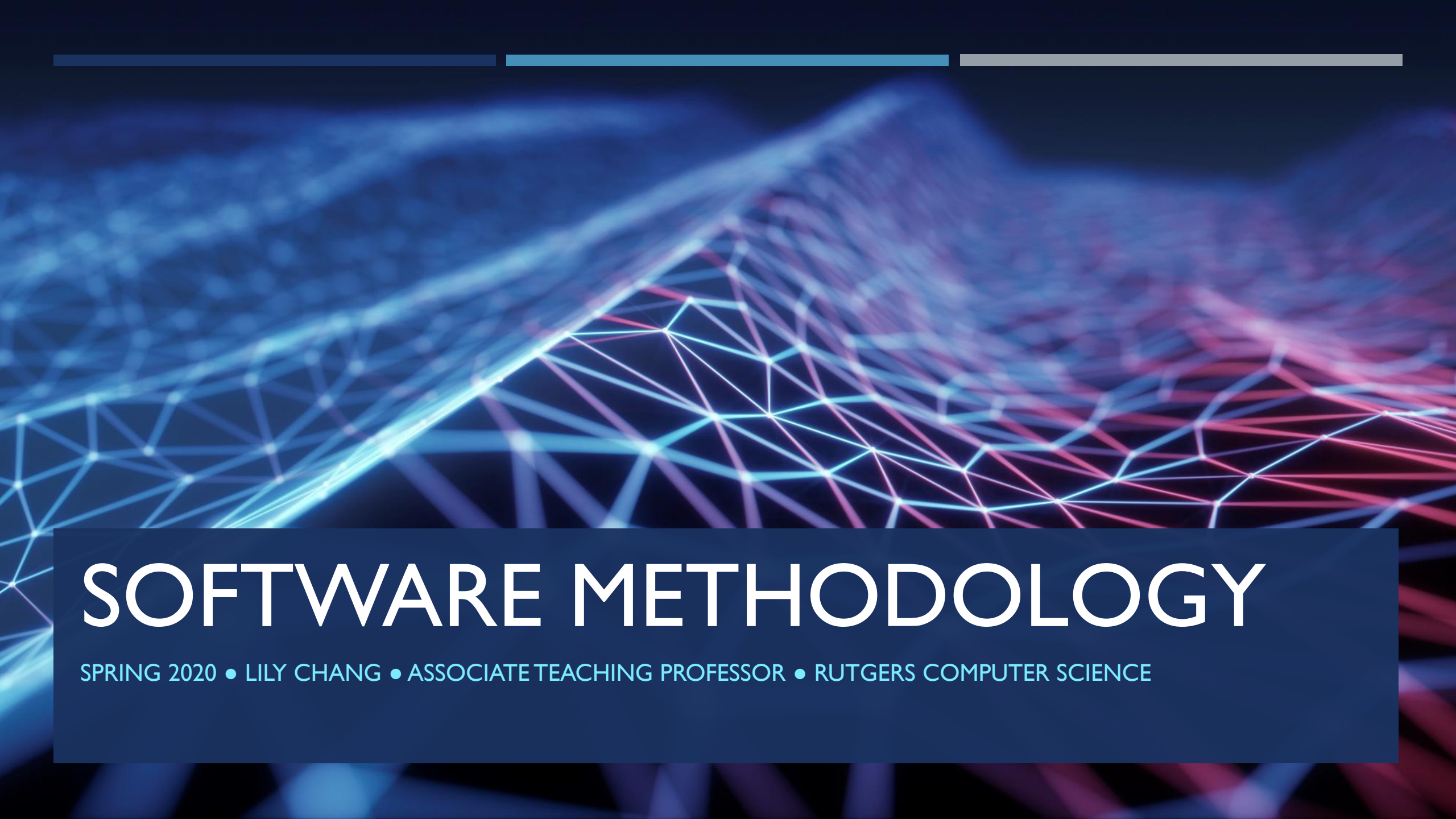


SOFTWARE METHODOLOGY

SPRING 2020 • LILY CHANG • ASSOCIATE TEACHING PROFESSOR • RUTGERS COMPUTER SCIENCE

JUNIT TESTING

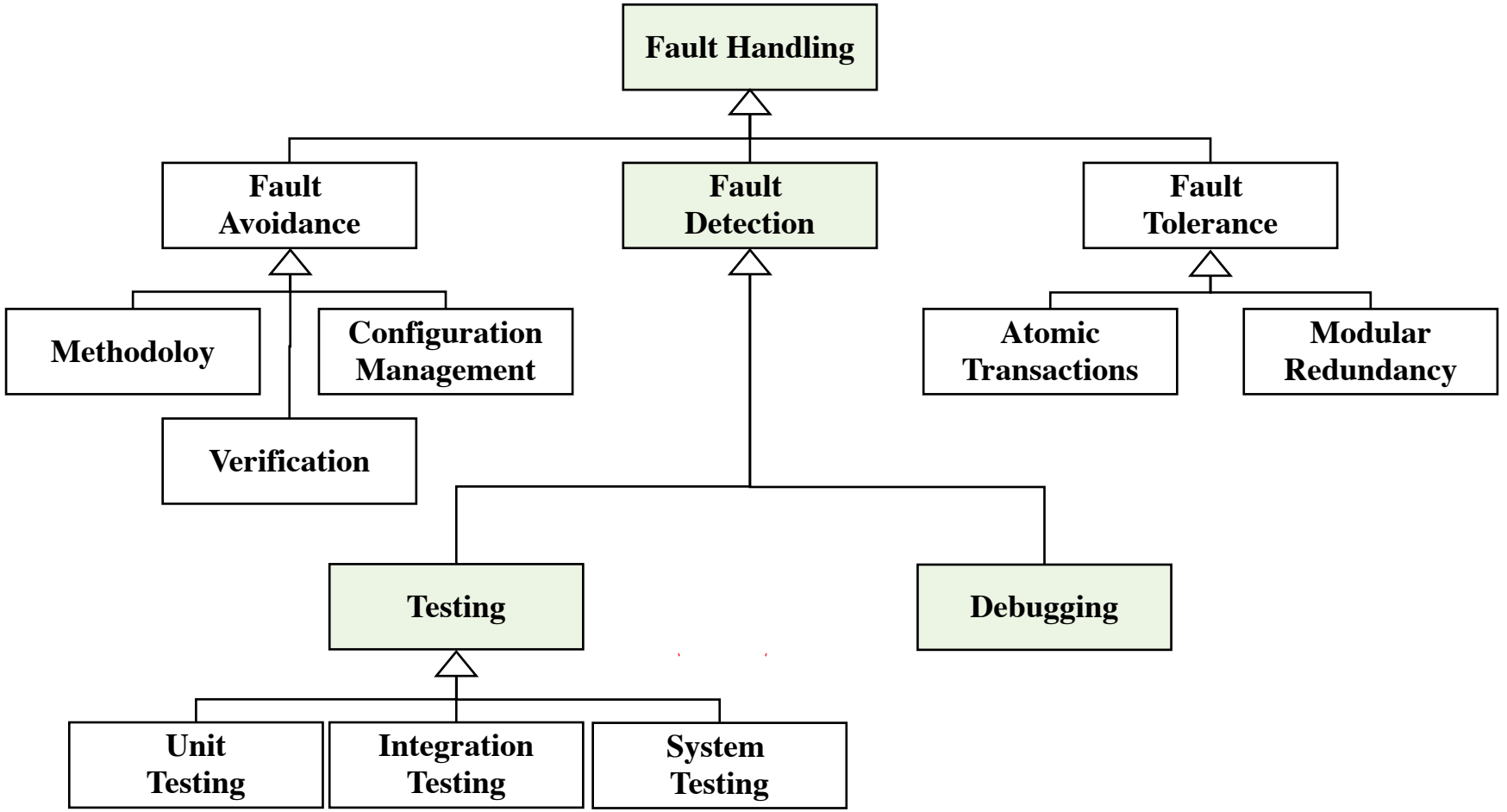
COST OF SOFTWARE FAILURE

- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason? Reuse of autopilot software
- 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 due to overdoses of radiation)
  - Reason: Bad event handling in the GUI program
- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem
- Boeing MAX 737 – lost hundreds of human lives
- Volvo recalled 59,000 cars over software fault that can temporarily shut down the engine

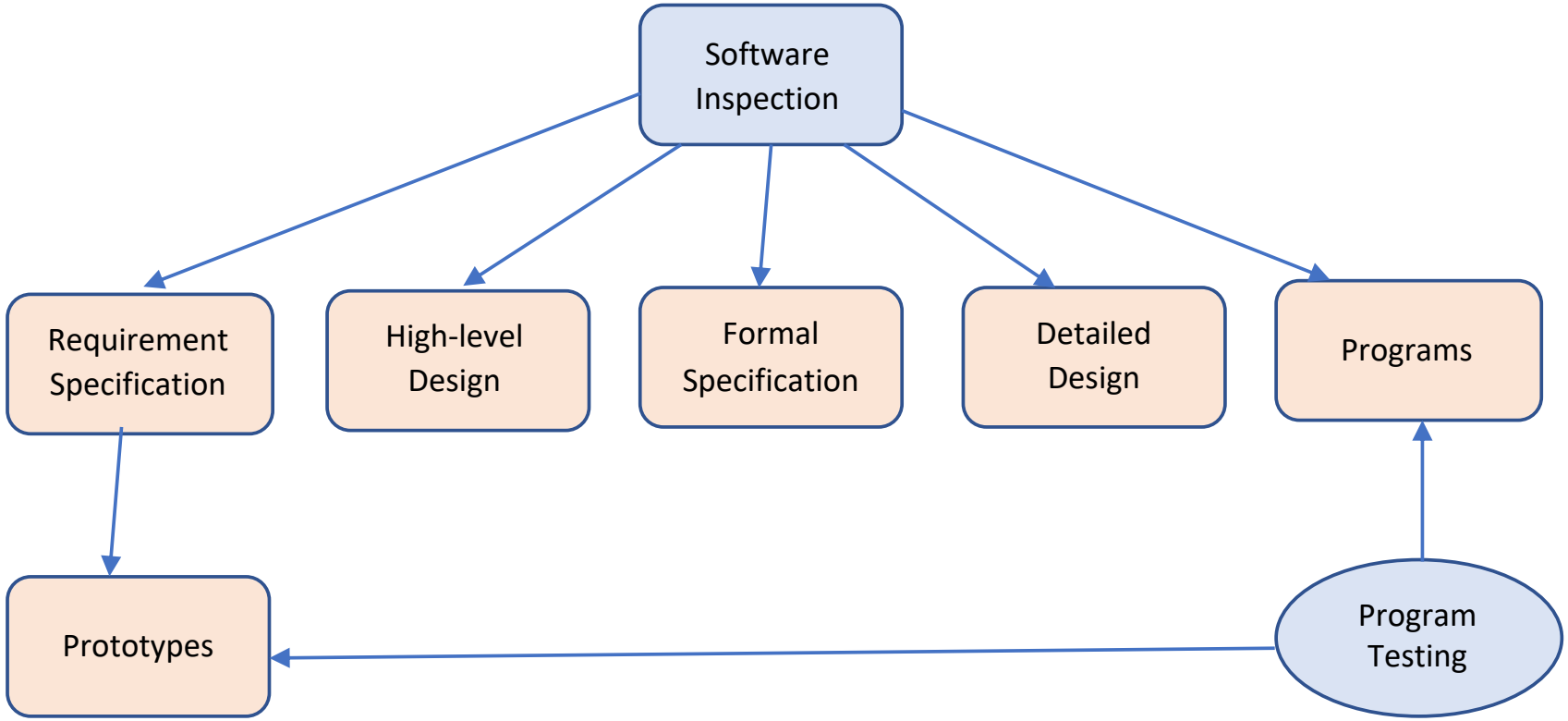
TERMINOLOGY

- Reliability: the probability that a software system WILL NOT cause system failure for a specified time under specified conditions [IEEE Std. 982-1989]
- 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 (or defect, or bug): The mechanical or algorithmic cause of an error
- Testing: systematic attempt to find faults in a planned way in the implemented software

FAULT HANDLING TECHNIQUES



STATIC AND DYNAMIC APPROACH FOR SOFTWARE TESTING



SOFTWARE INSPECTION PROCESS – STATIC APPROACH

SOFTWARE TESTING – DYNAMIC APPROACH

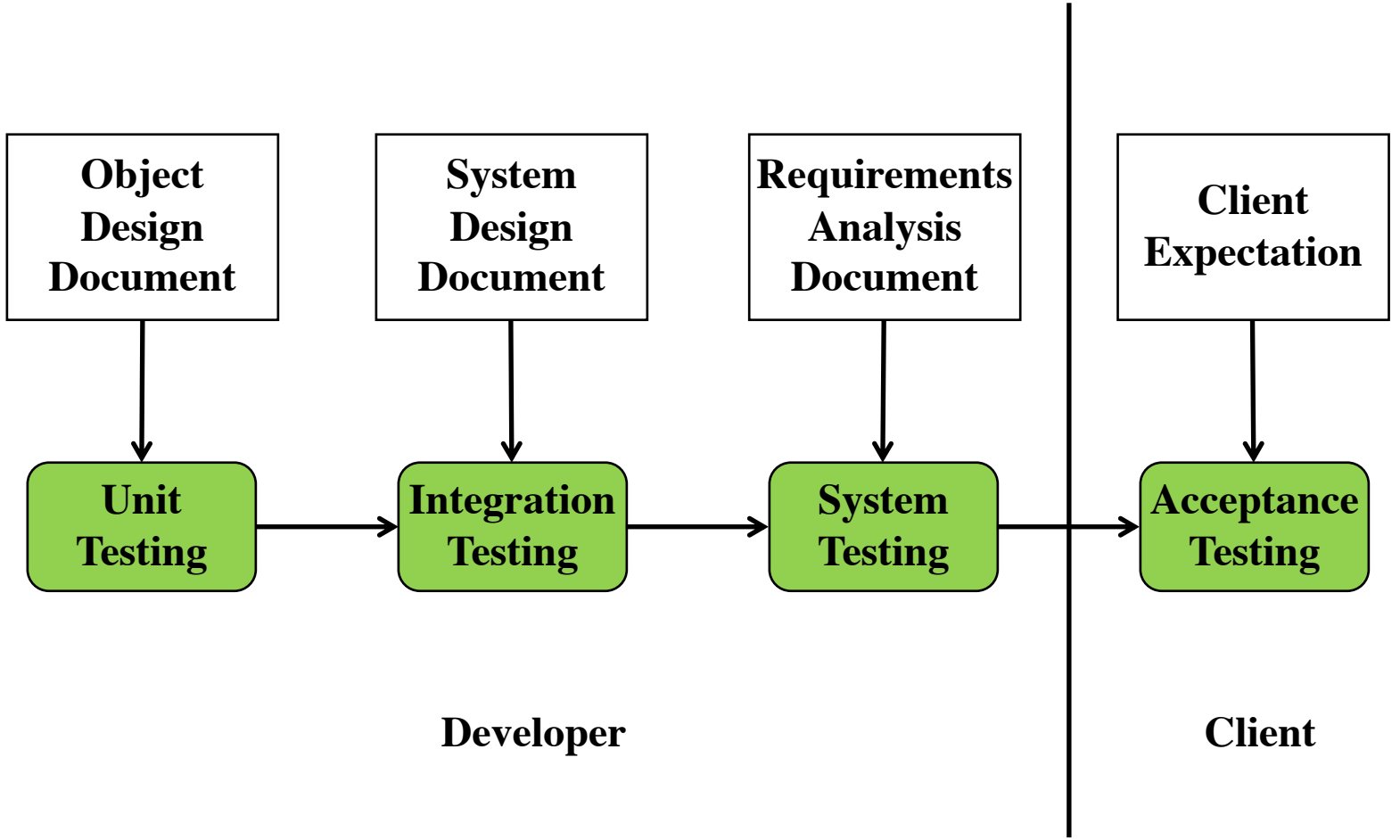
- The software system is executed
- The process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system
- The attempt to show that the implementation of the system is inconsistent with the system models
- The goal is to design tests that exercise defects in the system and to reveal problems
- Software Testing is aimed at breaking the system!

SOFTWARE TESTING PLAN

- 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!!

TESTING ACTIVITIES



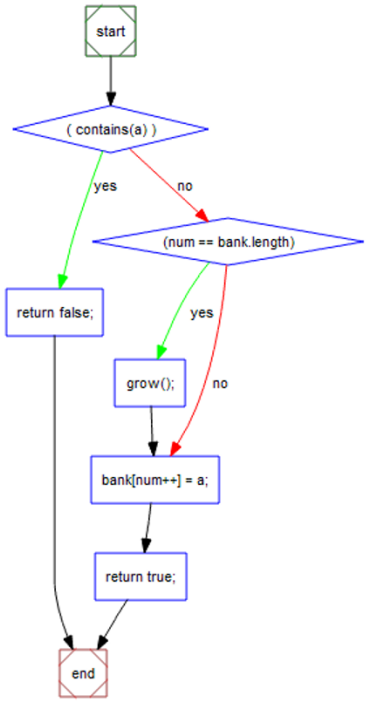
UNIT TESTING

- Individual component (class or subsystem)
- Carried out by developers
- Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality

UNIT TESTING TECHNIQUES

- Black-box testing

  - Functional testing
  - Does not focus on the implementation details
- White-box testing
  - Structural testing
  - Focus on the control structure and coverage of the code being exercised
    - Code coverage, Branch coverage, Condition coverage, Path coverage



BLACK-BOX TESTING

- Required Information: only requirement specification
- Independent of the implementation; test design can be in parallel with implementation
- Focus on the I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle (expected test results)
- Goal – Reduce number of test cases by equivalence class partitioning
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

BLACK-BOX TESTING – TEST CASE SELECTION

a) Input is valid across range of values

- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid if it is a member of a discrete set

- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values

C) Boundary value analysis

BLACK BOX TESTING – AN EXAMPLE

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

Representation for month:
1: January, 2: February, …., 12: December

Representation for year:
1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of getNumDaysInMonth() method?

EXAMPLE—EQUIVALENCE CLASSES

- For the month parameter,
  - Valid – 3 equivalence classes
    - Months with 31 days, JAN, MAR, MAY, JUL, AUG, OCT, DEC
    - Months with 30 days, APR, JUN, SEPT, NOV, and
    - February can have 28 or 29 days
  - Invalid – 0, non-positive integers and integers larger than 12
- For the year parameter,
  - Valid – 2 equivalence classes: Leap years and non-leap years
  - Invalid: 0 and negative integers

EXAMPLE – TEST CASES SELECTION

Equivalence class	Value for month input	Value for year input
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap years	6 (June)	1904
Months with 28 or 29 days, non-leap years	2 (February)	1901
Months with 28 or 29 days, leap years	2 (February)	1904

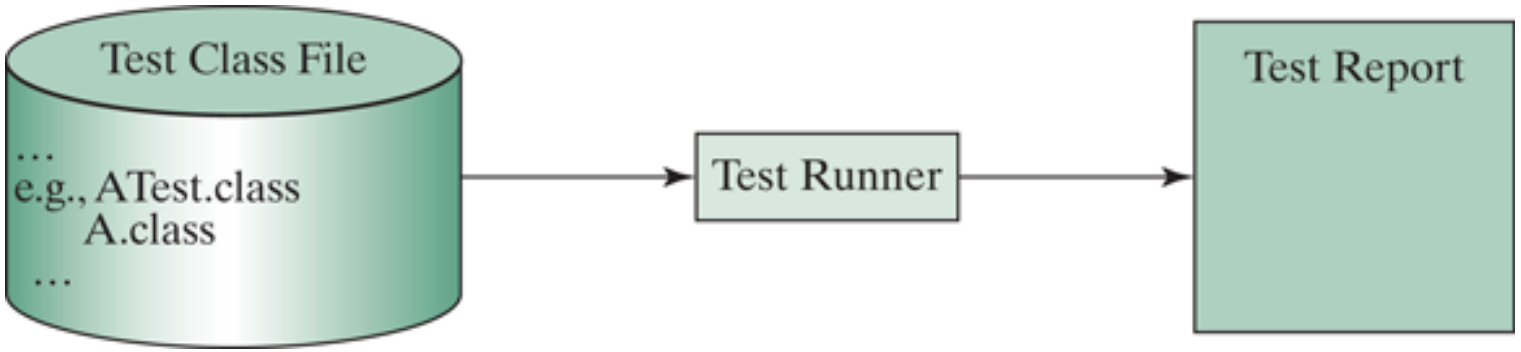
BOUNDARY TESTING

- Special case of equivalence testing focuses on the conditions at the boundary of the equivalence classes
- Select elements from the "edges" of the equivalence class

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
0 and Non-positive invalid month	0	1291
Positive invalid months	13	1315

JUNIT TEST FRAMEWORK

- Software testing is expensive and tedious, thus use CASE (Computer Aided Software Engineering) tools as much as possible
  - Automate the tests by implementing test cases, so they are repeatable
  - Regression testing, refactoring, software change
- JUnit is the de facto framework for testing Java programs.
- JUnit is a third-party open-source library packed in a jar file, which contains a tool called test runner to run test programs



JUNIT TEST FRAMEWORK

- Eclipse and IntelliJ incorporate the JUnit into their IDE
  - See the demo video for the steps of creating tests in Eclipse and IntelliJ
- Resources and documentation
  - https://junit.org/junit4/
  - https://junit.org/junit5/

USEFUL ASSERT CLASSES IN JUNIT

```
assertTrue(boolean condition)
assertFalse(boolean condition)
assertNull(Object testobject)
assertEquals(Object expected, Object actual) //according to equals() method
assertEquals(int expected, int actual); //according to ==
assertEquals(double expected, double expected); //less than or equal to the tolerance value
assertSame(Object expected, Object actual); //if refer to the same object in memory
```

CREATING THE TEST SUITE – JUNIT RUNNER CLASS

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

/**
 The following annotation specifies the test runner to use is
 org.junit.runners.Suite
 */
@RunWith(Suite.class)
/**
 The following annotation run all Java .class listed in the braces.
 Use comma to separate different .class files.
 */
@Suite.SuiteClasses({
    ComplexTest.class,
    PostfixEvaluatorTest.class,
    StackTest.class})

public class TestSuite {
//remains empty, used only as a holder for the above annotations.
}
```

FIVE STEPS OF UNIT TESTING OO SOFTWARE

1. Create an object and select a method to execute
2. Select values (test cases) for the input parameters of the method
3. Compute the expected values to be returned by the method
4. Execute the selected method on the created object using the selected input values
5. Verify the result of executing the method

THANK YOU