Software Engineering

Software Testing

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with a subject like: *SE345-Andy: On What …*

Download from c.program@yeah.net

/文件中心/网盘/SoftwareEngineering24S

Topics

- Overview of Software Testing
  1. Make a plan for systematic testing
  2. Write and run tests; test automation
- Designing Tests for High Coverage
- Practical Aspects of Unit Testing
- Integration and System Testing
- Security Testing

Do I Need to Do Testing?

- Everyone who develops software does testing—that's unavoidable
- The only question is whether testing is conducted haphazardly by random trial-and-error, or **systematically**, with a plan
- Why it matters? —The goal is to try as many "critical" input combinations as possible, given the **time and resource constraints**
  - i.e., to achieve as **high coverage** of the input space as is practical, while testing first the "highest-priority" combinations
  - Key issue: strategies for identifying the "highest priority" tests

Overview of Software Testing

- "Testing shows the presence, not the absence of bugs." —Edsger W. Dijkstra
- A **fault**, also called "defect" or "bug," is an erroneous hardware or software element of a system that can cause the system to fail
- **Test-Driven Development (TDD)**
  - Every step in the development process must start with a plan of how to verify that the result meets a goal
  - The developer should not create a software artifact (a system requirement, a UML diagram, or source code) unless they know how it will be tested
- A **test case** is a particular choice of input data to be used in testing a program and the expected output or behavior
- A **test** is a finite collection of test cases

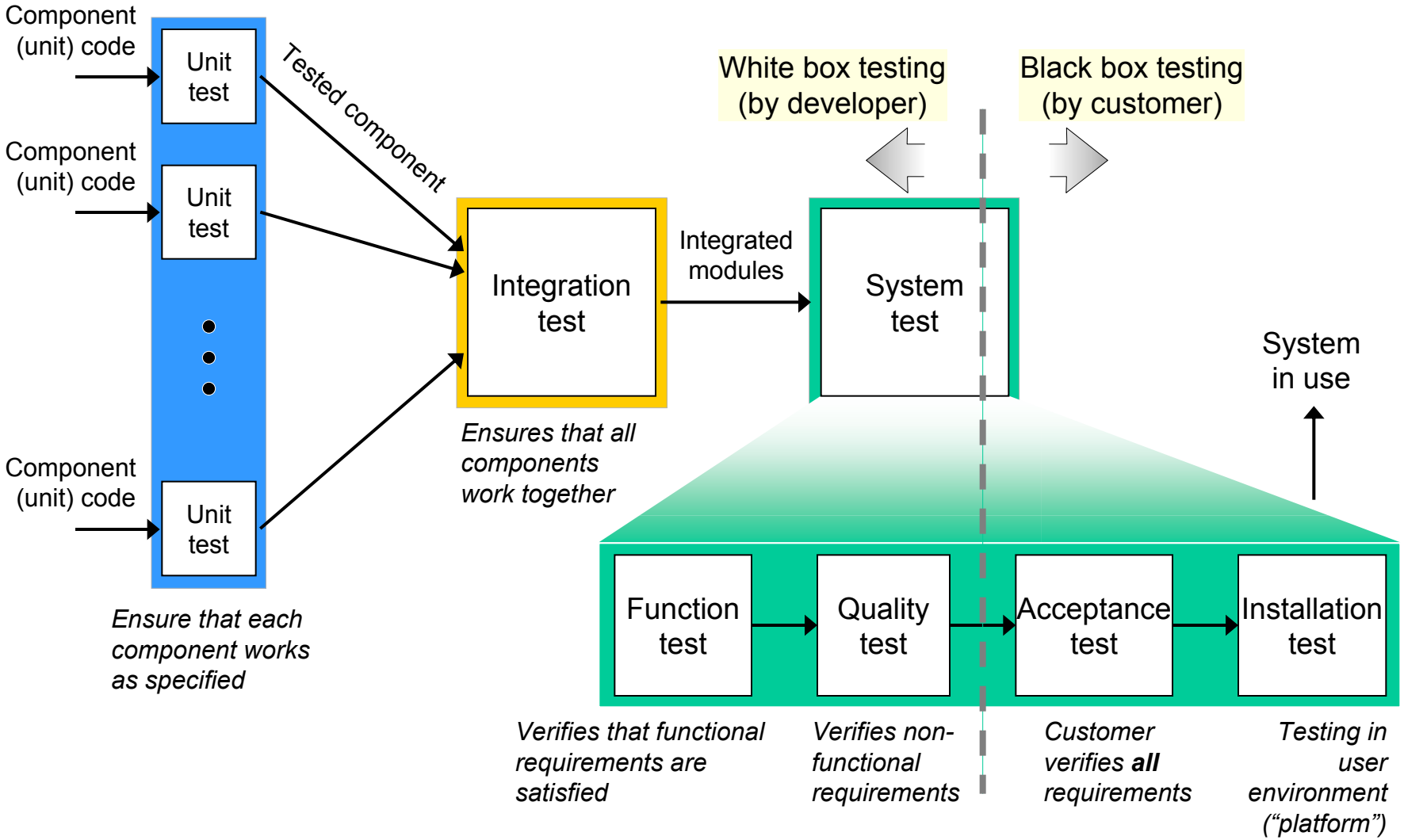
White-box testing exploits structure within the program (assumes program code available)
Black-box testing explores input space of functionality defined by an interface specification

Why Testing is Hard

- Any nontrivial system cannot be completely tested
  - Example: testing 4-digit numeric keycodes
    - You never know with certainty that all 4-digit codes work as expected unless you try all
    - But, what happens with "incomplete" (3-digit, 2-digit) codes? Will all of them timeout to reset state?
    - Or, will all tuples (pairs, triplets, etc.) of successive incomplete codes timeout to the reset state?
- Our goal is to find faults as cheaply and quickly as possible.
  - Ideally, we would design a single "right" test case to expose each fault and run it
- In practice, we have to run many "unsuccessful" test cases that do not expose any faults
- A key tradeoff of testing:
  - testing as many potential cases as possible (high degree of "test coverage") while keeping the economic costs limited
- Underlying idea of software testing:
  - the *correct behavior* on "critical" test cases is representative of correct behavior on *untested parts* of the state space

Logical Organization of Testing

(Usually **not** done in a linear step-by-step order and completed when the last step is reached!)



Acceptance Tests – Safe Home Access Examples

(“black box” testing: focus on the external behavior)

[Recall Section 2.2: Requirements Engineering]

Input data

- Test with the valid key of a current tenant on his/her apartment (pass)

Expected result

- Test with the valid key of a current tenant on someone else’s apartment (fail)
- Test with an invalid key on any apartment (fail)
- Test with the key of a removed tenant on his/her previous apartment (fail)
- Test with the valid key of a just-added tenant on his/ her apartment (pass)

Example: Test Case for Use Case

[Recall Section 2.3.3: Detailed Use Case Specification]

Test-case Identifier:	TC-1
Use Case Tested:	UC-1, main success scenario, and UC-7
Pass/fail Criteria:	The test passes if the user enters a key that is contained in the database, with less than a maximum allowed number of unsuccessful attempts
Input Data:	Numeric keycode, door identifier

Test Procedure:	**Expected Result:**
Step 1. Type in an incorrect keycode and a valid door identifier	System beeps to indicate failure; records unsuccessful attempt in the database; prompts the user to try again
Step 2. Type in the correct keycode and door identifier	System flashes a green light to indicate success; records successful access in the database; disarms the lock device

Test Coverage

- **Test coverage** measures the degree to which the specification or code of a software program has been exercised by tests
  - "Big picture": Specification testing focuses on the coverage of the input space, without necessarily testing each part of the software ← Acceptance tests
  - "Implementation details": Code coverage measures the degree to which the elements of the program source code have been tested ← Unit tests

Heuristic: Some Tests are More "Critical" than Others

- Test cases should be prioritized—some tests are more likely to uncover faults
- Tests are about finding developer errors, and people are prone to make certain kind of errors
- Some tests can easier pinpoint problems than others
- (some) **Heuristics for achieving high coverage**:
  (could be applied individually or in combination)
  - equivalence testing
  - boundary testing

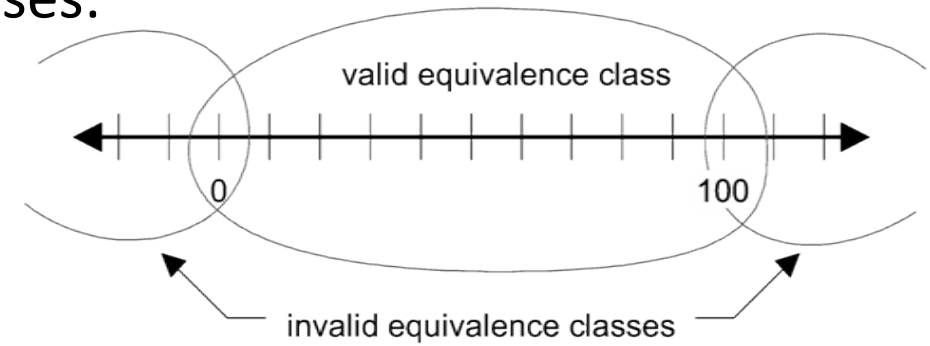
  } mostly for "black-box" testing
  - control-flow testing
  - state-based testing

  } mostly for "white-box" testing

Input Space Coverage:
Equivalence Testing

- **Equivalence testing** is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program is expected to "behave the same" on each input from the same group
  - Assumption: A well-intentioned developer may have made mistakes that affect a whole class of input values
  - Assumption: We do not have any reason to believe that the developer intentionally programmed special behavior for any input combinations that belong to a single class of input values
- Two steps:
  1. partitioning the values of input parameters into equivalence groups
  2. choosing the test input values from each group

Equivalence classes:



Heuristics for Finding Equivalence Classes

- For an input parameter specified over a **range of values**, e.g., a number or a letter, partition the value space into one valid and two invalid equivalence classes
- For an input parameter specified with a **single value**, e.g., a unique "security code," partition the value space into one valid and two invalid equivalence classes
- For an input parameter specified with a **set of values**, e.g., day or month names, partition the value space into one valid and one invalid equivalence class
- For an input parameter specified as a **Boolean value**, partition the value space into one valid and one invalid equivalence class

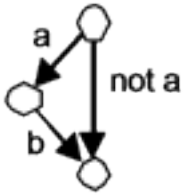
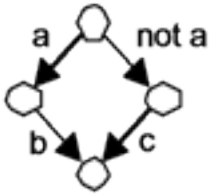
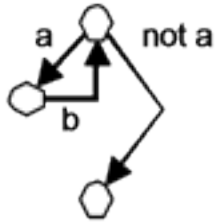
Input Space Coverage:
Boundary Testing

- **Boundary testing** is a special case of equivalence testing that focuses on the boundary values of input parameters
  - Based on the assumption that developers often overlook special cases at the boundary of equivalence classes (e.g., Microsoft Zune bug: https://en.m.wikipedia.org/wiki/Leap_year_bug)
- Selects elements from the "edges" of each equivalence class, or "outliers" such as
  - zero, min/max values, empty set, empty string, and null
  - confusion between > and >=
  - palindromes (impossible to build an FSM to recognize a palindrome code)
  - etc.

Code Coverage:
Control-flow Testing

- Statement coverage
  - Each statement executed at least once by some test case
- Edge coverage
  - Every edge (branch) of the control flow is traversed at least once by some test case
- Condition coverage
  - Every condition takes TRUE and FALSE outcomes at least once in some test case
- Path coverage
  - Finds the number of distinct paths through the program to be traversed at least once

Constructing the **control graph** of a program for Edge Coverage:

a;	a; b;	*if* a *then* b;	*if* a *then* b *else* c;	*while* a *do* b;
				

* Exceptions are also a form of control flow, as is concurrency or multithreading

Code Coverage:
State-based Testing

- **State-based testing** defines a set of *abstract* states that a software unit (object) can take and tests the unit's behavior by comparing its actual states to the expected states
  - This approach is popular with object-oriented systems
  - Like equivalence classes, state diagrams usually are mind constructs and may be incomplete (not showing all possible states and transitions) or incorrect
- The **state** of an object is defined as a constraint on the values of its attributes
  - Because the methods use the attributes in computing the object's behavior, the behavior depends on the object state
  - An object without attributes does not have states, but still can be unit-tested (shown later)

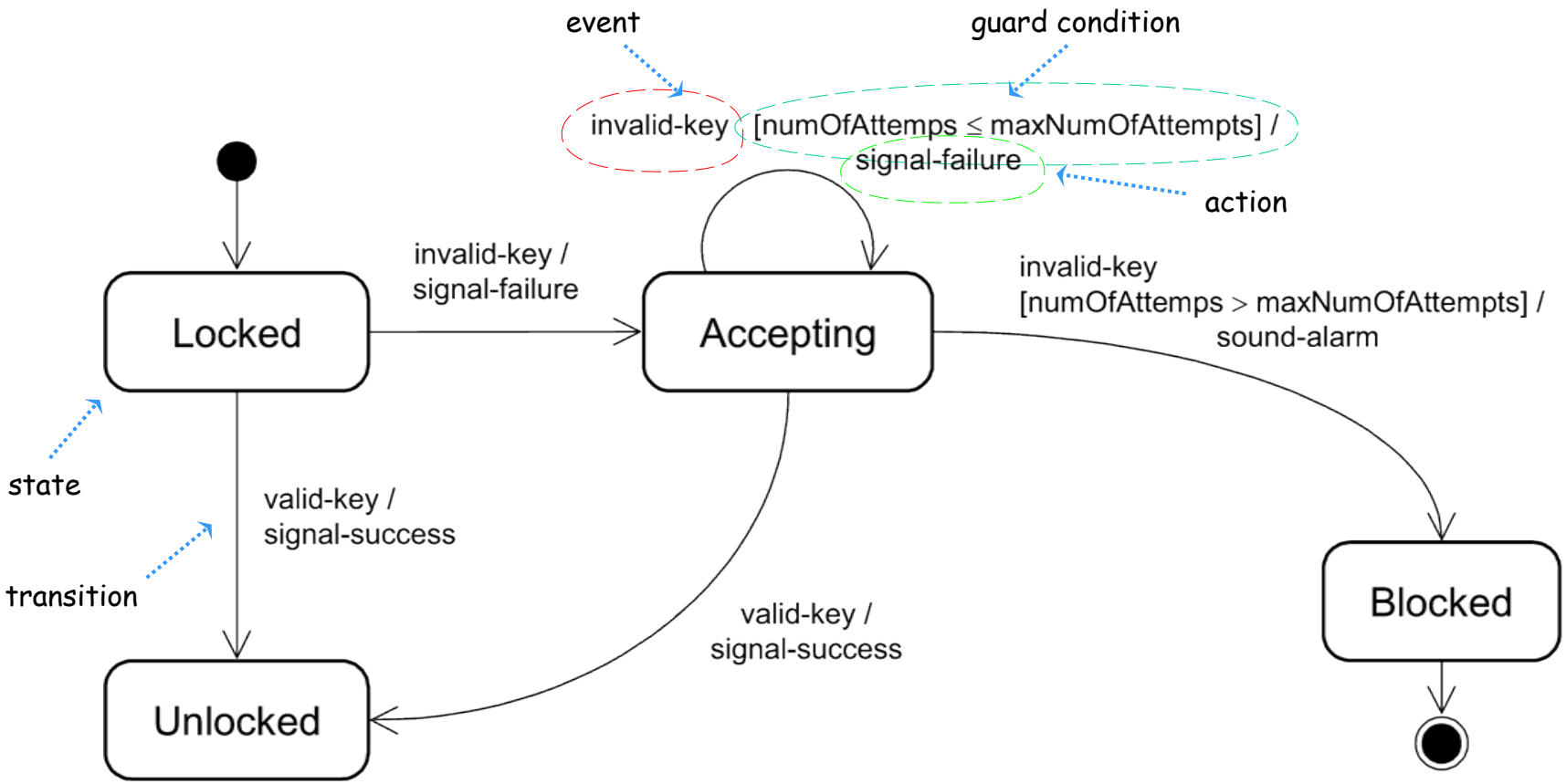
State-based Testing Example

(Safe Home Access System)

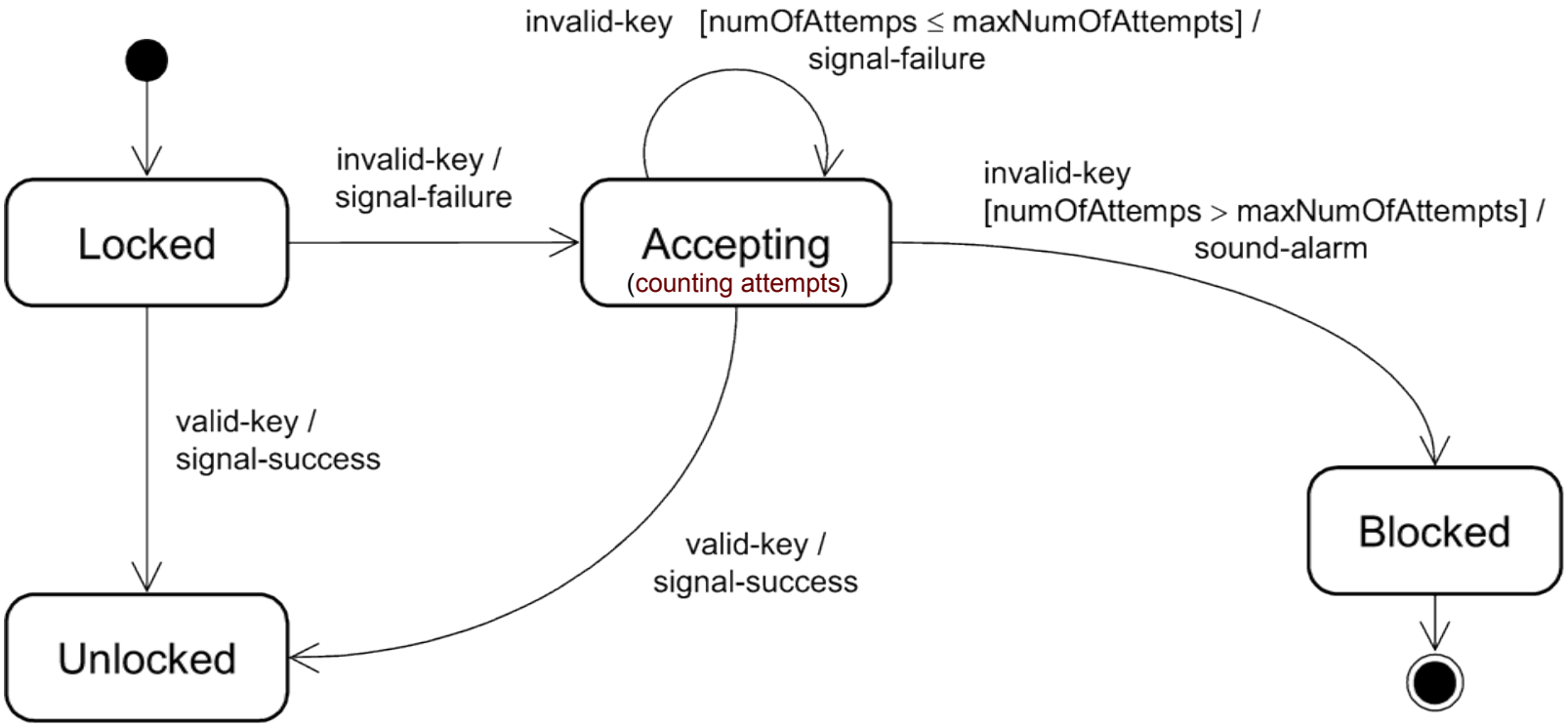
- Define the relevant states as combinations of object attribute values:
  1. "**Locked**" ≡ defined as:
     (lockDeviceArmed == true) && (numOfAttempts == 0)
  2. "**Accepting**" ≡ defined as(*):
     (lockDeviceArmed == true) && (0 < numOfAttempts < maxNumOfAttempts)
  3. "**Unlocked**" ≡ defined as:
     (lockDeviceArmed == false) && (numOfAttempts == 0)
  4. "**Blocked**" ≡ defined as:
     (lockDeviceArmed == true) && (numOfAttempts == maxNumOfAttempts)
  5. "**Undefined**" ≡ defined as:
     ((numOfAttempts < 0) || (maxNumOfAttempts < numOfAttempts)) ||
     ((lockDeviceArmed == false) && (numOfAttempts != 0))
- Define the relevant events:
  1. User entered a valid key
  2. User entered an invalid key

(*) One may argue that "Locked" is a sub-state of "Accepting" …

State-based Testing Example



State-based Testing Example



Controller State Diagram Elements

- Four states
  { Locked, Unlocked, Accepting, Blocked }
- Two events
  { valid-key, invalid-key }
- Five valid transitions
  { Locked→Unlocked, Locked→Accepting,
  Accepting→Accepting, Accepting→Unlocked,
  Accepting→Blocked }

Ensure State Coverage Conditions

- Cover all identified states at least once

  (each state is part of at least one test case)

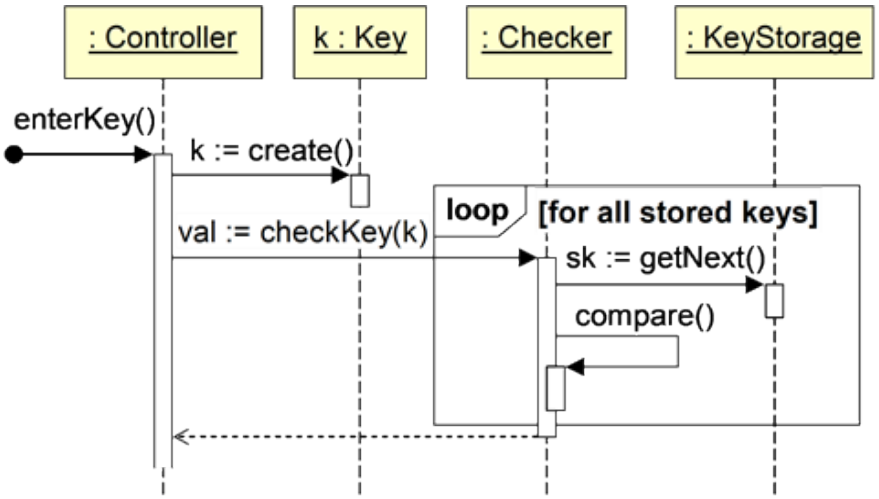
- Cover all valid transitions at least once

- Trigger all invalid transitions at least once

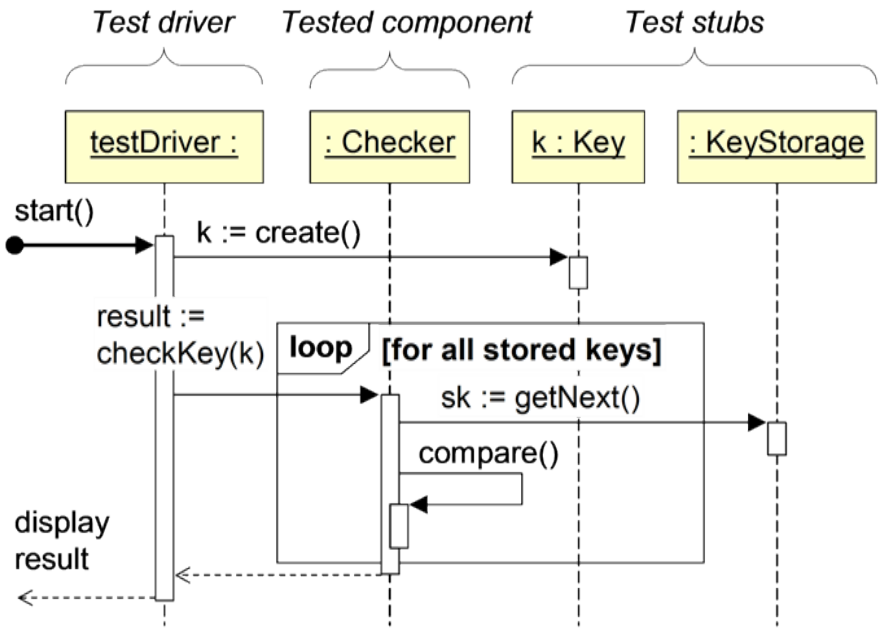
Practical Aspects of Unit Testing

- Mock objects:
  - A test **driver** simulates the part of the system that invokes operations on the tested component
  - A test **stub** simulates the components that are called by the tested component
- Mock objects needed because:
  - The corresponding actual objects are not available
  - Easier to locate faults using mock objects that are trivial and therefore not a source of faults
- The unit to be tested is also known as the **fixture**
- Unit testing follows this cycle:
  1. Create the thing to be tested (fixture), the driver, and the stub(s)
  2. Have the test driver invoke an operation on the fixture
  3. Evaluate that the actual state equals expected state

Testing the KeyChecker (Unlock Use Case)



(a)



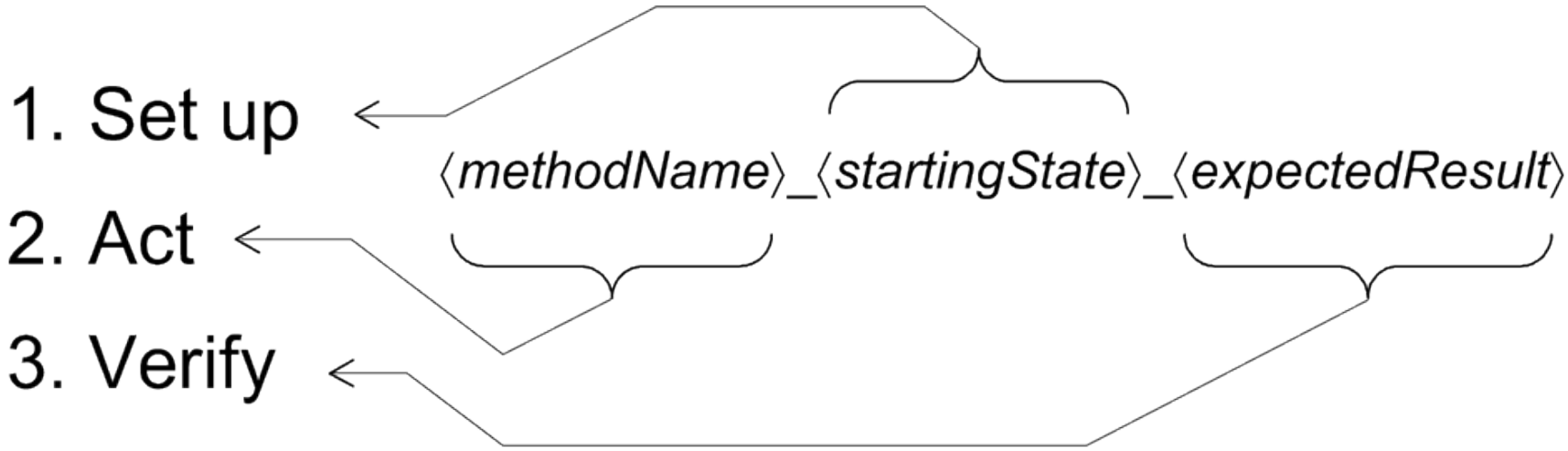
(b)

Example Test Case

Listing 2-1: Example test case for the Key Checker class.

```
public class CheckerTest {
  // test case to check that invalid key is rejected
  @Test public void
    checkKey_anyState_invalidKeyRejected() {

    // 1. set up          // no states defined for Key Checker
    Checker fixture = new Checker( /* constructor params */ );

    // 2. act      // test driver (this object) invokes tested object (Key Checker)
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    boolean result = fixture.checkKey(invalidTestKey);

    // 3. verify          // check that invalid key is rejected
    assertEqual(result, false);
  }
}
```

Test Case Method Naming



Example test case method name:

```
checkKey_anyState_invalidKeyRejected()
```

xUnit / JUnit

- Verification of the expected result is usually done using the assert_*_() methods that define the expected state and report errors if the actual state differs
- http://www.junit.org/
- Examples:
  - assertTrue(4 == (2 * 2));
  - assertEquals(expected, actual);
  - assertNull(Object object);
  - etc.

Another Test Case Example

Listing 2-2: Example test case for the Controller class.

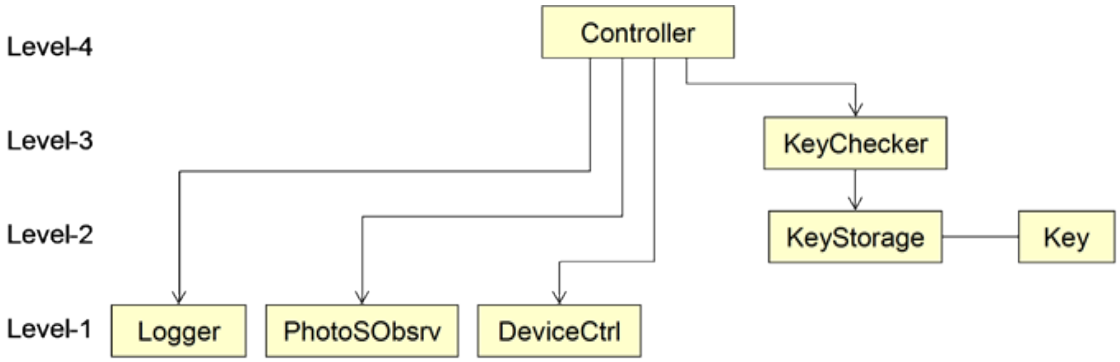
```
public class ControllerTest {
  // test case to check that the state Blocked is visited
  @Test public void
    enterKey_accepting_toBlocked() {

    // 1. set up: bring the fixture to the starting state
    Controller fixture = new Controller( /* constructor params */ );
    // bring Controller to the Accepting state, just before it blocks
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    for (i=0; i < fixture.getMaxNumOfAttempts(); i++) {
        fixture.enterKey(invalidTestKey);
    }
    assertEqual(   // check that the starting state is set up
        fixture.getNumOfAttempts(), fixture.getMaxNumOfAttempts() - 1
    );

    // 2. act
    fixture.enterKey(invalidTestKey);

    // 3. verify
    assertEqual(   // the resulting state must be "Blocked"
        fixture.getNumOfAttempts(), fixture.getMaxNumOfAttempts()
    );
    assertEqual(fixture.isBlocked(), true);
  }
}
```

Integration Testing

- Key Issue:
  - How to assemble individually-tested “units” and quickly locate faults that surface during integration
- Horizontal Integration Testing
  - “Big bang” integration
    - Problem: hard to isolate individual unit(s) that caused an integration fault
  - Bottom-up integration
  - Top-down integration
  - Sandwich integration
- Vertical Integration Testing
  - Vertically slice the system by use cases and first integrate within each slice (using any of the horizontal integration techniques)

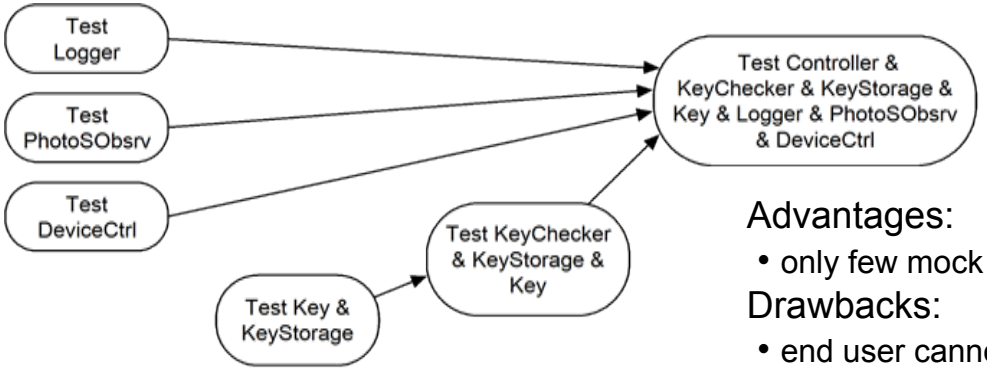
Horizontal Integration Testing

System hierarchy:



Bottom-up integration testing:

Implementation and testing proceeds from Level-1 (bottom) to Level-4 (top)



Advantages:

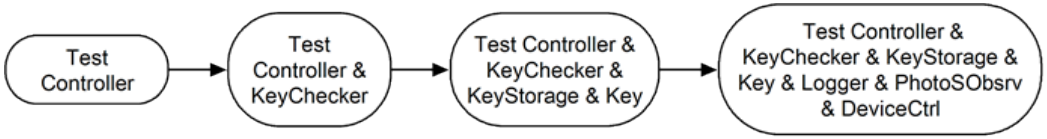
- only few mock objects need to be implemented

Drawbacks:

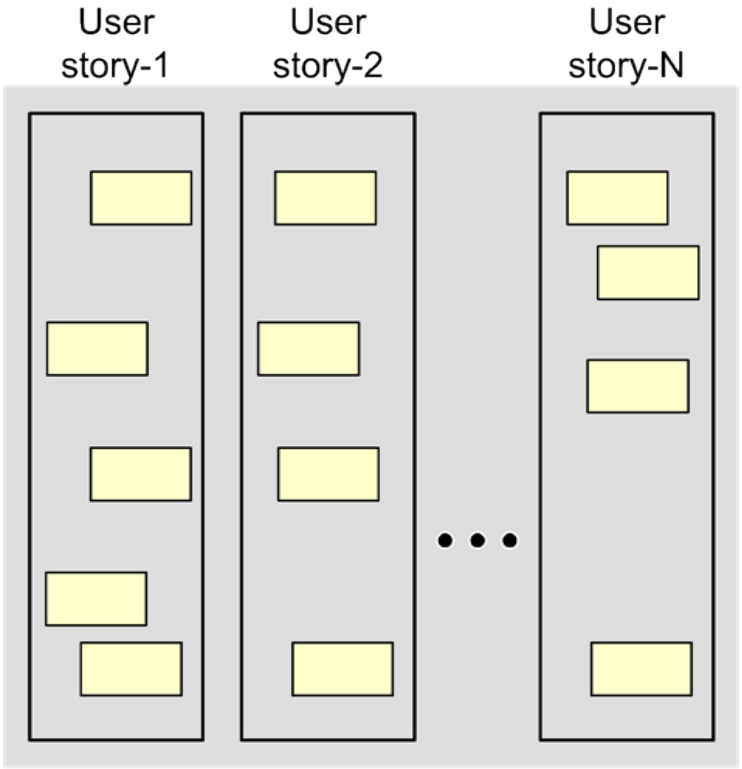
- end user cannot see any functionality until late in the development process

Top-down integration testing:

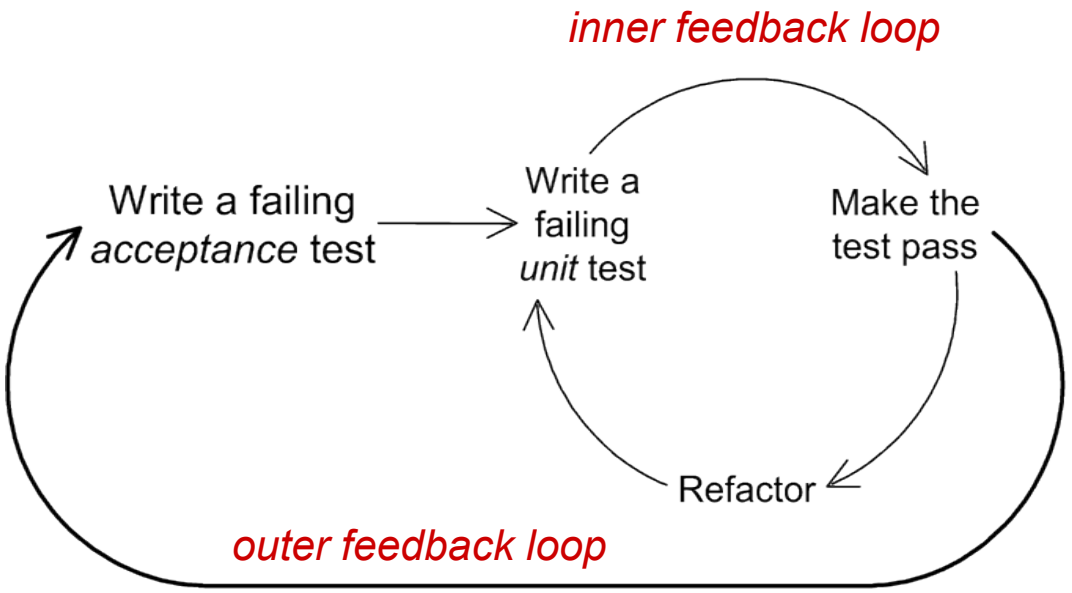
Implementation and testing proceeds from Level-4 (top) to Level-1 (bottom)



Vertical Integration Testing



Whole system

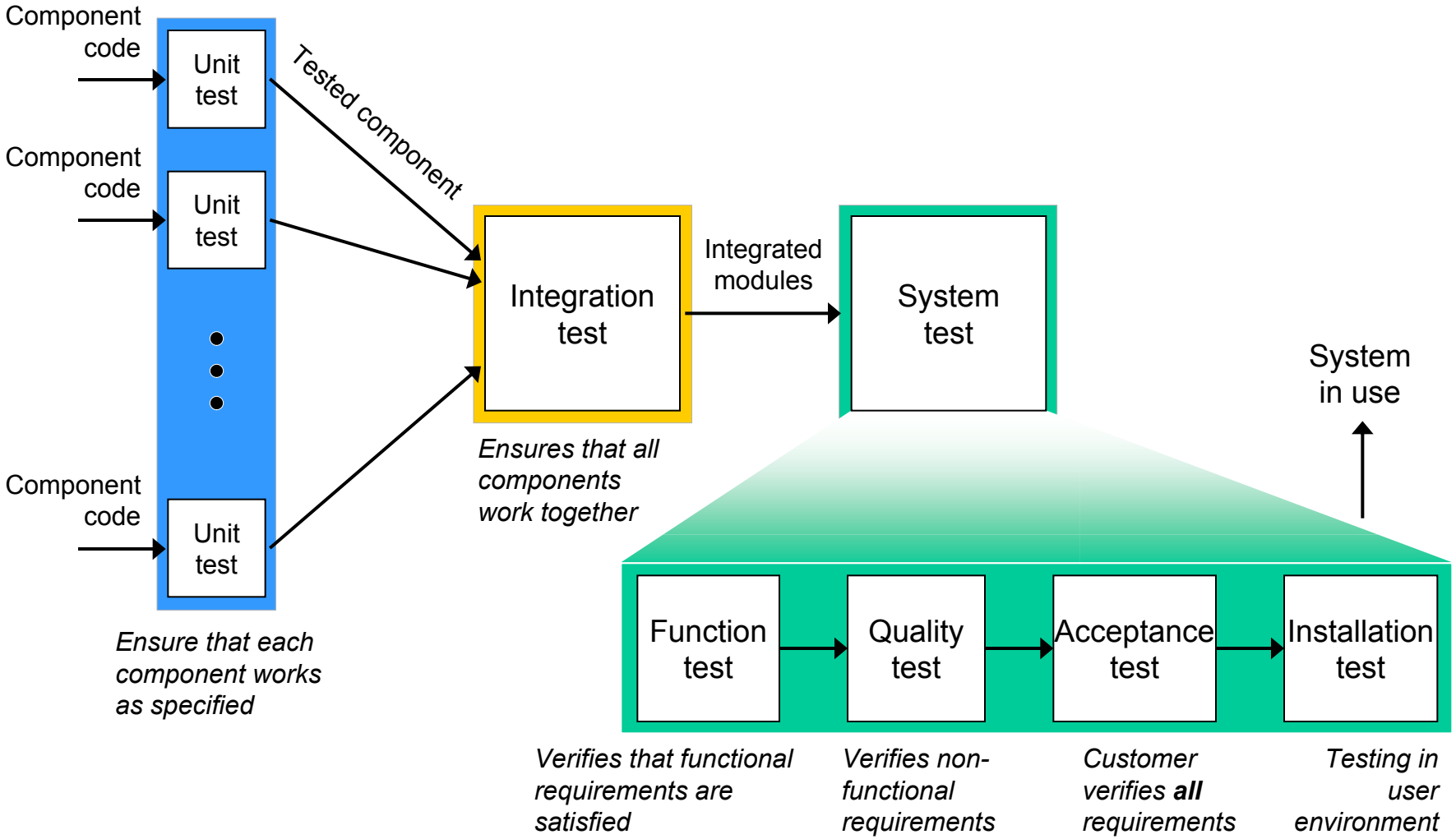


Developing user stories:

Each story is developed in a cycle that integrates **unit tests** in the **inner feedback loop** and the **acceptance test** in the **outer feedback loop**

Logical Organization of Testing

(Not necessarily how it's actually done!)

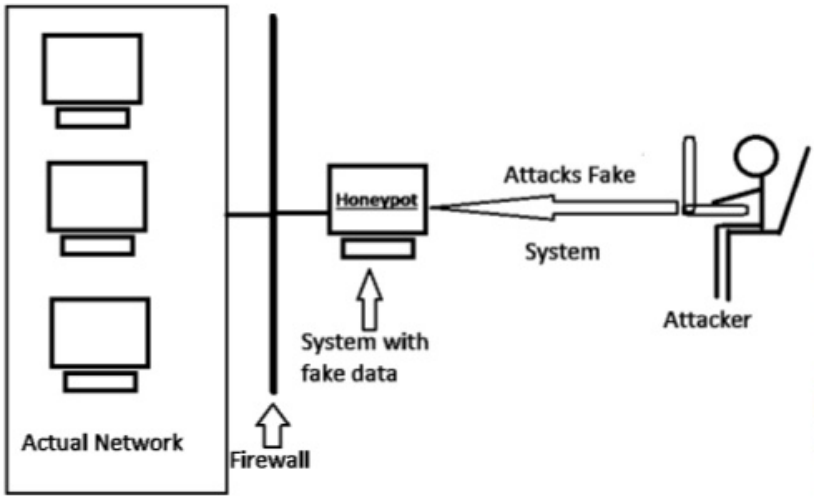


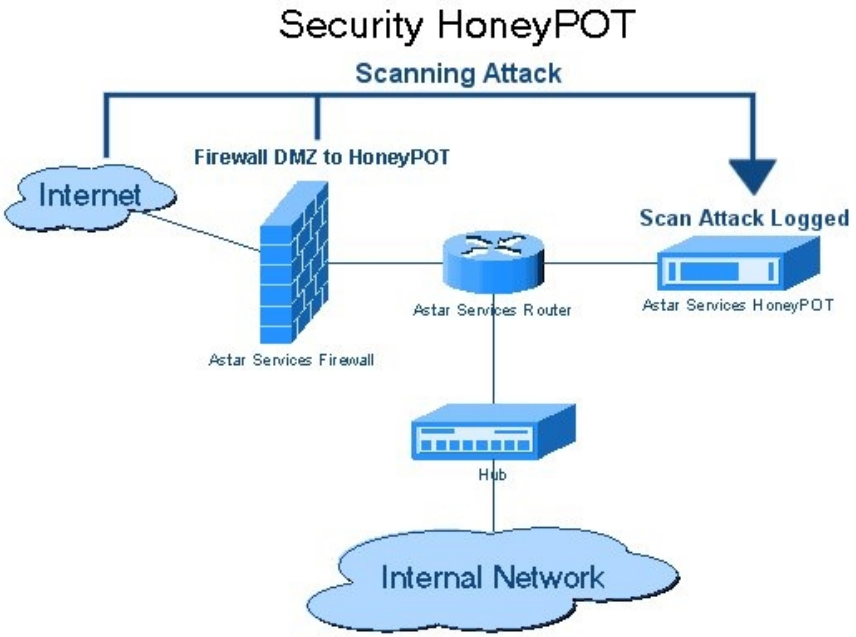
System Testing

- To test the whole system in our case study of safe home access,
  we need to work with a locksmith and electrician to check that the lock and electrical wiring are properly working,
  or with a third party software developer (if 3rd party software is used)

Security Honeypots

- A deception used to detect and study network attacks
  - Part of security testing of software systems



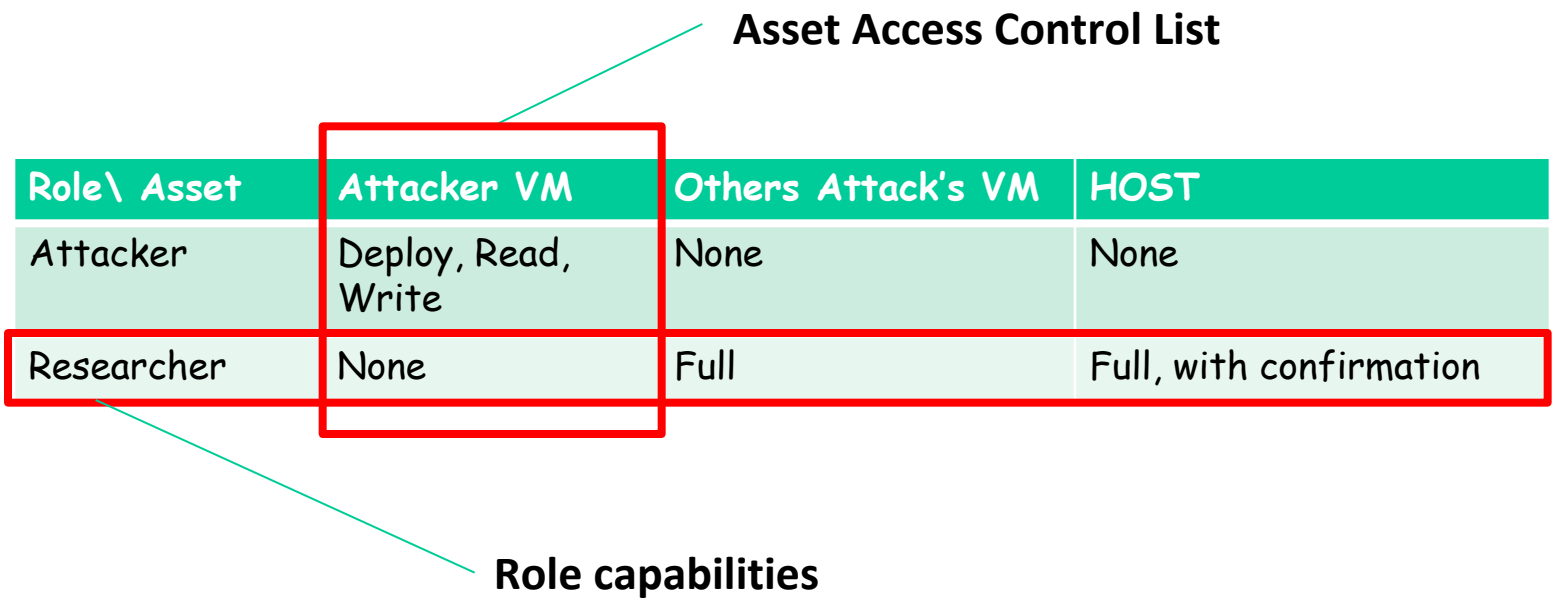


Honeypot Access Control

Who are the users?

- Attackers:
  - Access the VM set up for them
- Researchers
  - Access the host, the attackers' VM
  - Should not erase files by mistake…

Honeypot Access Control Matrix



Role\ Asset	Attacker VM	Others Attack's VM	HOST
Attacker	Deploy, Read, Write	None	None
Researcher	None	Full	Full, with confirmation

Honeypot Access Control

Who are the users?

- Attackers
  - Script kiddies: 95%
    - Untrusted
    - Run malware found on the internet
    - Easy to protect against
  - Advanced attackers: 5%
    - Untrusted
    - Are able to understand and evade the system
    - Harder to protect against
- Researchers ← **Refine the access control for the new distinction!**
  - Trusted
  - Can erase file by mistake…