Project 3 Grades

- Grades are available on Blackboard
- Self evaluation is 5% and team evaluation 10%
- **Overall: Great job!** (In general, we noticed improvements across the board.)
- Evaluations: use professional language (these are not blog posts)
- **Communication is the key (be open to your teammates' ideas)**
- Make sure teammates are happy with the tasks they are covering
- Everybody should code
- Reminder: *Appeals: Should you wish to appeal a grade that you have received on a laboratory assignment, exam, or anything else, you must do so within one week of receiving the graded item.* (Syllabus)

Software Testing

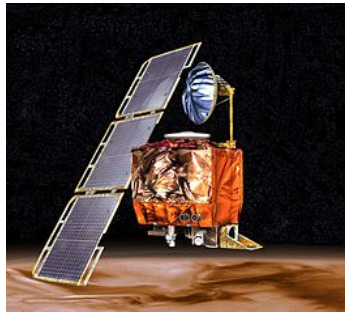
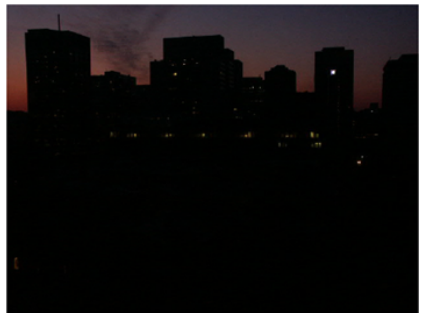
Prof. Alex Bardas

Why testing?

- **Errors** can happen in any engineering discipline
  - Software is one of the most error-prone products of all engineering areas
    - Vague requirements
    - Complex in nature, undecidable problems are everywhere
    - On average, 1-5 bugs per KLOC (thousand lines of code)
  - Almost all software applications in the market have a number of bugs
- Testing is **costly**
  - The most time consuming and expensive part in software development
  - Extensive hardware-software integration requires more testing

Why testing?

- **However, not testing is even more expensive!**

- THERAC-25 Radiation Therapy (2 death, 1985)
- Shooting down of Airbus A300 (290 death, 1988)
- Mars Climate Orbiter ($165M, 1998)
- Northeast Blackout ($6 billions, 2003)
- Inadequate software testing costs in the US between $22 and $59 billion in 2002

[2002 NIST report “The Economic Impacts of Inadequate Infrastructure for Software Testing”]

Approach to reduce bugs

- **Inspection**: manually review the code to detect faults
  - Hard to evaluate
- **Static checking**: identify specific problems in software by scanning the code or all possible paths
  - Limited problem types; false positive
- **Formal Proof**: formally prove that the program implements the specification
  - Difficult to have a formal specification; effort costly
- **Testing**: feed input to software and run it to see whether its behavior is as expected
  - Need test oracles; limited coverage; no 100% error free guarantee

Approach to reduce bugs

- The winner is testing
  - More reliable and cheaper than inspection
  - Inspection was the major answer in the old days (or when testing is expensive)
  - Linear rewards: “you get what you pay for”

“50% of my employees are testers, and the rest spends 50% of their time testing”

---- Bill Gates, 1995

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

- **Testing** (IEEE definition)
  - An execution of software in a controlled environment (input) and “validating” the output

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

- **Test case**
  - An execution of the software with a given list of input values
  - Include:
    - Input values, sometimes fed in different steps
    - Expected outputs

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

- **Test oracle**
  - The expected outputs of software by feeding in a list of input values
  - **A mechanism for determining whether a test has passed or failed**
  - Part of test cases
  - Hardest problem in auto-testing: test oracle problem

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

- **Test Suite**
  - A collection of test cases
  - Usually these test cases share similar pre-requisites and configurations
  - Usually can be ran together in sequence
  - Different test suites for different purposes
    - smoke test, certain platforms, certain feature, performance, etc.

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

- **Test Script**
  - A script to run a sequence of test cases or a test suite automatically
- **Test Driver**
  - A software framework that can load a collection of test cases or a test suite
  - It can usually handle the configuration and comparison between expected outputs and actual outputs

Testing: Concepts

Test case

Test oracle

Test suite

Test script

Test driver

Test result

Test coverage

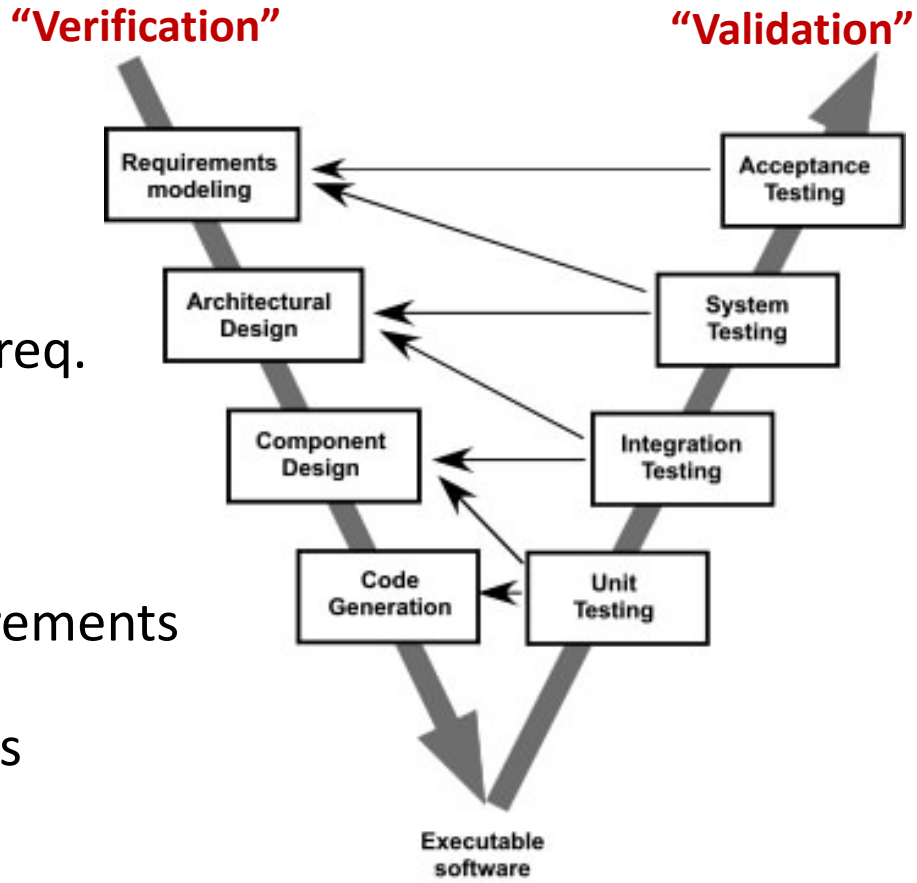
- **Test Coverage**
  - A measurement to evaluate how well the testing is done
  - Can be based on multiple elements
    - code
    - input combinations
    - specifications

Testing: Granularity

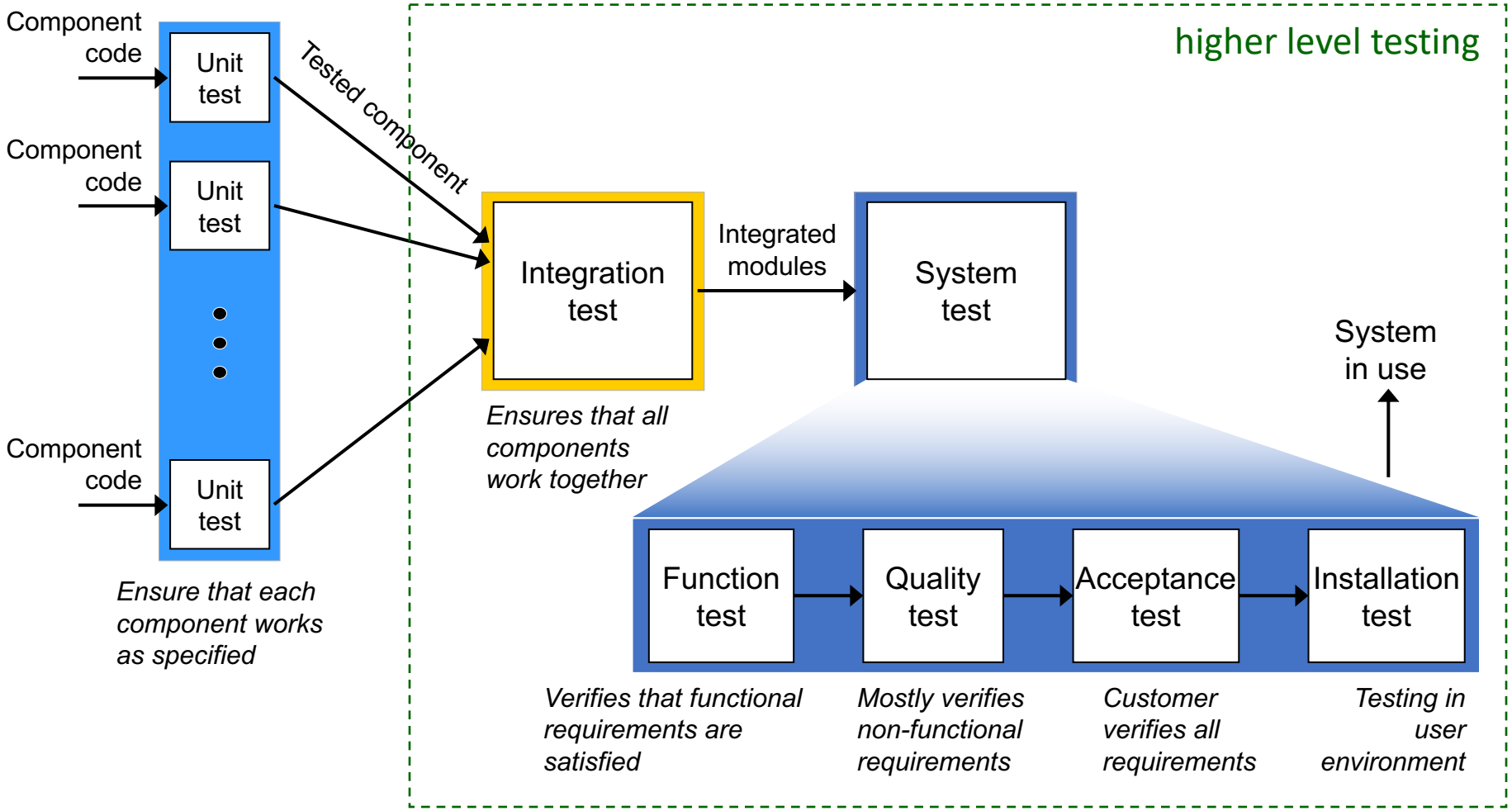
- **Unit Testing**
  - Test of a single unit/module
- **Integration Testing**
  - Test the interaction between modules
- **System Testing**
  - Test the system as a whole; comply to requirements
  - Environments, external exceptions
  - By developers on test cases
- **Acceptance Testing**
  - Validate the system against user requirements
  - Usually on GUI
  - By customers with no formal test cases

Testing: Granularity

- **Unit Testing**
  - Test of a single unit/module
- **Integration Testing**
  - Test the interaction between modules
- **System Testing**
  - Test the system as a whole; comply to req.
  - Environments, external exceptions
  - By developers on test cases
- **Acceptance Testing**
  - Validate the system against user requirements
  - Usually on GUI
  - By customers with no formal test cases



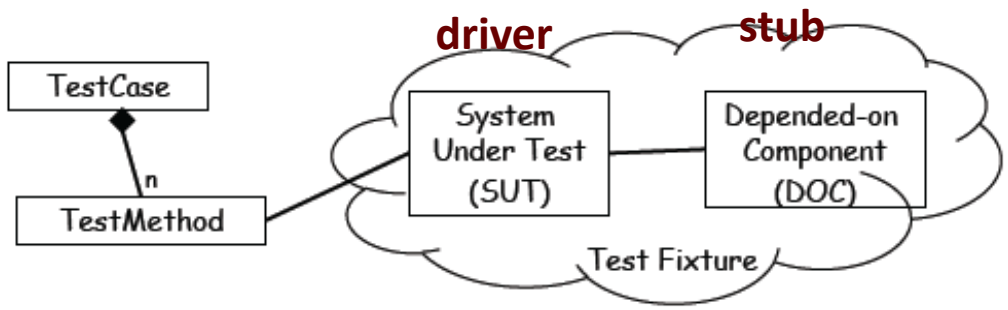
Testing: Logical Organization



Unit Testing

- Testing a basic module of the software
  - A function, a class, a component
  - The goal is to find differences between the design model and its implementation
- Typical problems revealed
  - Interface
  - Local data structures
  - Algorithms
  - Boundary conditions
  - Error handling

To isolate the module to be tested from the rest of the system



Unit Test Framework

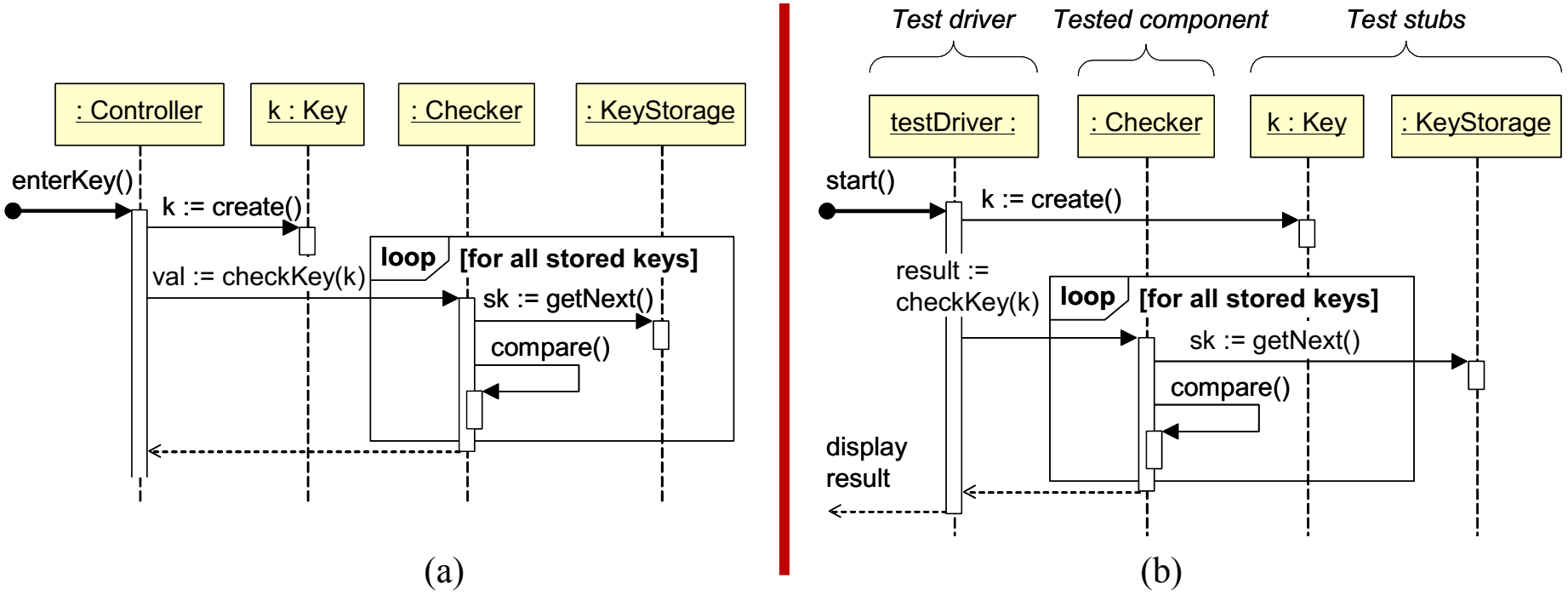
Unit Testing Framework

- xUnit/JUnit
  - Created by Kent Beck in 1989, 70 xUnit frameworks for corresponding languages
  - First one was sUnit (for smalltalk)

  - JUnit is the most popular xUnit framework: http://www.junit.org
  - Usually use the assert*() methods that define expected state
    - assertTrue(4 == (2 * 2));
    - assertEquals(expected, actual);
    - assertNull(Object object);

Unit Testing Example

- Safe Home Access Project: “unlock” use case
  - Let’s test the “Key Checker” module



Unit Testing Example

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 objects
    Checker checker = new Checker( /* constructor params */ );

    // 2. act on the tested object
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    boolean result = checker.checkKey(invalidTestKey);

    // 3. verify the outcome is as expected
    assertEqual(result, false);
  }
}
```

Example test case method:

checkKey_anyState_invalidKeyRejected()

Unit Testing

- Assertions
  - “assertEqual(result, false)”
    - We expect the output as “false” for an invalid key
  - Consider the following test method:

```
public void testCapacity() {
   int size= fFull.size();
   for (int i= 0; i < 100; i++){
      fFull.addElement(new Integer(i));
   }
   assertTrue(fFull.size() == 100+size);
```

```
assertTrue(fFull.size() == 100+size);
```

Assertion failed: myTest.java:150
(expected true but was false)

```
assertEquals(100+size, fFull.size());
```

Assertion failed: myTest.java:150
(expected 102 but was 103)

Integration Testing

- Testing the interaction among a number of interactive components
- Strategies
  - Big Bang
  - Bottom Up
  - Top down

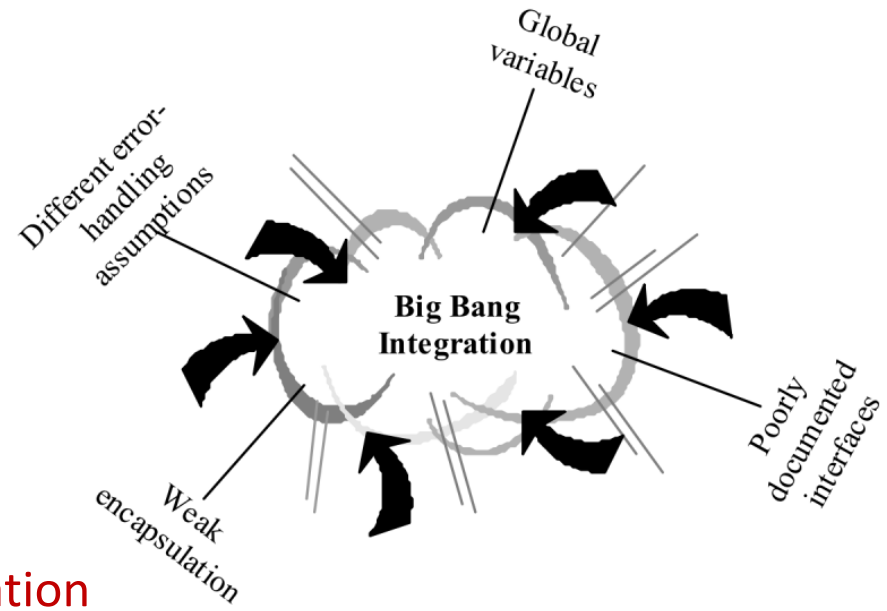
Big Bang

Prepare all relevant components

Data, global variables, etc.

Put them together

Pray!



Common in small projects

Requires no extra cost for integration

Difficult to address errors → may work well if interfaces are well-defined

Integration Testing

- Testing the interaction among a number of interactive components
- Strategies
  - Big Bang
  - Bottom Up
  - Top down

Bottom-Up

A hierarchical structure of all software features

The lowest units first - the unit that depends on nothing else

Do not need test stubs

Requires test drivers, which can be re-used

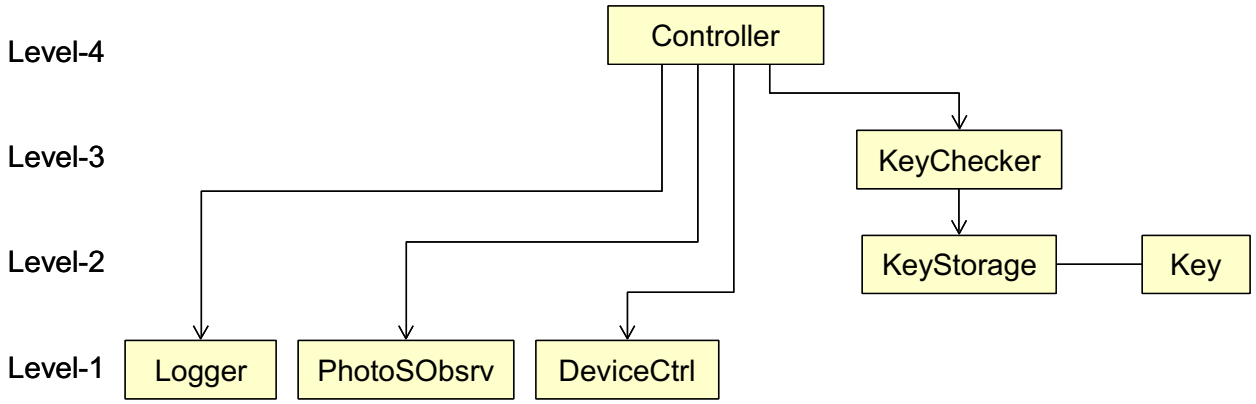
Support parallel integration

No working system until the end!

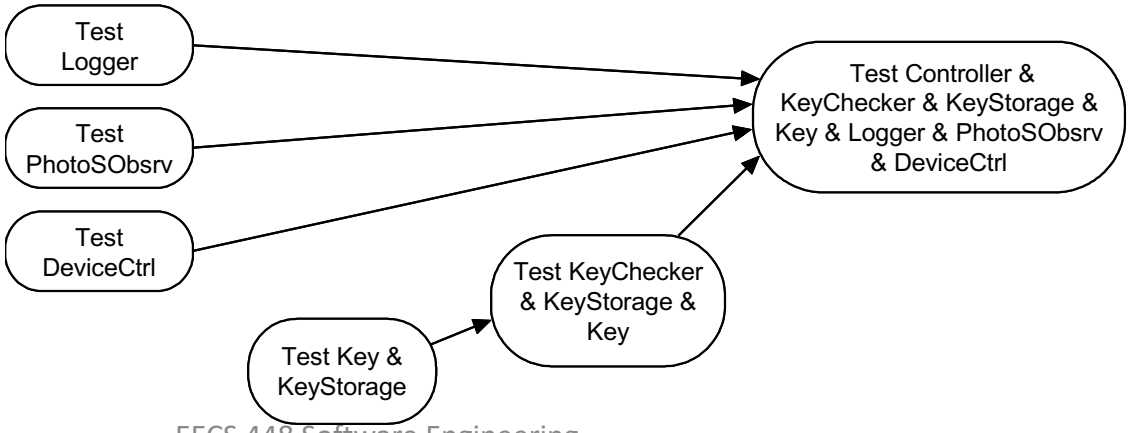
Need more interactions among teams working on each component

Integration Testing Example

System hierarchy:



Bottom-up integration testing:



Integration Testing

- Testing the interaction among a number of interactive components
- Strategies
  - Big Bang
  - Bottom Up
  - Top down

Top Down

Starts by testing the units at the highest level of hierarchy

Gradually add components

Depth-first or breadth-first integration

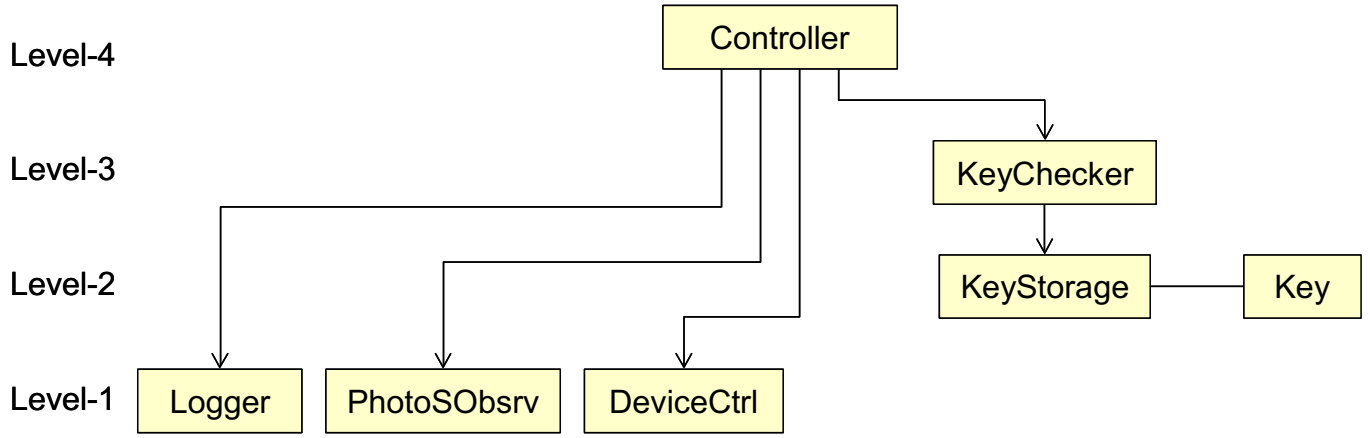
Easy to understand; lead to a working system earlier

No need to build test drivers (only test stubs)

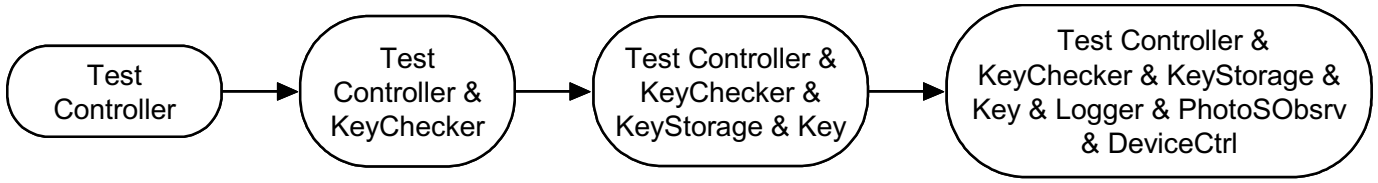
Centralized (cannot be parallelized)

Integration Testing Example

System hierarchy:



Top-down integration testing:



Integration Testing

- **Sandwich integration**
  - Combine top-down and bottom-up
  - The middle level is the *target* level
  - Incrementally use components of the target level in both directions
  - Can test earlier; can be parallelized
- **Regression Testing**
  - Test a new version with old test cases
  - Used when a new module is added

System Testing

- Test the system as a whole
  - Test if the system complies with the functional and non-function requirements
- Usually test against specifications
  - For each item in the specification
    - Work out a test case and a test oracle
    - Test boundary values
    - Test with invalid inputs
    - Test with environment errors

System Testing

- Test the system as a whole
  - Test if the system complies with the functional and non-function requirements
- Consider environment issues
  - Building
    - Compile options and configurations
  - Underlying platforms
    - OS, database, application server, browser
  - Compatibility
    - Different platforms, configurations

Acceptance Testing

- Testing by users of the system as a whole
  - If specifications comply with user requirements

e.g., REQ3: "given a valid key code, the system should unlock the door"

Acceptance Testing

- Testing by users on the system as a whole
  - If specifications comply with user requirements

e.g., REQ3: "given a valid key code, the system should unlock the door"

We can derive test cases to

- Test with a valid key of a current tenant on his apartment (pass)
- Test with a valid key of a current tenant on another apartment (fail)
- Test with an invalid key on any apartment (fail)
- Test with the key of removed tenant on his previous apartment (fail)
- Test with a valid key of a newly added tenant on his apartment (pass)

User Acceptance Testing

- Write as **user acceptance test case –** a detailed procedure that fully tests a use case or one of its flows of events

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

Acceptance Testing

- GUI Testing
  - Hard to automate and hard to compare results using oracles
- **Manual testing** is still widely performed for GUI testing
  - Manually explore the user interface
    - “Record and replay”: keyboard inputs, clicks, UI events
  - Record the steps in the test for future testing
  - Observe the GUI for errors

References

- Prof. Fengjun Li's EECS 448 Fall 2015 slides

- This slide set has been extracted and updated from the slides designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill 2014) by Roger Pressman