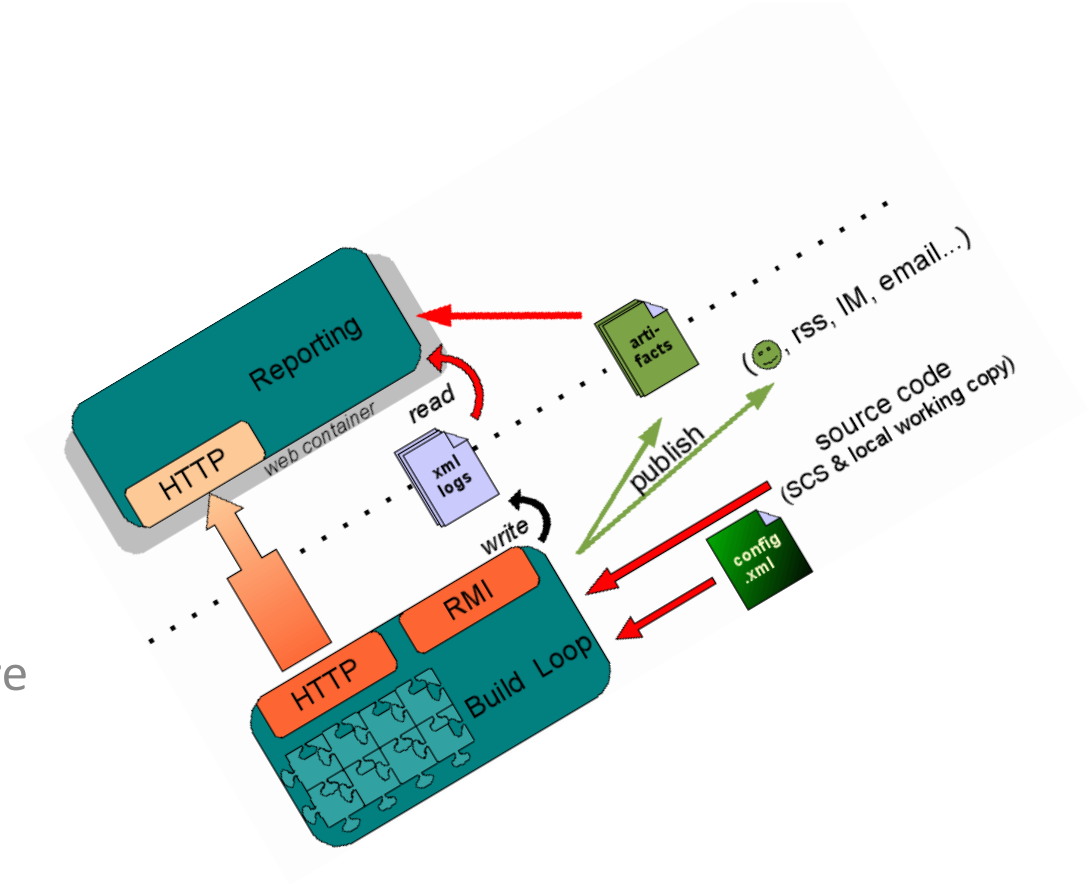


Analysis, Design, & Software Architecture (BDSA)

Jakob E. Bardram

QUALITY ASSURANCE & TESTING

IT UNIVERSITY OF COPENHAGEN

This Lecture

- Literature
  - [OOSE] ch. 11
  - [SE9] ch. 8 (+24)

- Introduction to Software Testing
- Testing Terminology
- Testing Activities
  - Unit / Component Testing
  - Integration Testing
  - System Testing
  - Client / Acceptance Testing
- Managing Testing
  - Test Cases
  - Test Teams
  - Test Driven Development
  - Documenting Testing

Program Testing

- Testing is
  - intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
  - the process of finding difference between the expected behavior specified by system models and the observed behavior of the implemented system
  - the attempt to show that the implementation of the system is inconsistent with the system models
- The goal of testing is to
  - design tests that exercise defects in the system
  - to reveal problems
- Testing is in contrast to all other system activities
  - testing is aimed at breaking the system

- HENCE : testing can reveal the presence of errors – NOT their absence!
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

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

- 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

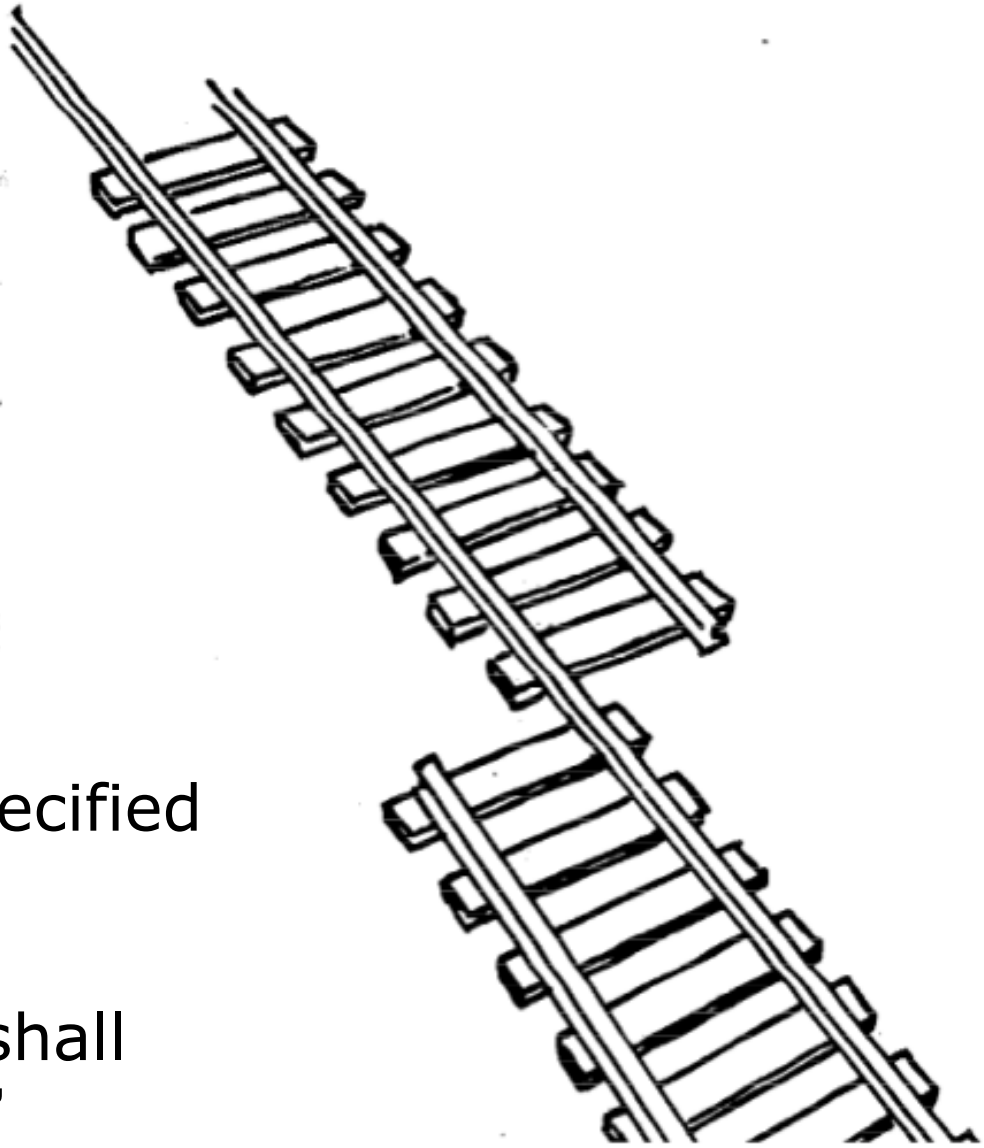
- Failure to properly reuse the old software from Therac-6 and Therac-20 when using it for new machine
- Cryptic warning messages
- End users did not understand the recurring problem (5 patients died)
- Lack of communication between hospital and manufacturer
- The manufacturer did not believe that the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

Testing Terminology

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?

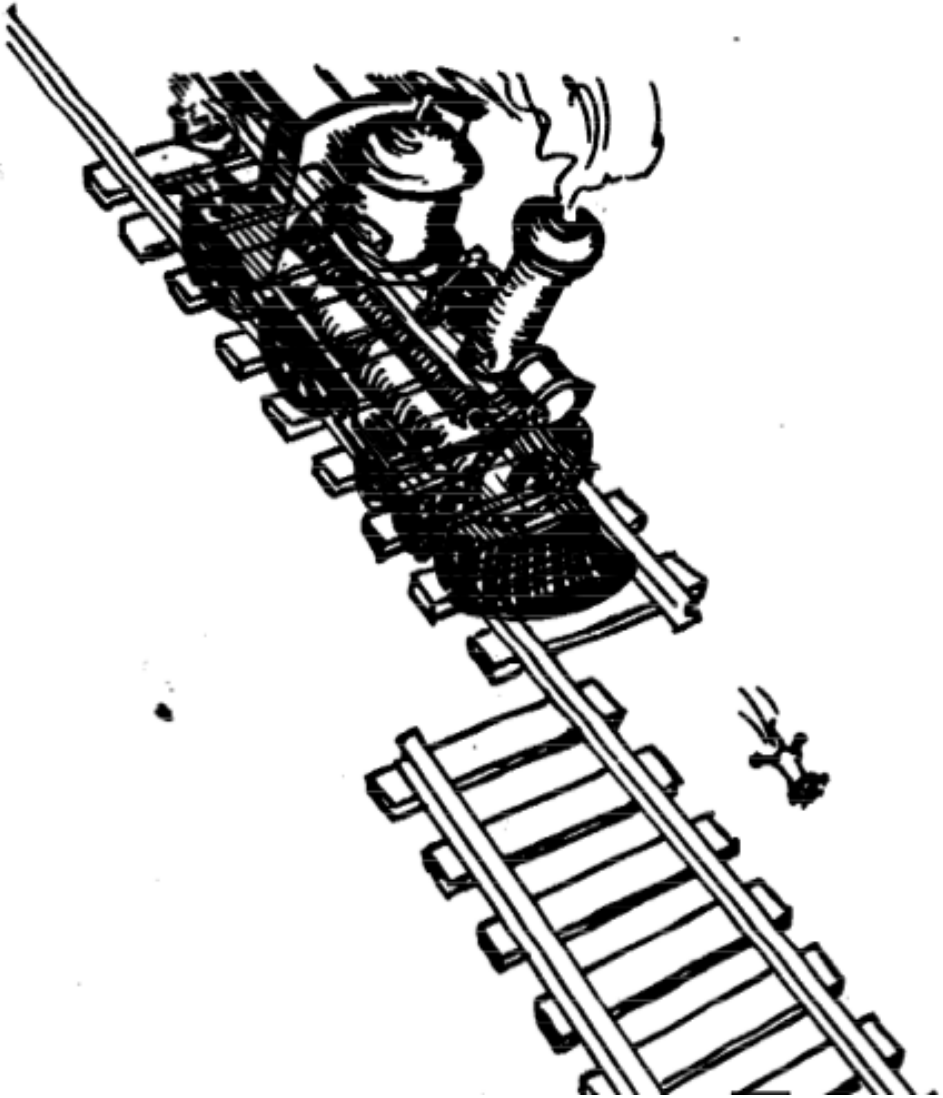
An error?

A fault?

We need to describe specified behavior first!

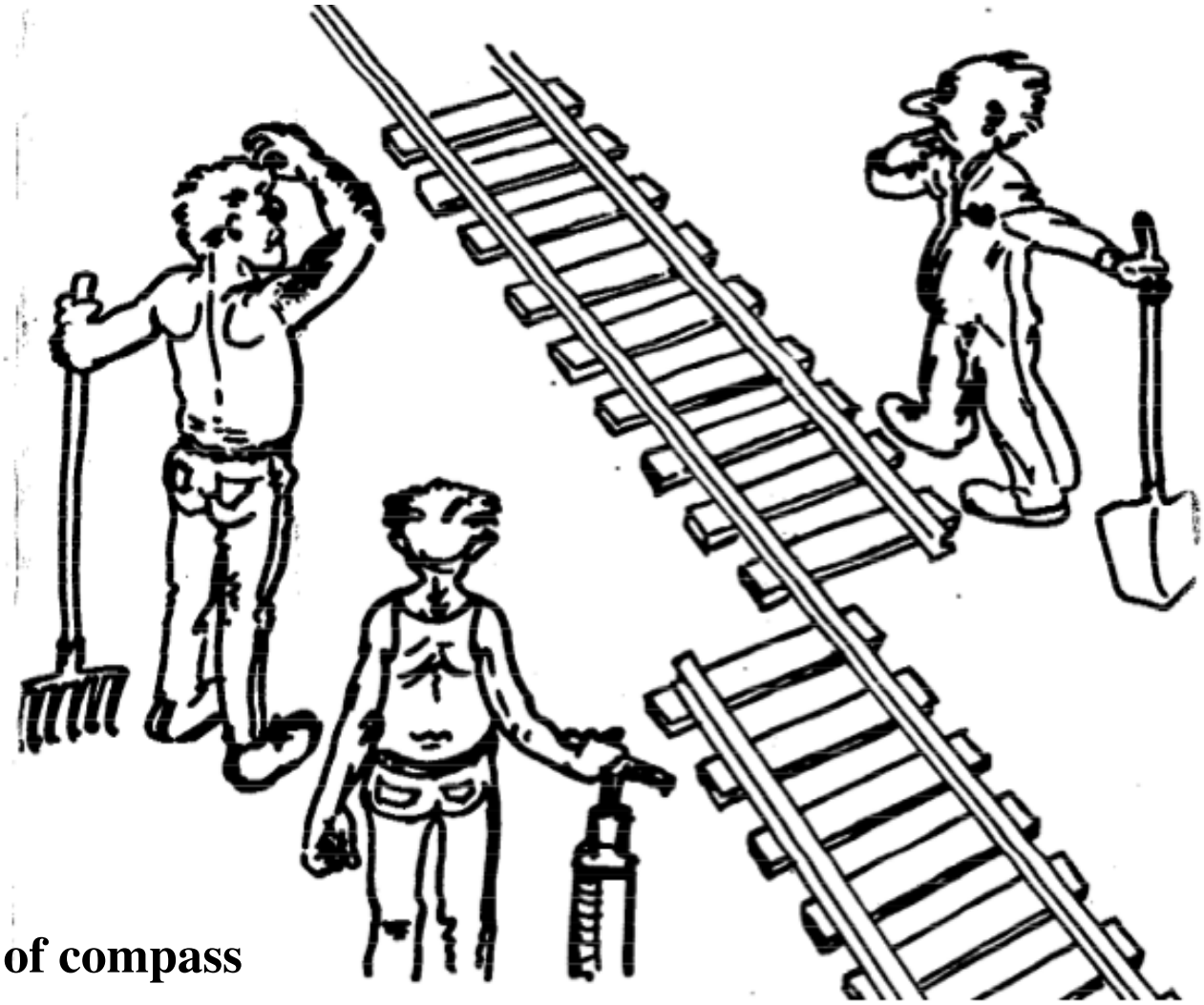
Specification: “A track shall support a moving train”

Erroneous State ("Error")



Fault

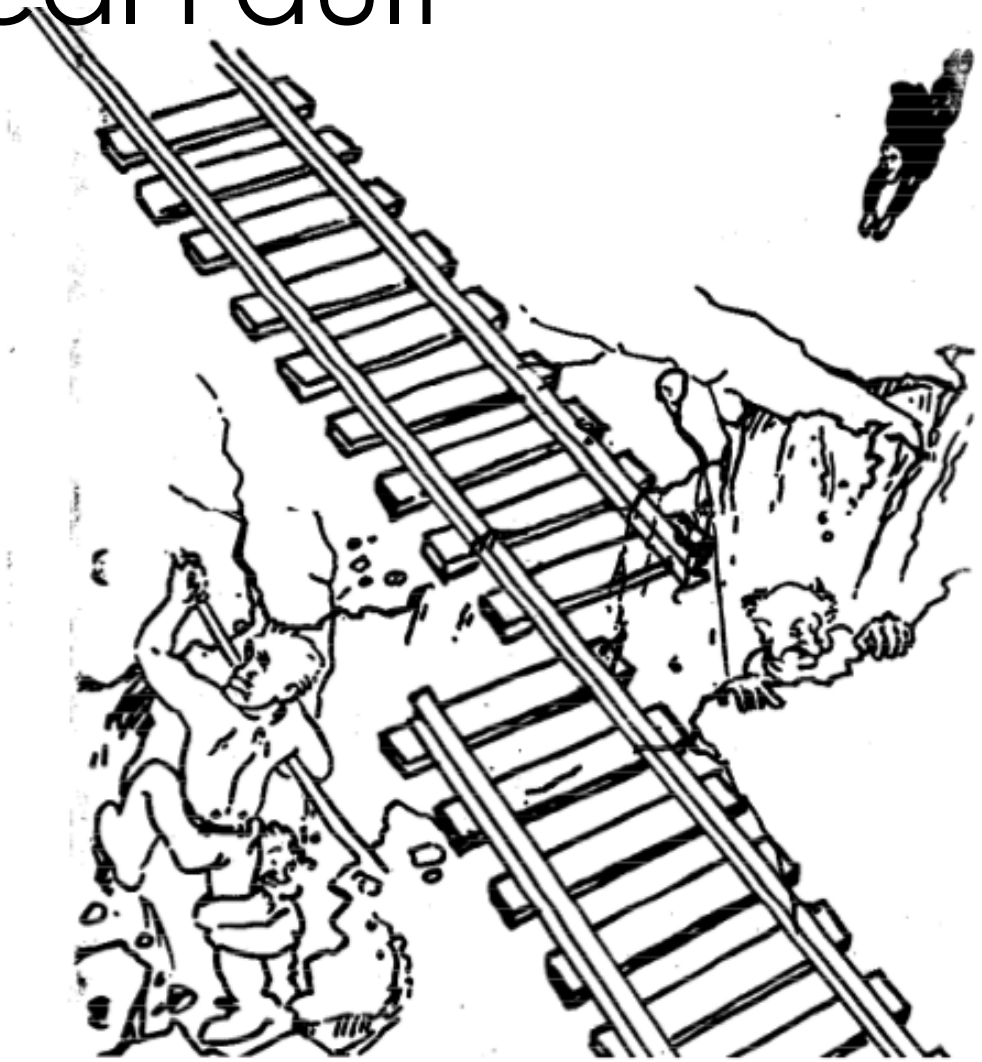
Possible algorithmic fault: Compass shows wrong reading



Or: Wrong usage of compass

Another possible fault: Communication problems between teams

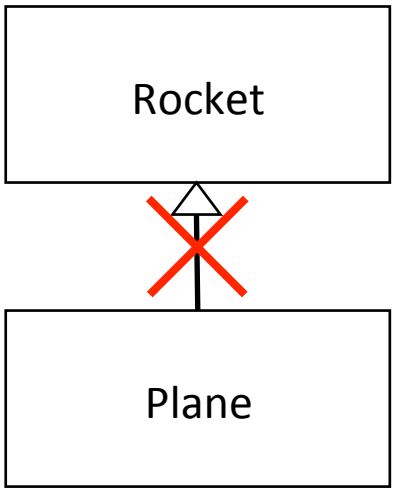
Mechanical Fault



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.



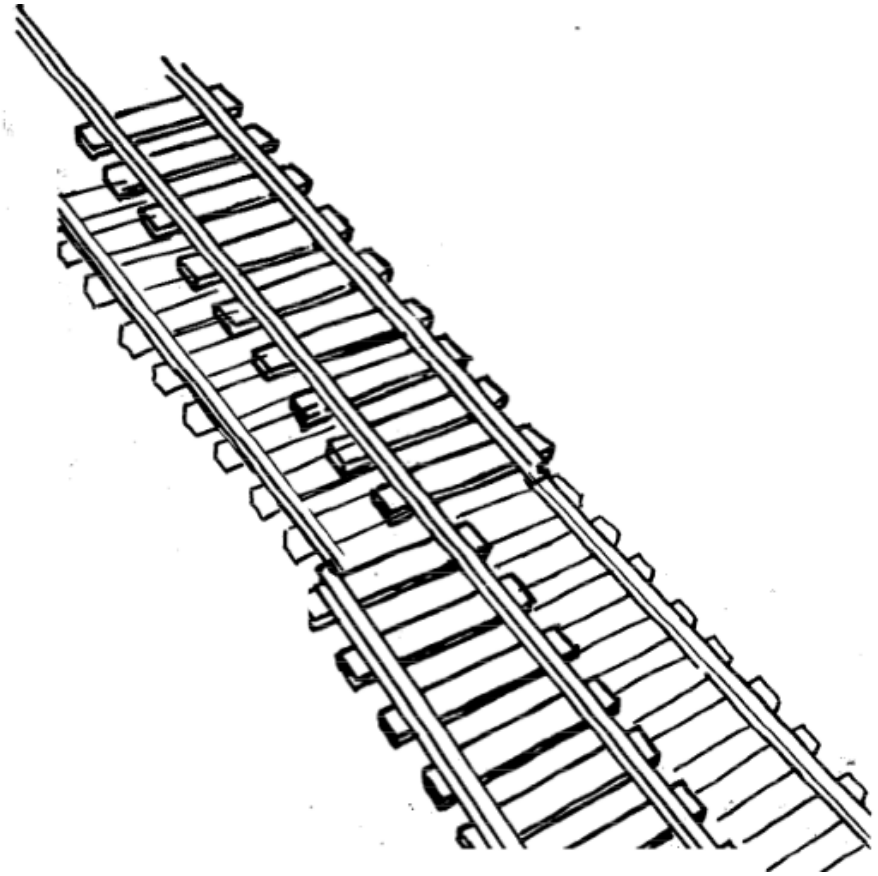
Examples of Faults and Errors

- 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

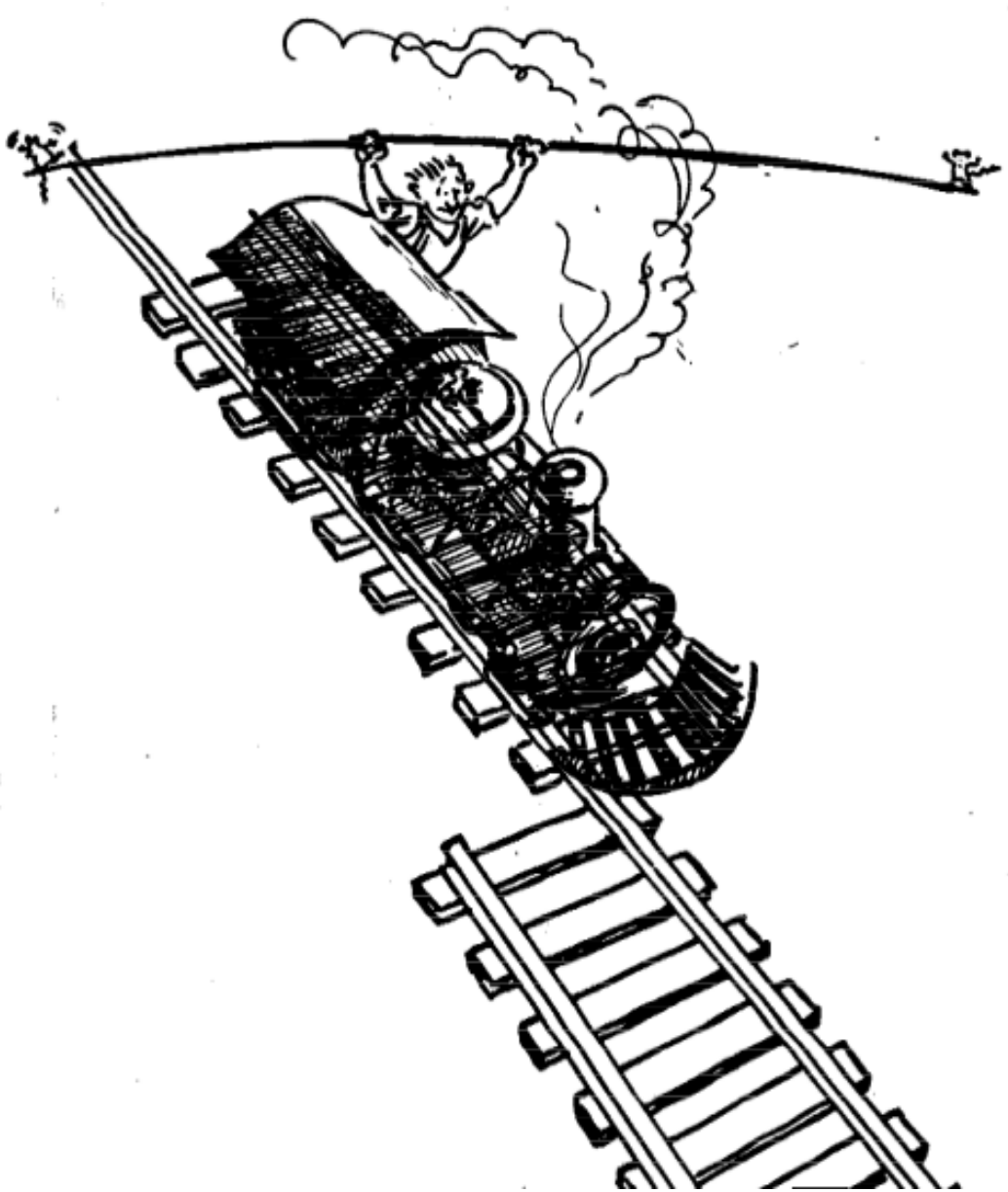
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Wrong user input
  - Null reference errors
  - Concurrency errors
  - Exceptions.

How do we deal with Errors, Failures and Faults?

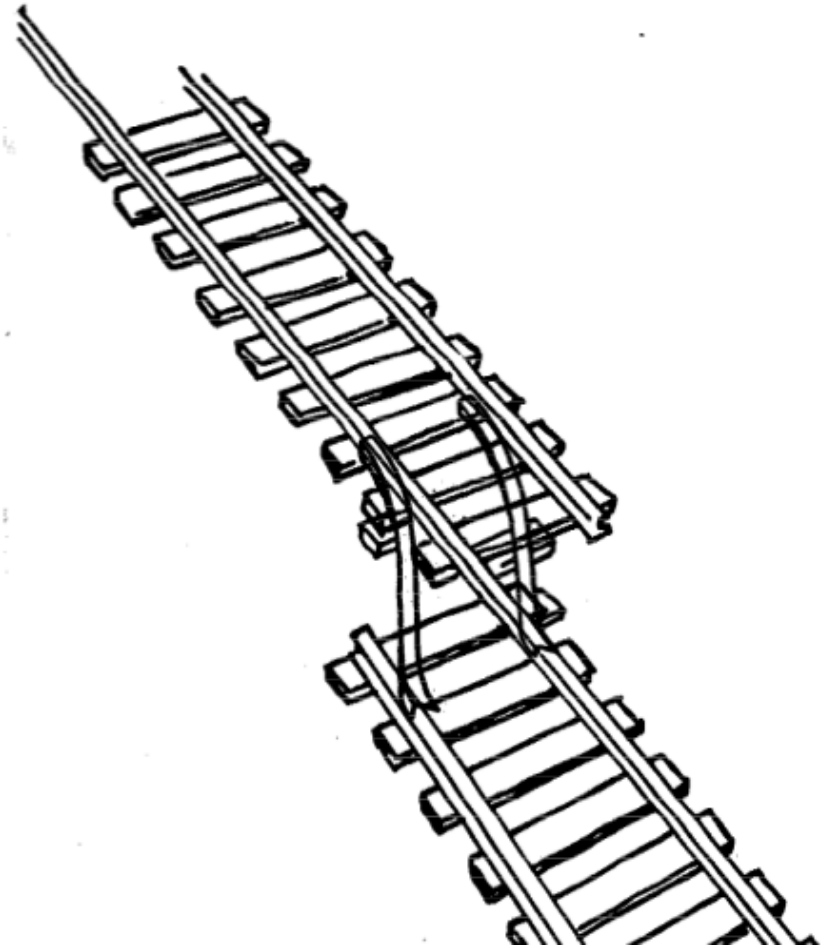
Modular Redundancy



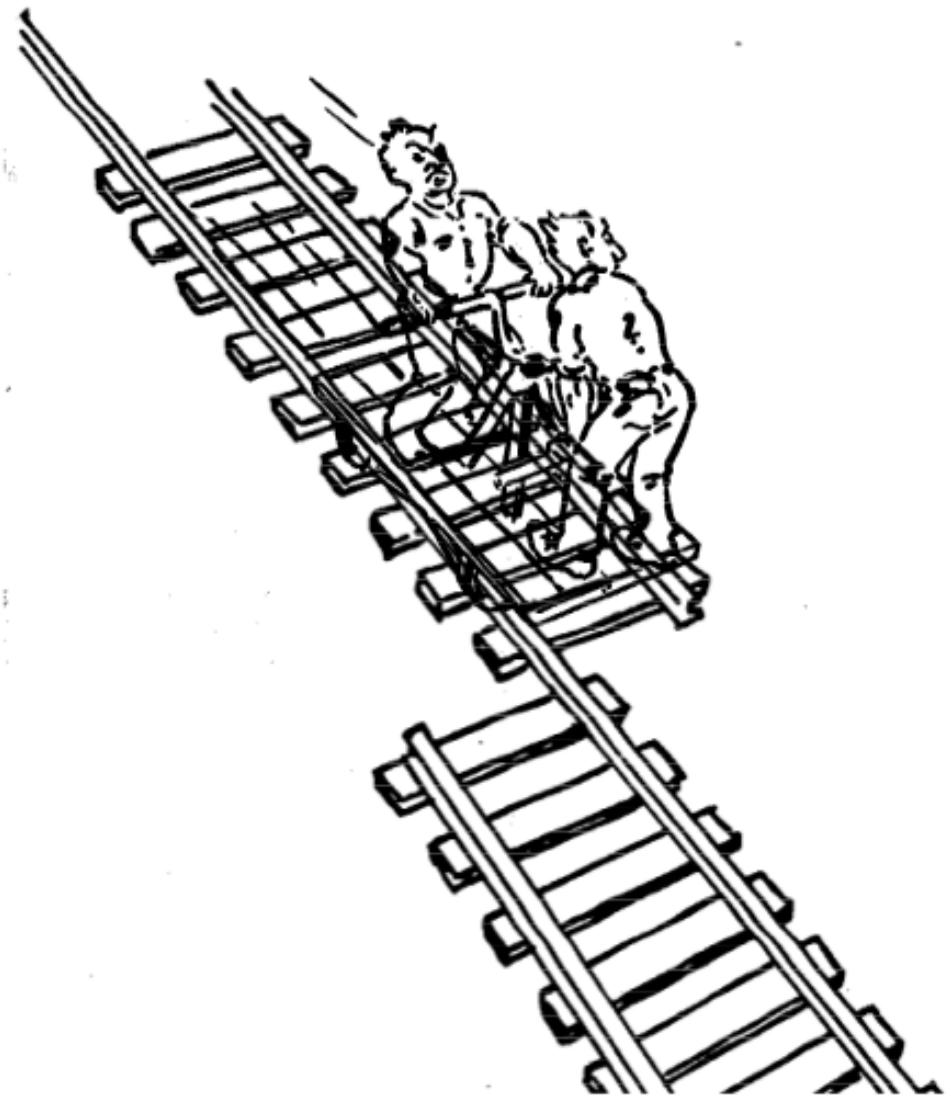
Declaring the Bug as a Feature



Patching

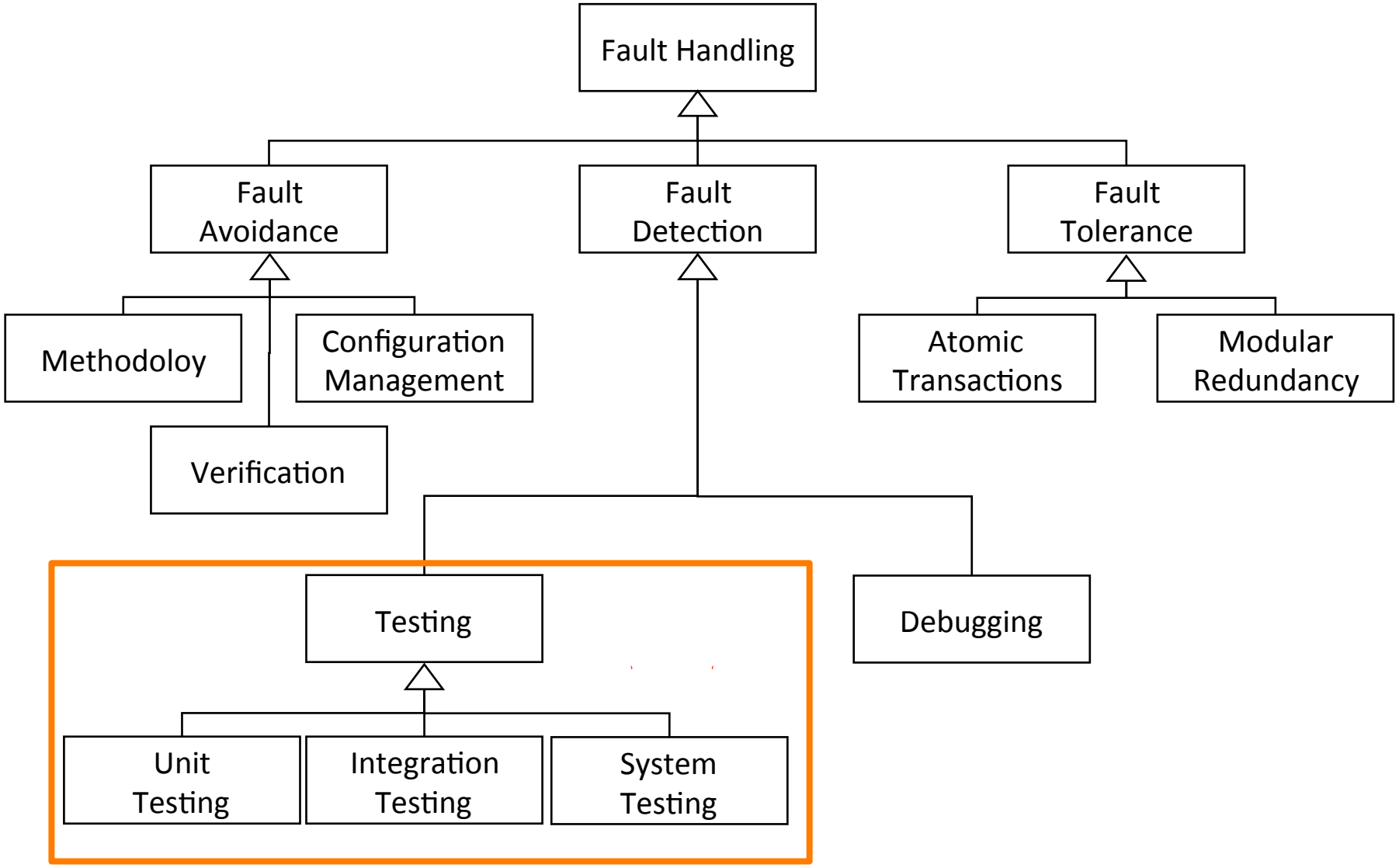


Testing



Another View on How to Deal with Faults

- Fault avoidance
  - 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
  - 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
  - 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 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.

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.

Automated Testing

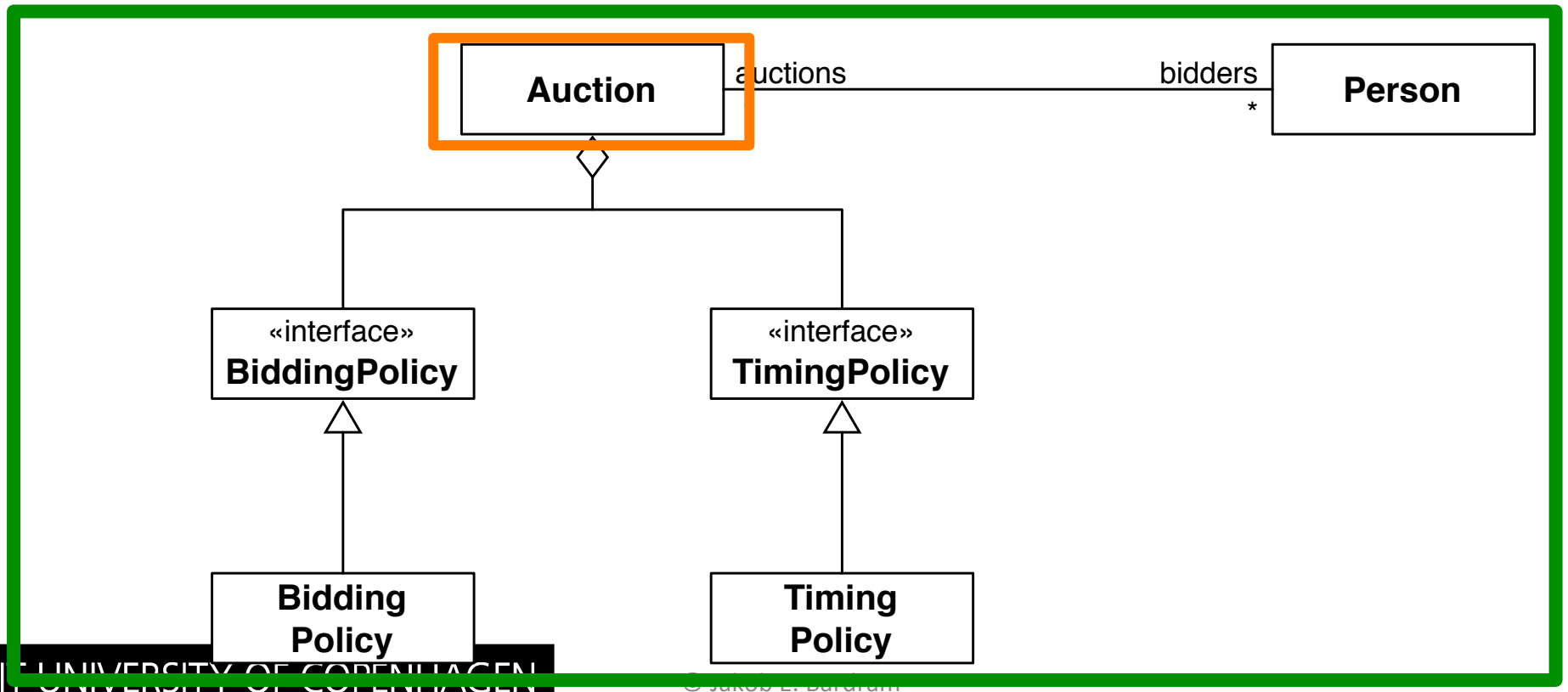
- There are two ways to generate the test model
  - Manually: The developers set up the test data, run the test and examine the results themselves. Success and/or failure of the test is determined through observation by the developers
  - Automatically: *Automated generation* of test data and test cases. Running the test is also done automatically, and finally the comparison of the result with the oracel 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.

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.

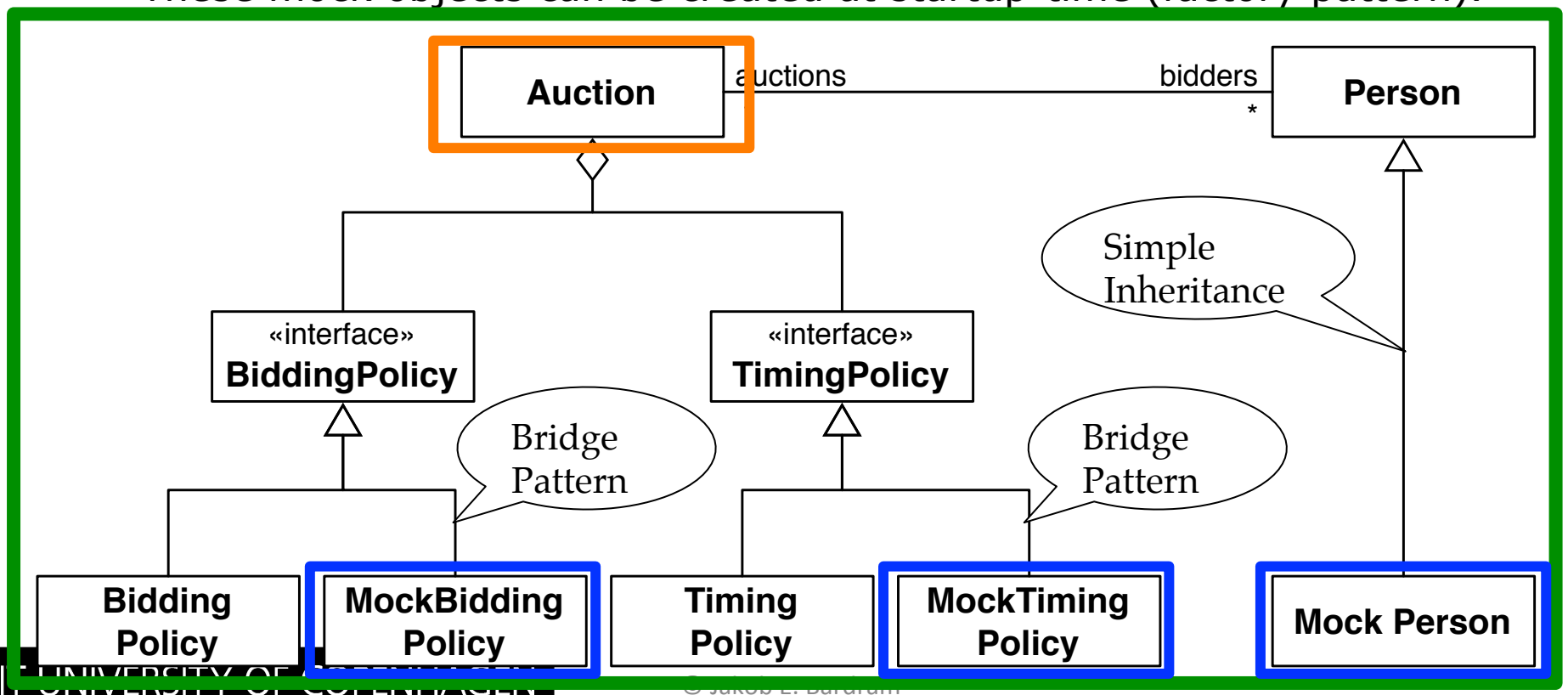
Motivation for the Mock Object Pattern

- Let us assume we have a system model for an auction system with 2 types of policies. We want to unit test Auction, which is our SUT

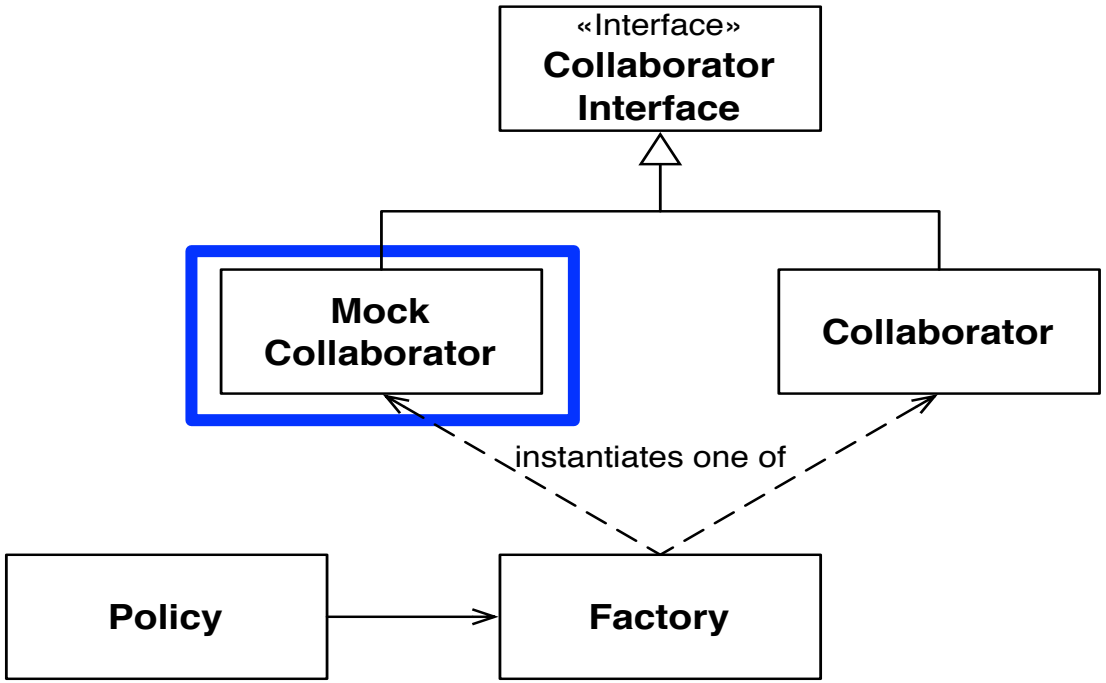


Motivation for the Mock Object Pattern

- Let us assume we have a system model for an auction system with 2 types of policies. We want to unit test Auction, which is our SUT
- The mock object test pattern is based on the idea to replace the interaction with the collaborators in the system model, that is Person, the Bidding Policy and the TimingPolicy by mock objects
- These mock objects can be created at startup-time (factory pattern).



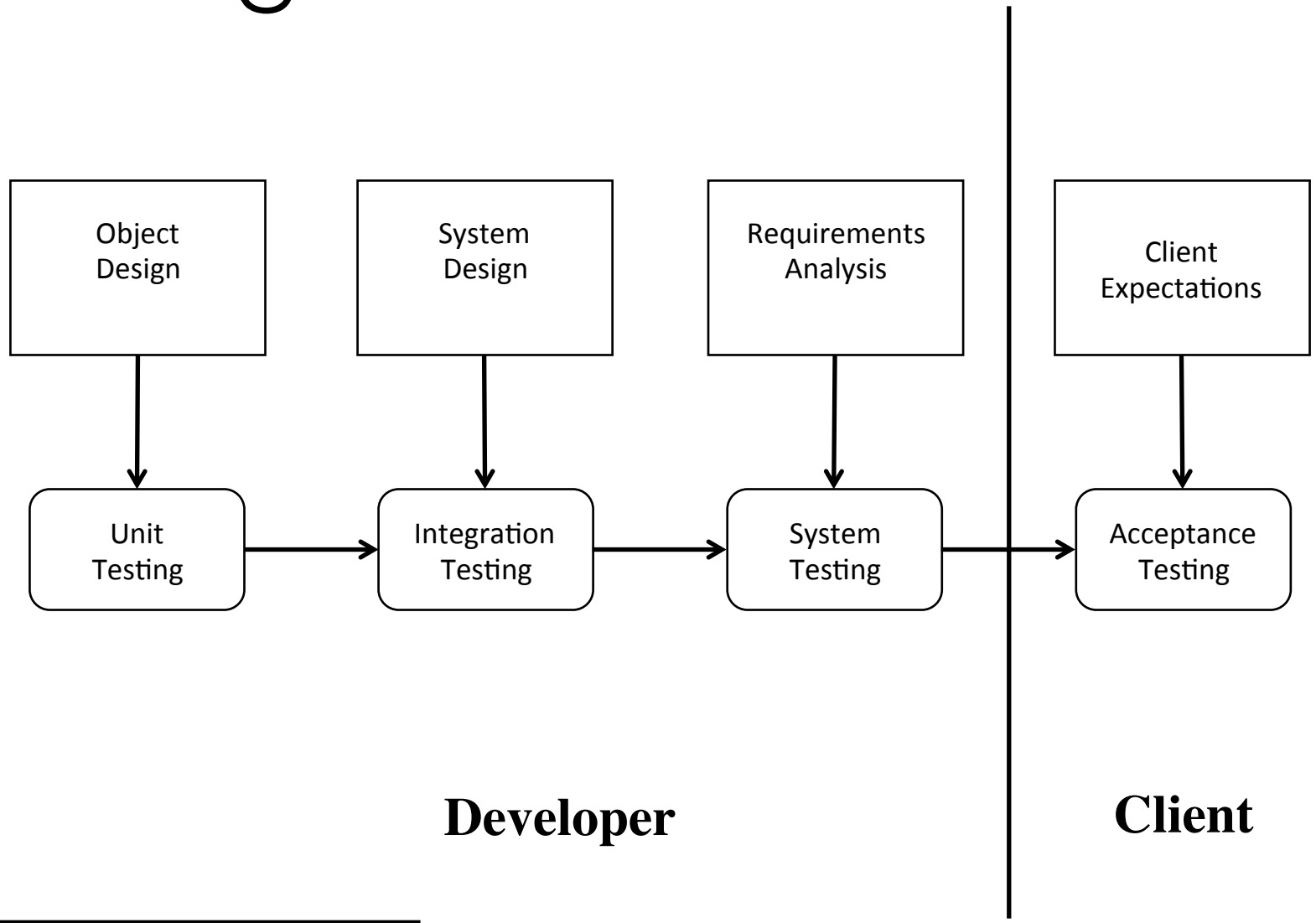
Mock-Object Pattern



- In the mock object pattern a mock object replaces the behavior of a real object called the collaborator and returns hard-coded values
- These mock objects can be created at startup-time with the factory pattern
- Mock objects can be used for testing *state of individual objects* as well as the *interaction between objects*, that is, to validate that the interactions of the SUT with collaborators behave is as expected.

Testing Activities

Testing Activities and Models

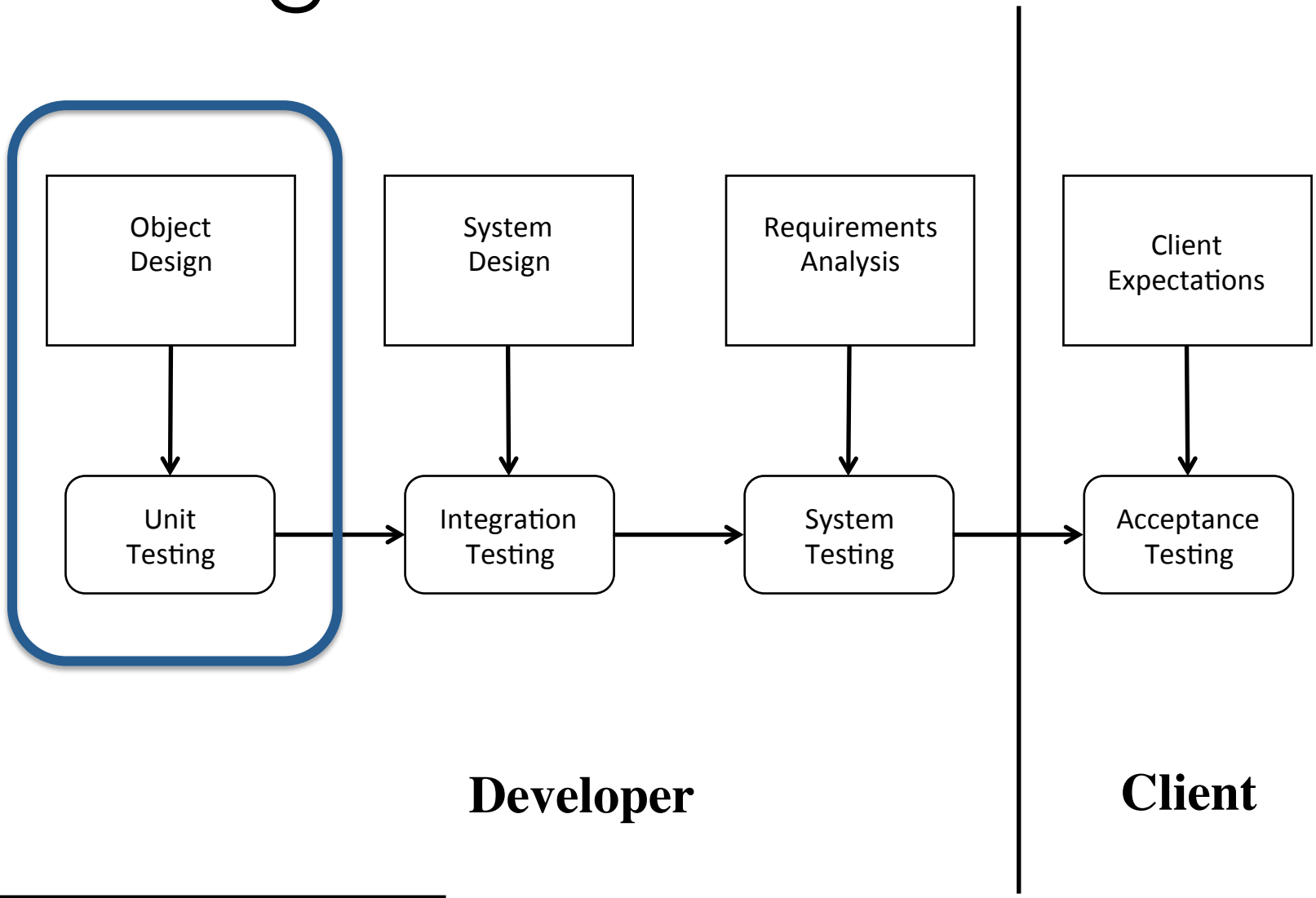


Types of Testing

- 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.

Unit / Component Testing

Testing Activities and Models



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**

White-box Testing

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

White-box Testing (Continued)

- Statement Testing (Algebraic Testing)
  - Tests each statement (Choice of operators in polynomials, etc)
- Loop Testing
  - Loop to be executed exactly once
  - Loop to be executed more than once
  - Cause the execution of the loop to be skipped completely
- Path testing:
  - Makes sure all paths in the program are executed
- Branch Testing (Conditional Testing)
  - Ensure that each outcome in a condition is tested at least once
  - Example:

```
if ( i = TRUE) printf("Yes");     else printf("No");
```

How many test cases do we need to unit test this statement?

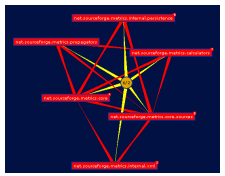
Example of Branch Testing

```
if ( i =  TRUE) printf(”Yes");      else printf(”No");
```

- We need two test cases with the following input data
  1) i = TRUE
  2) i = FALSE
- What is the expected output for the two cases?
  - In both cases: Yes
  - This a typical beginner‘s mistake in languages, where the assignment operator also returns the value assigned (C, Java)
- So tests can be faulty as well☹
- Some of these faults can be identified with static analysis.

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



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
  - Important to find the intersection of stupid and important mistakes
- 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 ("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.

When should you write a unit test?

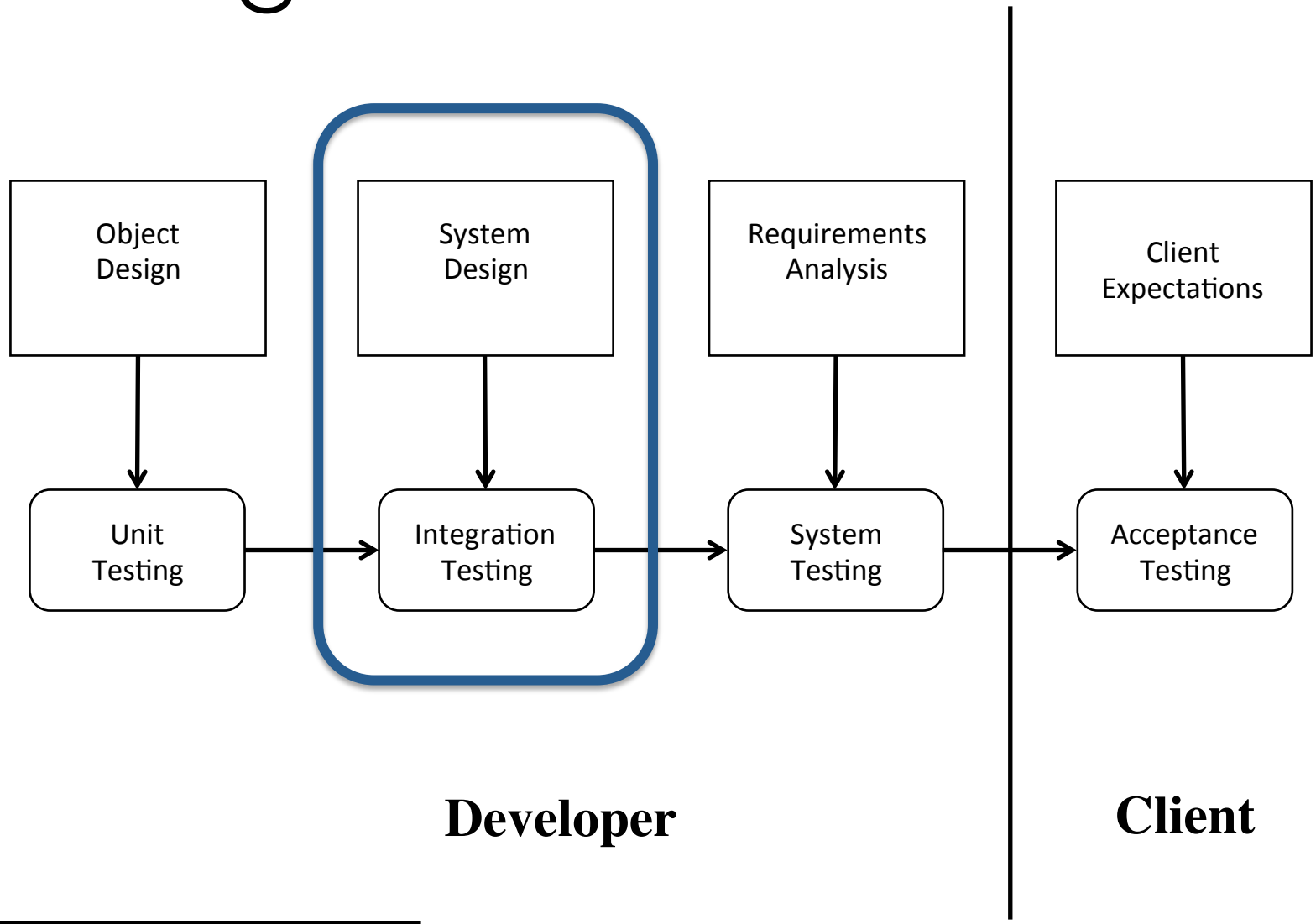
- Traditionally after the source code is written
- In XP/TDD before the source code is written

- Test-Driven Development Cycle
  - Add a new test to the test model
  - Run the automated tests
    => the new test will fail
  - Write code to deal with the failure
  - Run the automated tests

  => see them succeed
  - Refactor code.

Integration Testing

Testing Activities and Models



Integration Testing

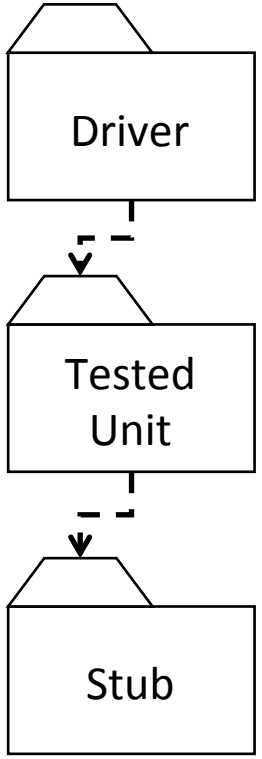
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- When Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

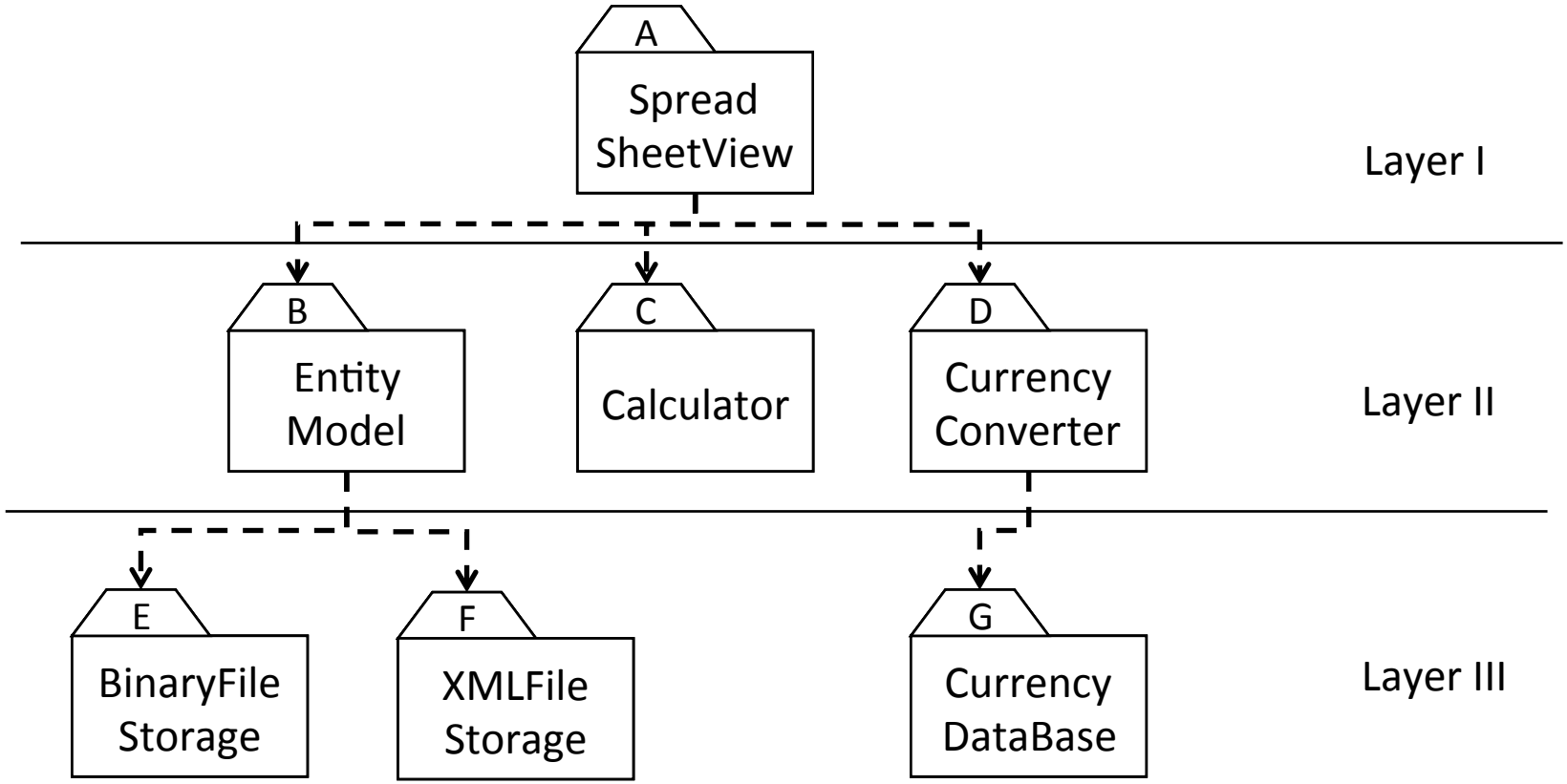
Test Stubs and Drivers

- Test driver
  - simulates the part of the system that calls the component under test
  - a component, that calls the `TestedUnit`
  - controls the test cases

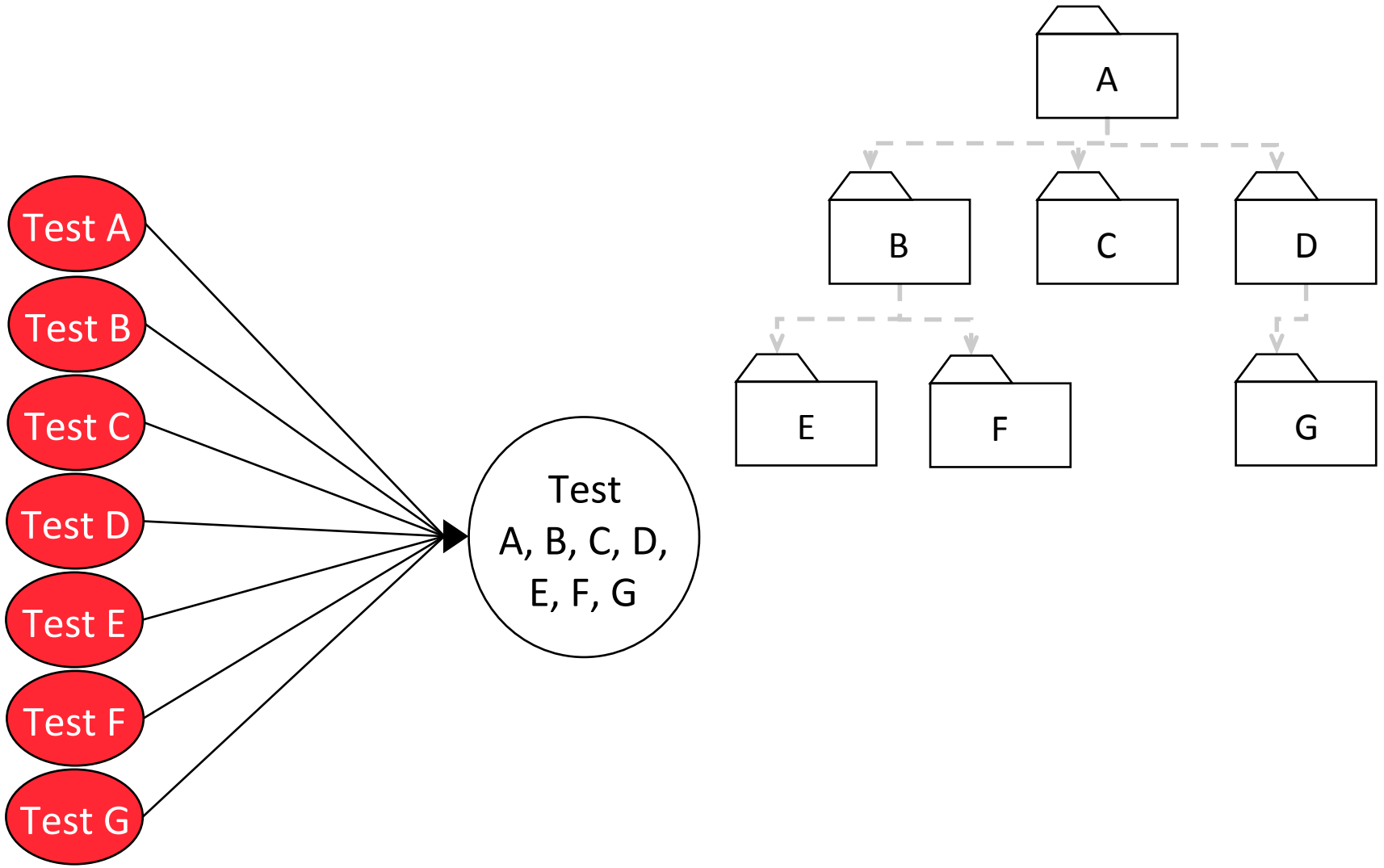


- Test stub
  - simulates a component that is being called by the tested component
  - provides / implements the same API as the component
  - a component, the `TestedUnit` depends on
  - partial implementation
  - returns fake values.

Example – 3 layered architecture



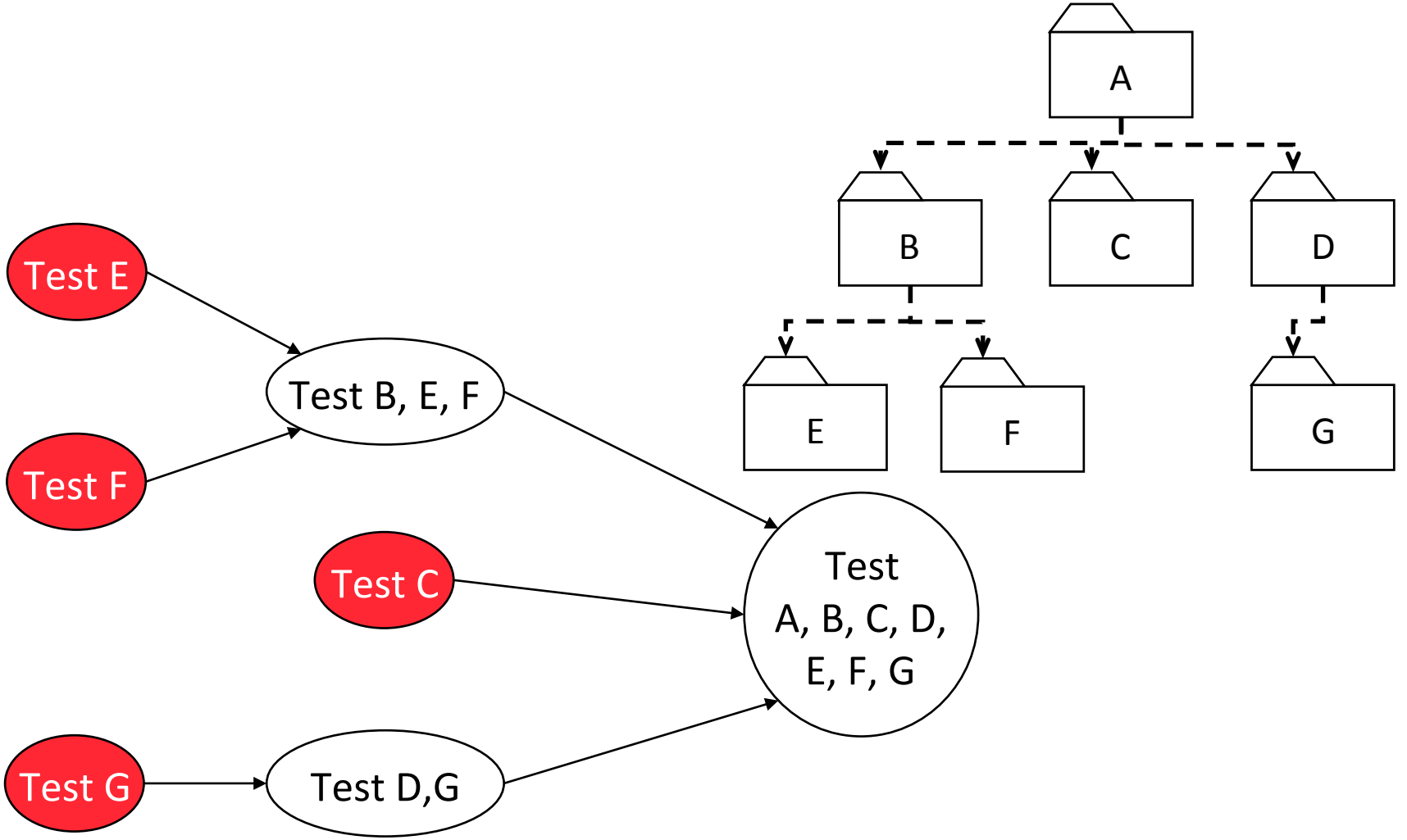
Big Bang Approach



Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the subsystems above this layer are tested that call the previously tested subsystems
- This is repeated until all subsystems are included.

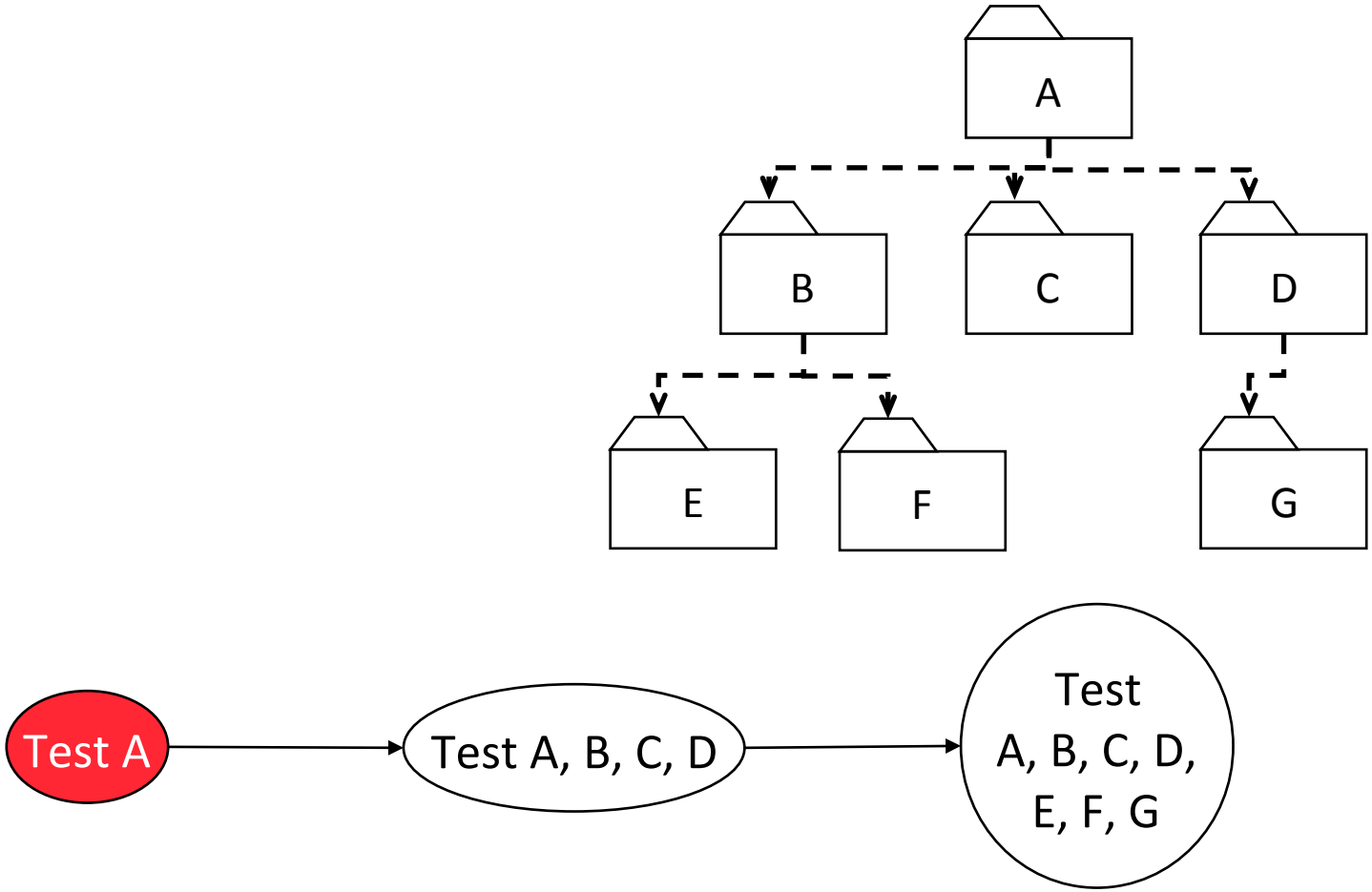
Bottom-up Testing Strategy



Top-down Testing Strategy

- Test the subsystems in the top layer first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the tests.

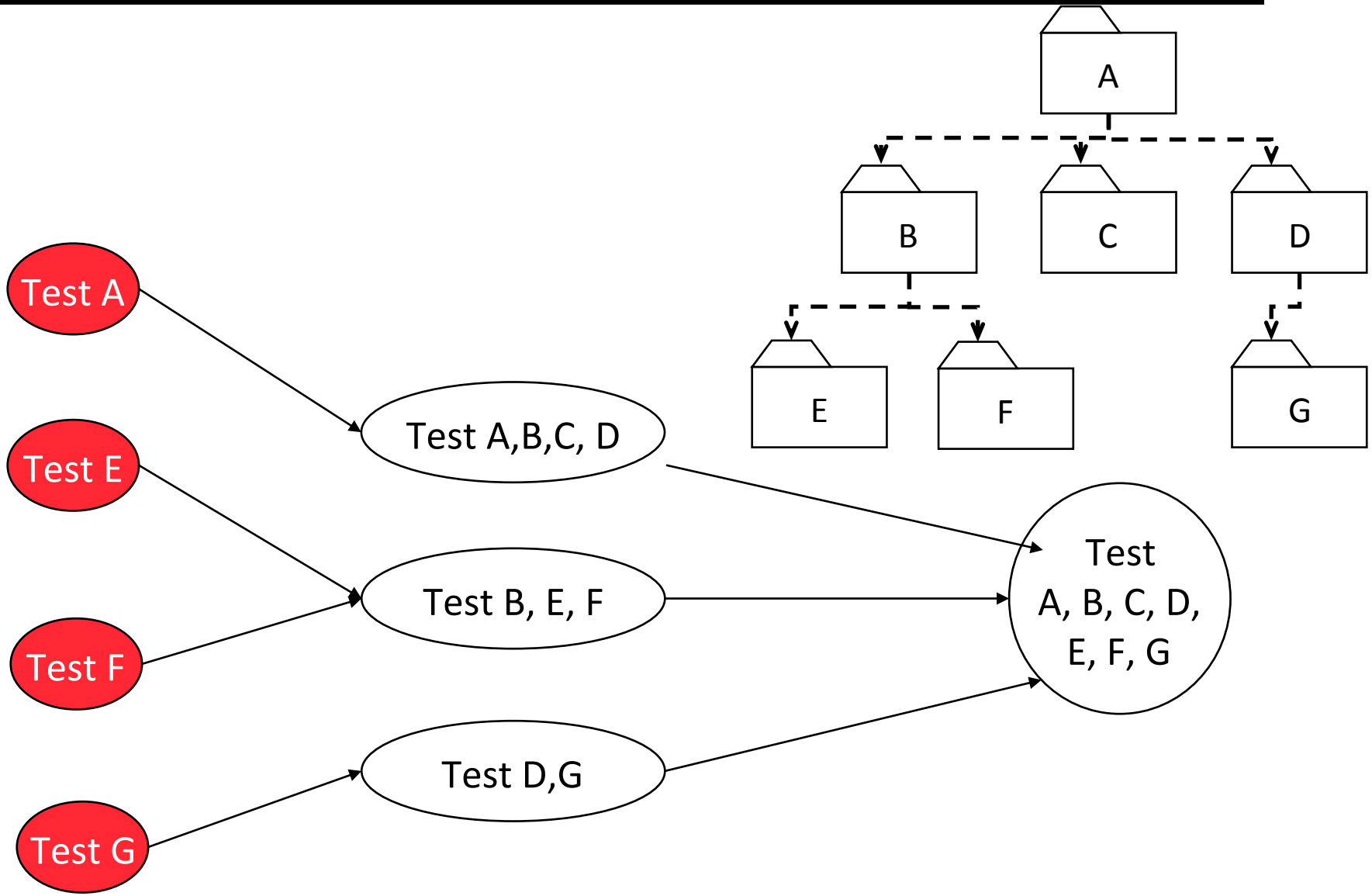
Top-down Testing Strategy



Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is viewed as having three layers
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
- Testing converges at the target layer.

Sandwich Testing Strategy



Pros and Cons: Top-Down Integration Testing

Pros:

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

Cons:

- Stubs are needed
- Writing stubs is difficult: Stubs must allow all possible conditions to be tested
- Large number of stubs may be required, especially if the lowest level of the system contains many methods
- Some interfaces are not tested separately.

Pros and Cons: Bottom-Up Integration Testing

- Pro
  - No stubs needed
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements
- Con:
  - Tests an important subsystem (the user interface) last
  - Drivers are needed.

Pros and Cons of Sandwich Testing

- Pro:
  - Top and bottom layer tests can be done in parallel
- Con:
  - Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy.

Typical Integration Questions

- Do all the software components work together?
- How much code is covered by automated tests?
- Were all tests successful after the latest change?
- What is my code complexity?
- Is the team adhering to coding standards?
- Were there any problems with the last deployment?
- What is the latest version I can demo to the client?

Regression testing

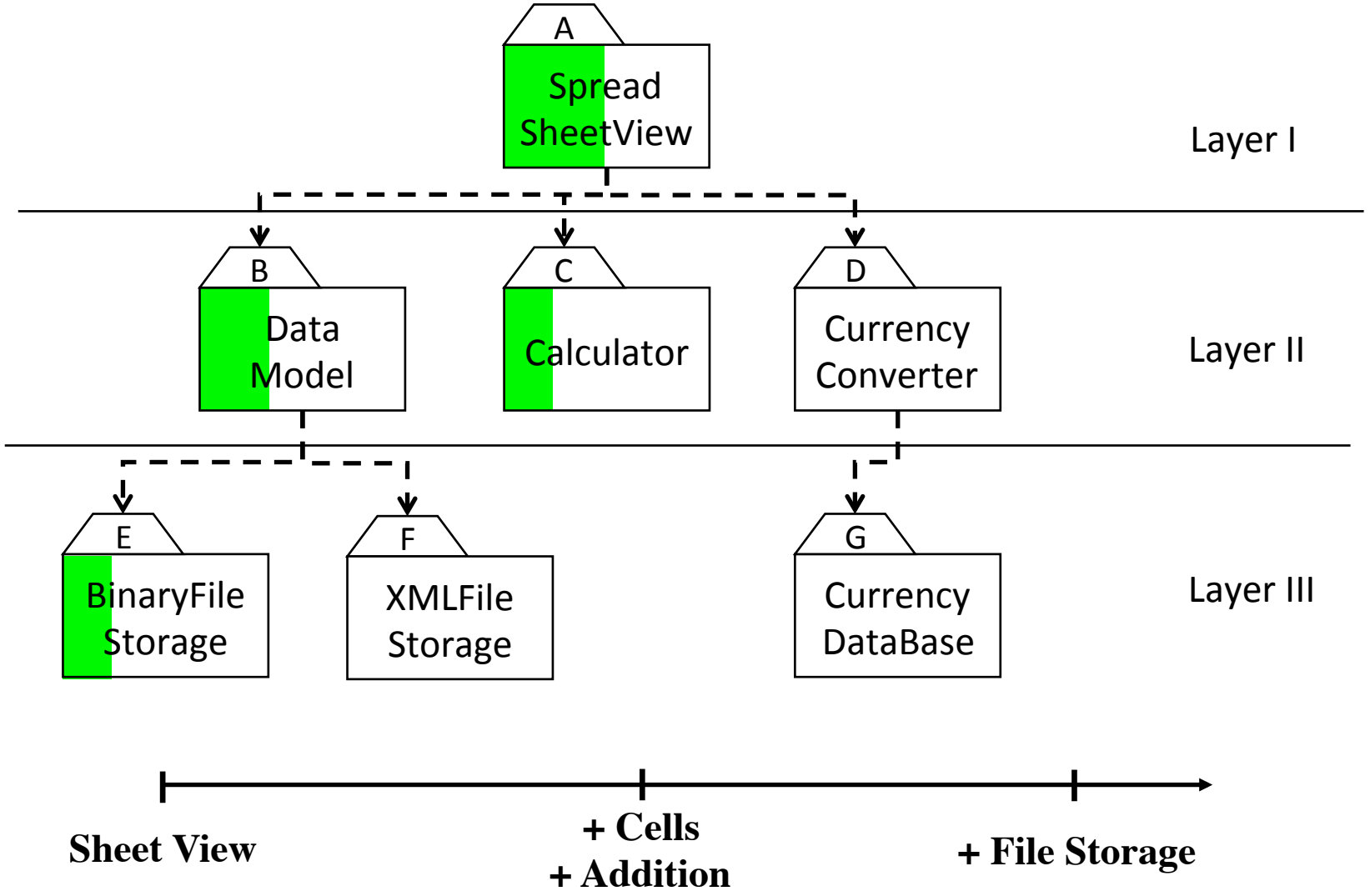
- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run 'successfully' before the change is committed.

Risks in Integration Testing Strategies

- Risk #1: The higher the complexity of the software system, the more difficult is the integration of its components
- Risk #2: The later integration occurs in a project, the bigger is the risk that unexpected faults occur
- Bottom up, top down, sandwich testing (Horizontal integration strategies) don't do well with risk #2
- Continous integration addresses these risks by building as early and frequently as possible
- Additional advantages:
  - There is always an executable version of the system
  - Team members have a good overview of the project status.

Continuous Integration (Testing)

Continuous Testing Strategy (Vertical Integration)



Definition Continuous Integration

Continuous Integration: A software development technique where members of a team *integrate* their work *frequently*, usually each person integrates at least daily, leading to multiple integrations per day.

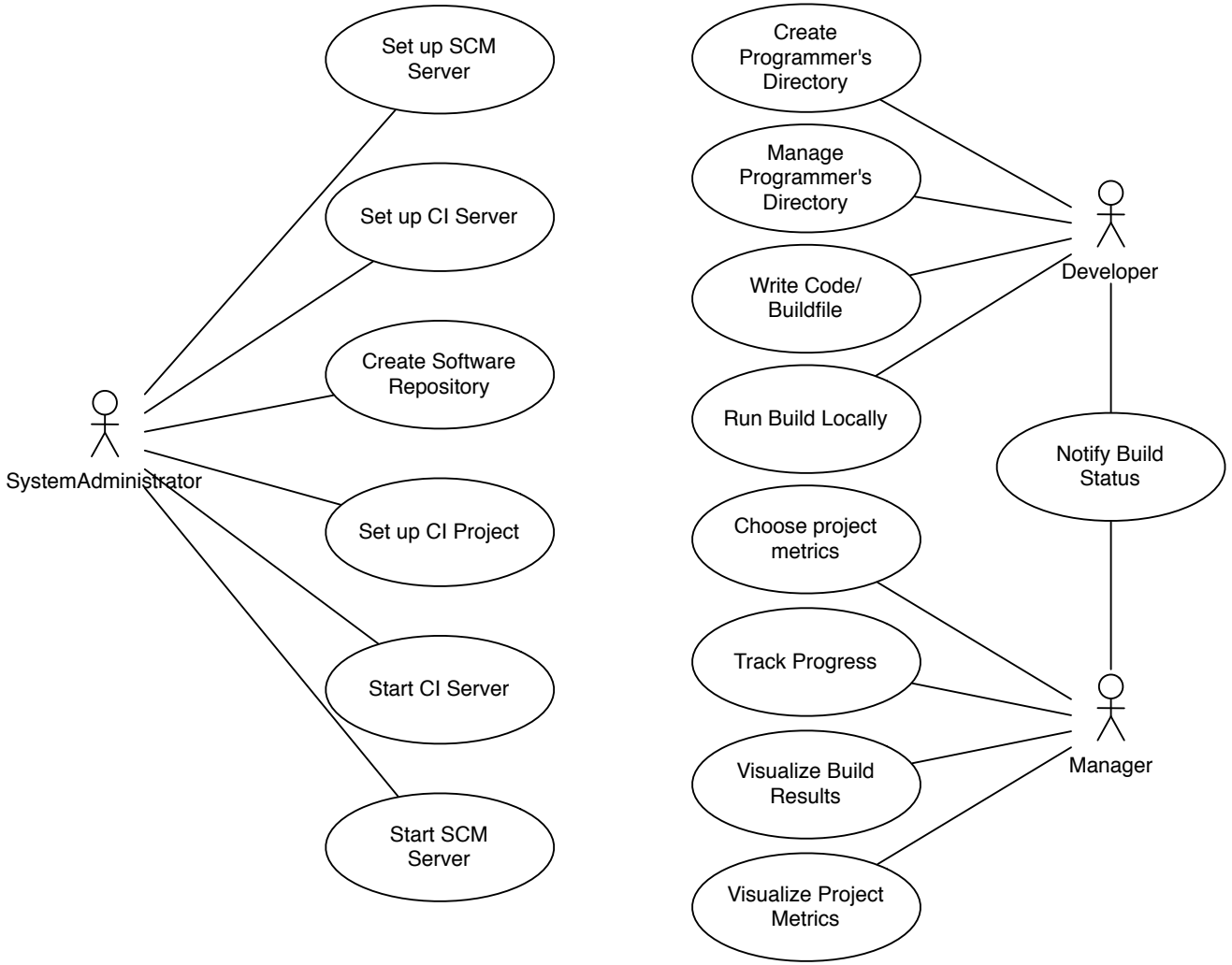
Each integration is verified by an *automated build which includes the execution of tests - regres* to detect integration errors as quickly as possible.

Source: http://martinfowler.com/articles/continuousIntegration.html

Modeling a Continuous Integration System

- Functional Requirements
  - Set up the scheduling strategy (poll, event-based)
  - Detect change
  - Execute build script when change has been detected
  - Run unit test cases
  - Generate project status metrics
  - Visualize status of the projects
  - Move successful builds into software repository
- Components (Subsystems)
  - Master Directory: Provides version control
  - Builder Subsystem: Executes build script when a change has been detected
  - Continuous Integration Server
  - Management Subsystem: Visualizes project status via Webbrowser
  - Notification Subsystem: Publishes results of the build via different channels (E-Mail Client, RSS Feed)

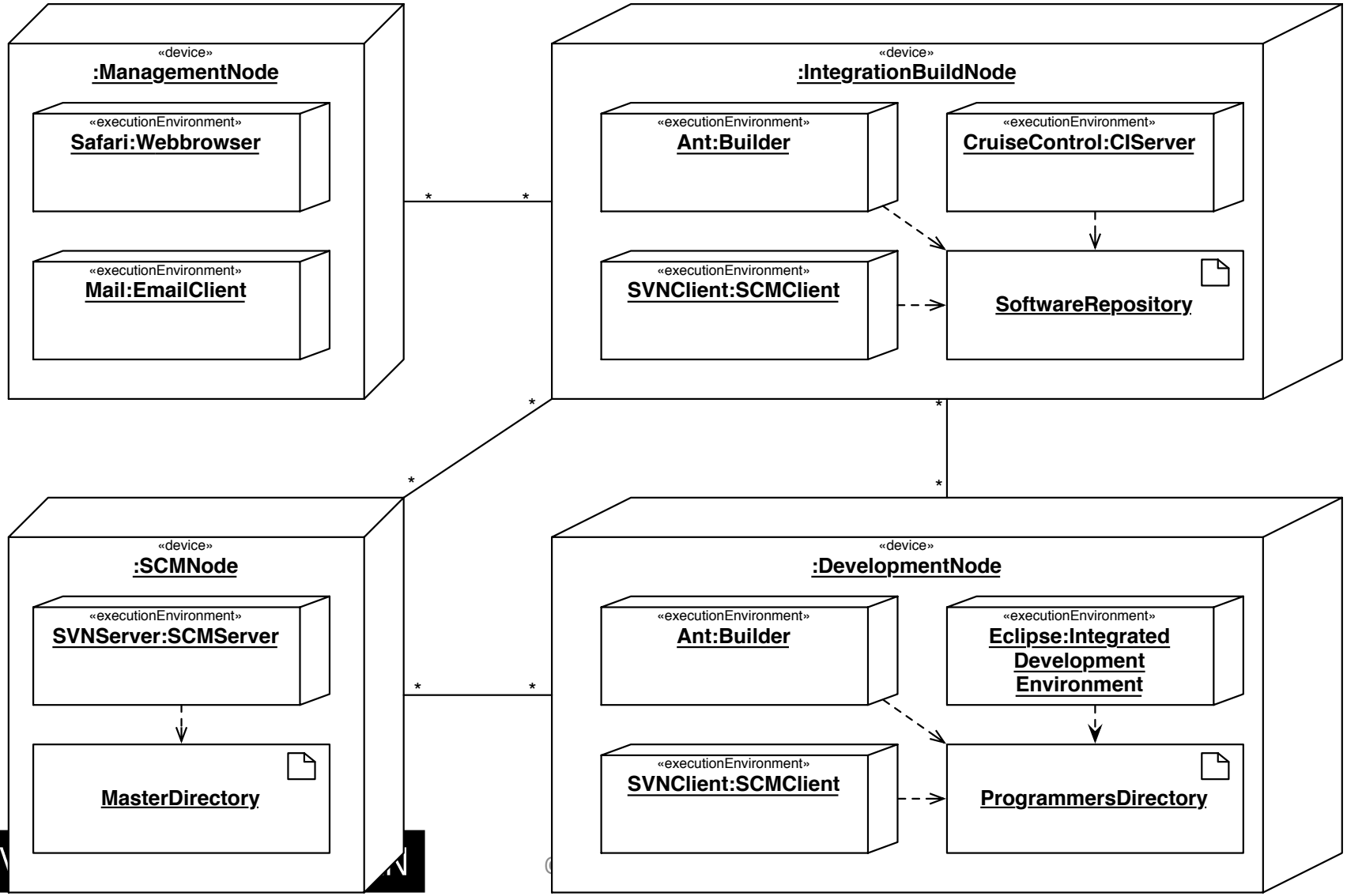
Analysis: Functional Model for Continuous Integration



Design of a Continuous Integration System

- Continuous build server
- Automated tests with high coverage
- Tool supported refactoring
- Software configuration management
- Issue tracking.

Design: Deployment Diagram of a Continuous Integration System

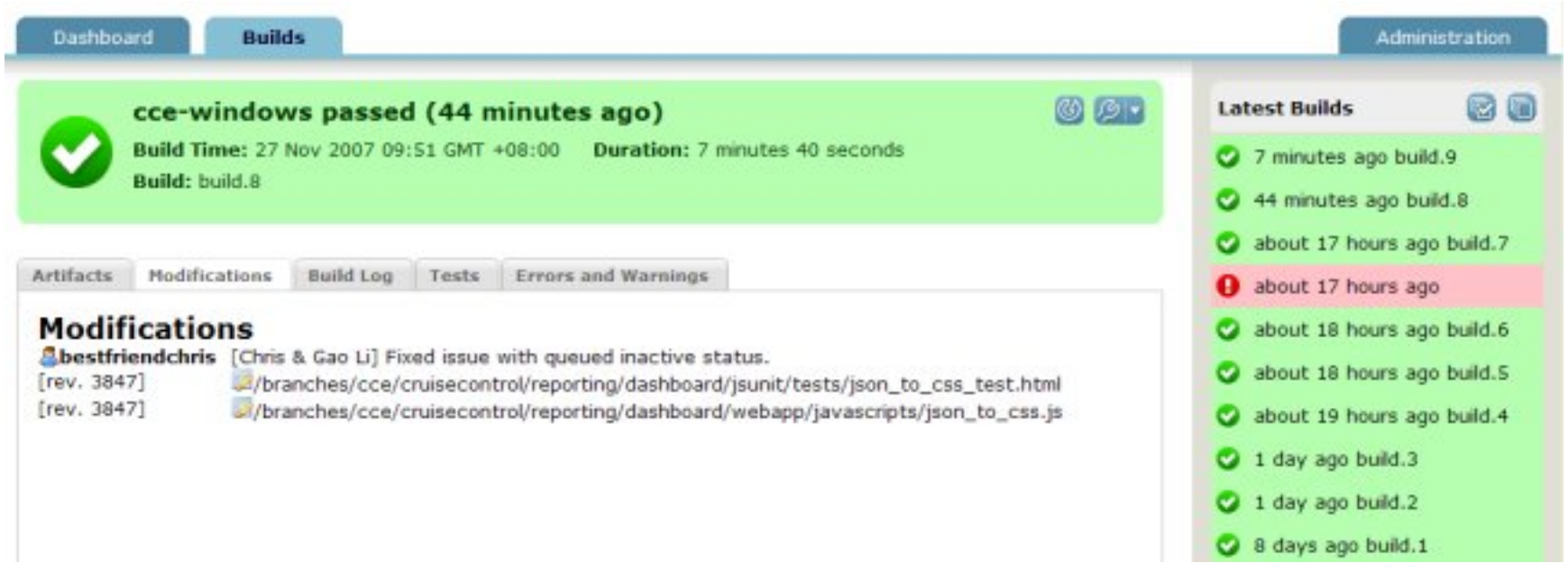


Examples of Continous Integration Systems

- CruiseControl and CruiseControl.NET
- Anthill
- Continuum
- Hudson
- and many more….

Feature comparison of continuous integration tools and frameworks:
http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix

Cruise Control Dashboard



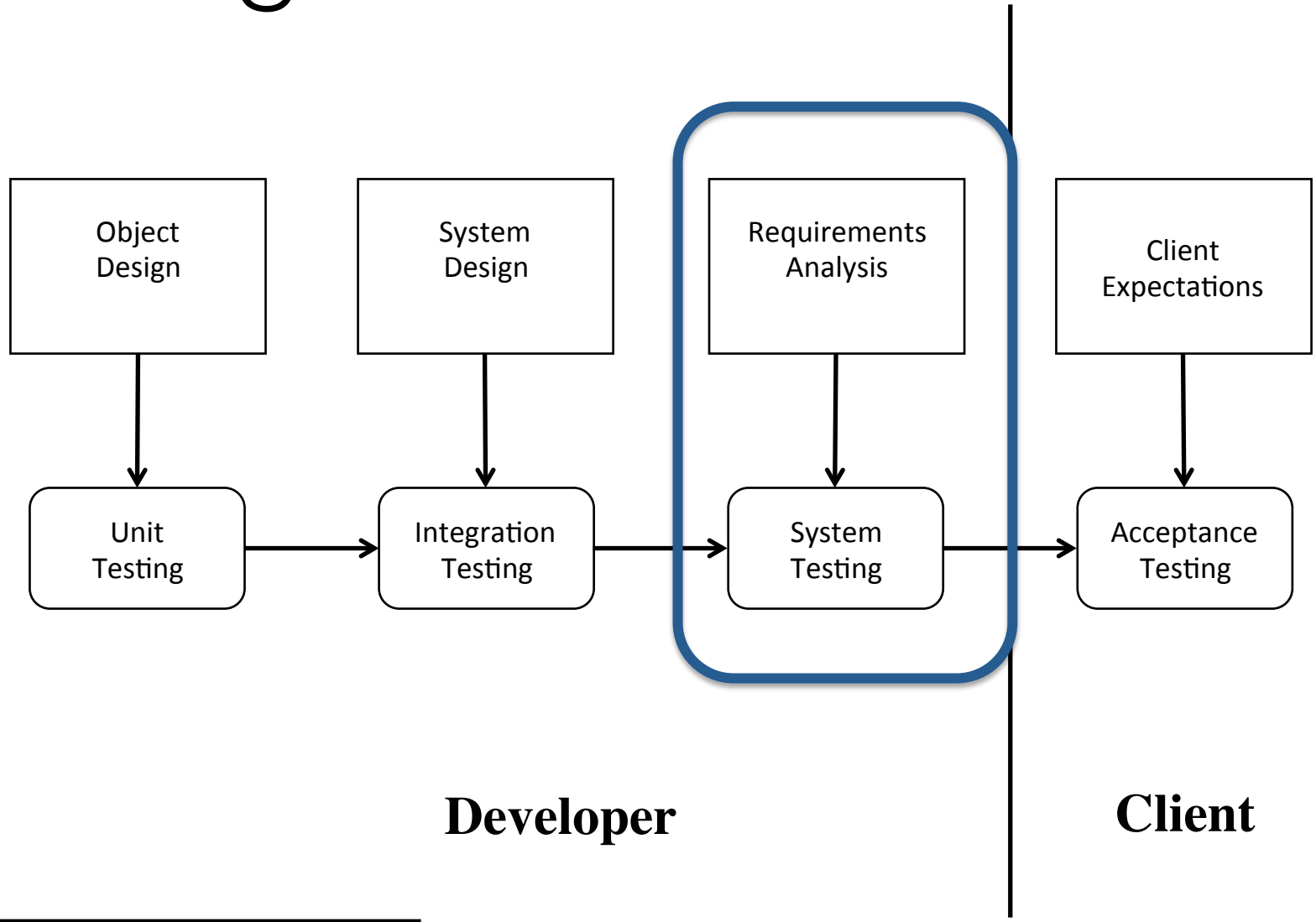
Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements*:* Define test cases that exercise all uses cases with the selected component
4. Test subsystem decomposition*:* Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify failures* with the (current) component *configuration*.

System Testing

Testing Activities and Models



System Testing

- Functional Testing
  - Validates functional requirements
- Performance Testing
  - Validates non-functional requirements
- Acceptance Testing
  - Validates clients expectations

Functional Testing

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

Performance Testing

Goal: Try to violate non-functional requirements

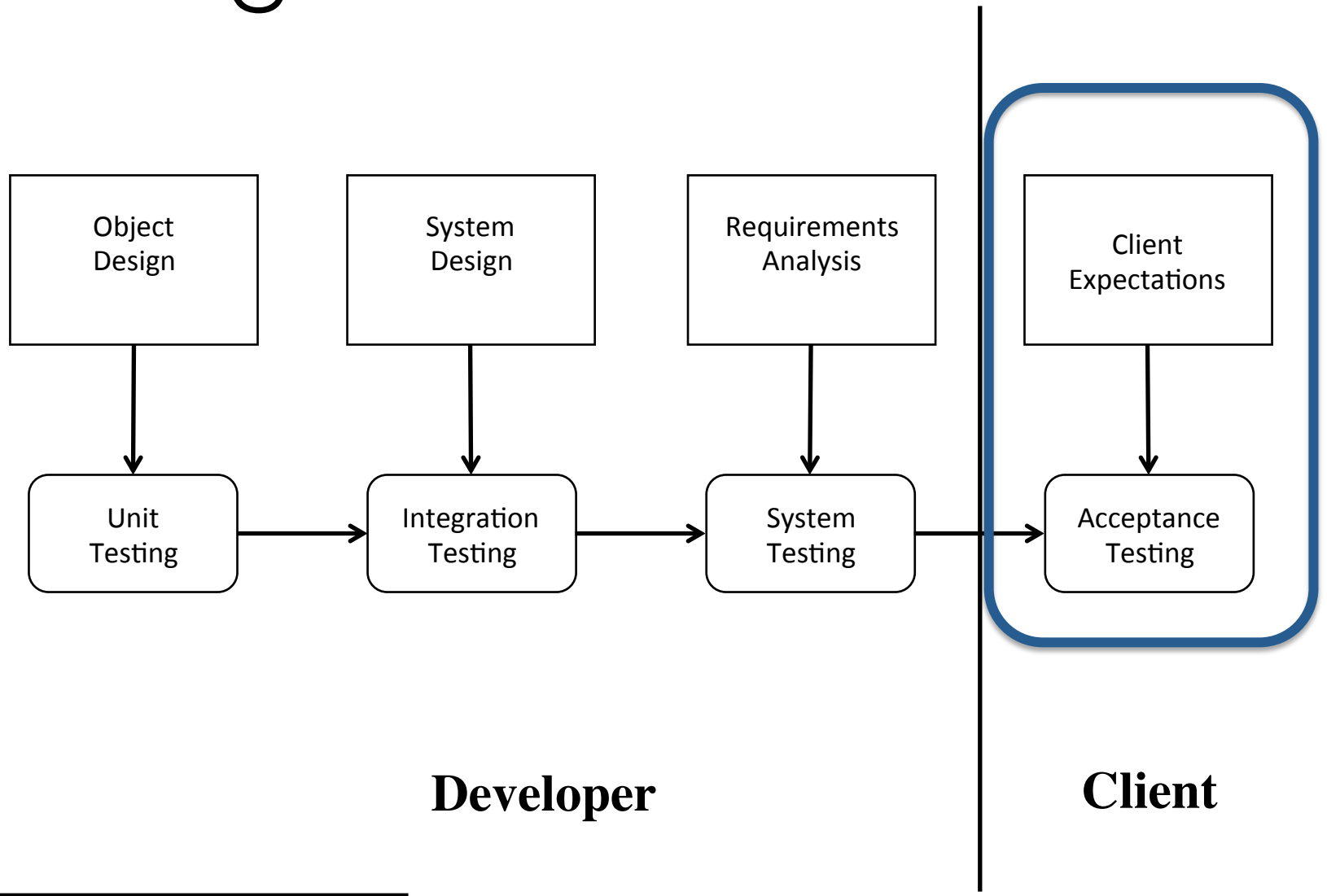
- Test how the system behaves when overloaded.
  - Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- Try unusual orders of execution
  - Call a receive() before send()
- Check the system's response to large volumes of data
  - If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?
  - Are typical cases executed in a timely fashion?

Types of Performance Testing

- Stress Testing
  - Stress limits of system
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Timing testing
  - Evaluate response times and time to perform a function
- Security testing
  - Try to violate security requirements
- Environmental test
  - Test tolerances for heat, humidity, motion
- Quality testing
  - Test reliability, maintain-ability & availability
- Recovery testing
  - Test system's response to presence of errors or loss of data
- Human factors testing
  - Test with end users.

Acceptance (Client) Testing

Testing Activities and Models



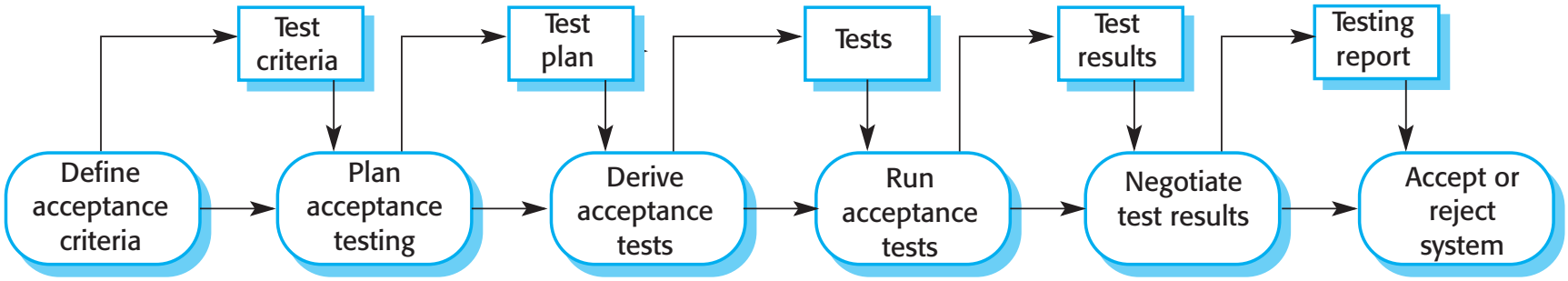
Client testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

The acceptance testing process



Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

Managing Testing

Test Cases

- Test case
  - a set of input data and expected results that exercise a component

Table 11-1 Attributes of the class `TestCase`.

Attributes	Description
`name`	Name of test case
`location`	Full path name of executable
`input`	Input data or commands
`oracle`	Expected test results against which the output of the test is compared
`log`	Output produced by the test

Managing Testing

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

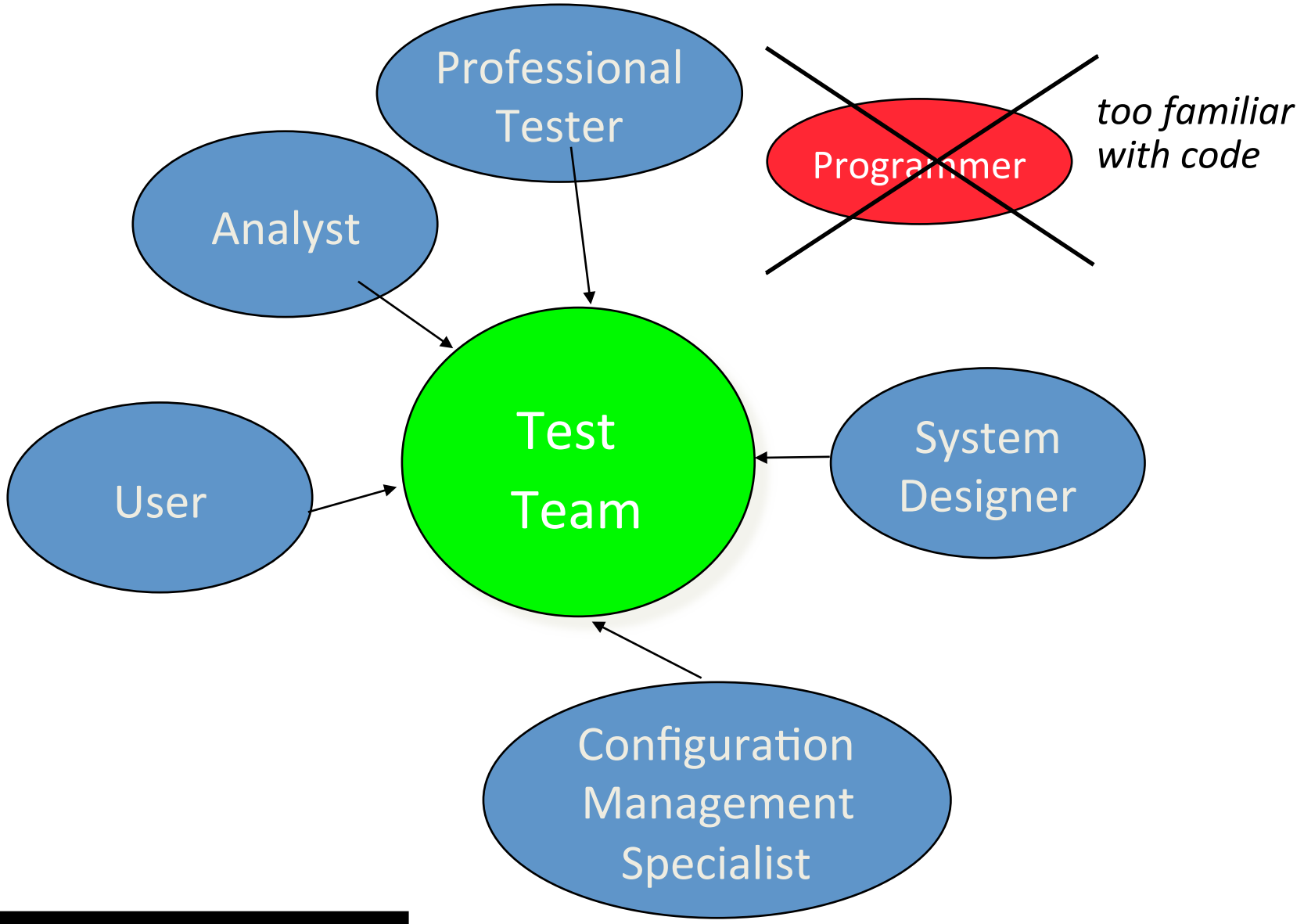
Execute the tests

Evaluate the test results

Change the system

Do regression testing

The Test Team



The 4 Testing Steps

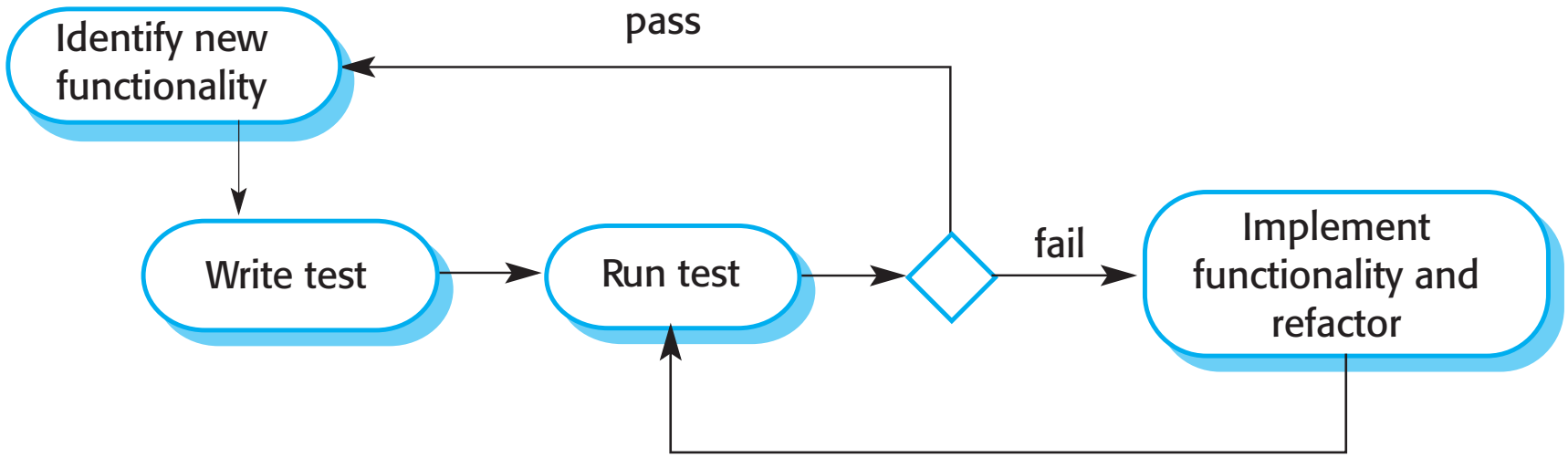
1. Select what has to be tested
   - Analysis: Completeness of requirements
   - Design: Cohesion
   - Implementation: Source code
2. Decide how the testing is done
   - Review or code inspection
   - Proofs (Design by Contract)
   - Black-box, white box,
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)
3. Develop test cases
   - A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured
4. Create the test oracle
   - An oracle contains the predicted results for a set of test cases
   - The test oracle has to be written down before the actual testing takes place.

Test Driven Development

Test-driven development

- Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- Tests are written before code and ‘passing’ the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development



Benefits of test-driven development

- Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

Test Documentation

- Test Plan
- Test Case Specification
- Test Incident Report
- Test Report Summary

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

Key Points in Software Testing

Key points I

- Testing can only show the presence of errors in a program.
  - It cannot demonstrate that there are no remaining faults.
- Development testing is the responsibility of the software development team.
  - A separate team should be responsible for testing a system before it is released to customers.
- Development testing includes
  - unit testing, in which you test individual objects and methods
  - component testing in which you test related groups of objects
  - system testing, in which you test partial or complete systems.

Key points II

- When testing software, you should try to 'break' the software
  - using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- Wherever possible, you should write **automated tests**.
  - The tests are embedded in a program that can be run every time a change is made to a system.
- You should establish a **continuous integration** testing setup
- **Test-first development** is an approach to development where tests are written before the code to be tested.
- **Scenario testing** involves inventing a typical usage scenario and using this to derive test cases.
- **Acceptance testing** is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.

This Lecture

- Literature
  - [OOSE] ch. 11
  - [SE9] ch. 8 (+24)

- Introduction to Software Testing
- Testing Terminology
- Testing Activities
  - Unit / Component Testing
  - Integration Testing
  - System Testing
  - Client / Acceptance Testing
- Managing Testing
  - Test Cases
  - Test Teams
  - Test Driven Development
  - Documenting Testing