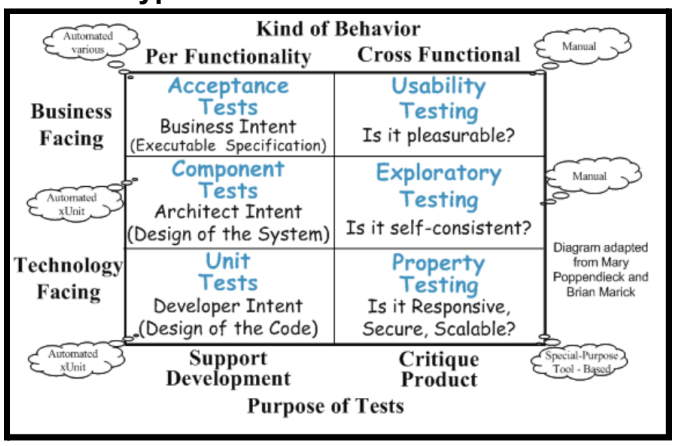
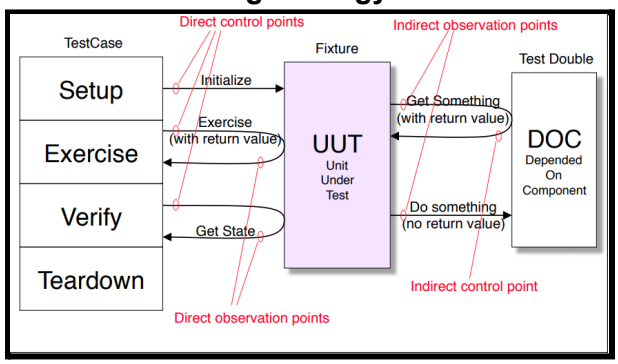
Automated Testing:

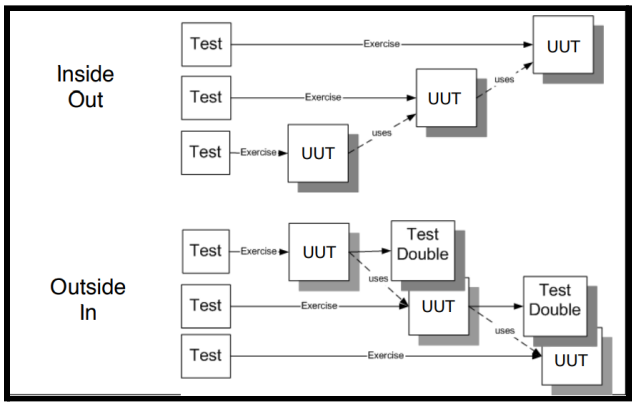
- **Different types of tests:**



- Where possible, automate your testing. By doing regression testing, tests can be repeated whenever the code is modified. This takes the tedium out of extensive testing and makes more extensive testing possible.
- In order to do automated testing, you'll need:
  - **Test drivers** which will automate the process of running a test set. It sets up the environment, makes a series of calls to the Unit-Under-Test (UUT), saves the results and checks if they were right and generates a summary for the developer.
  - **Test stubs** which will simulate part of the program called by the UUT. It checks whether the UUT set up the environment correctly, checks whether the UUT passed sensible input parameters to the stub and passes back some return values to the UUT.
- **Automated Testing Strategy:**

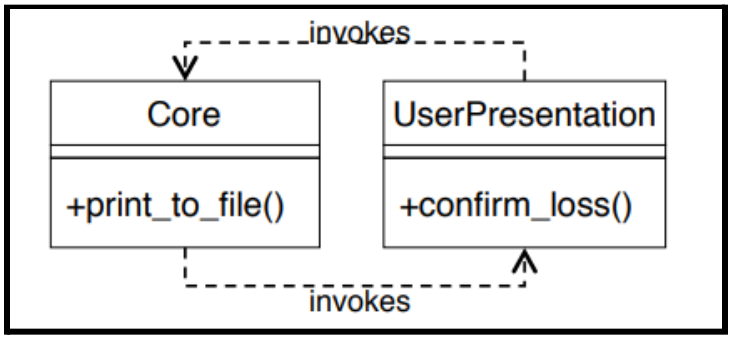


- Test Order

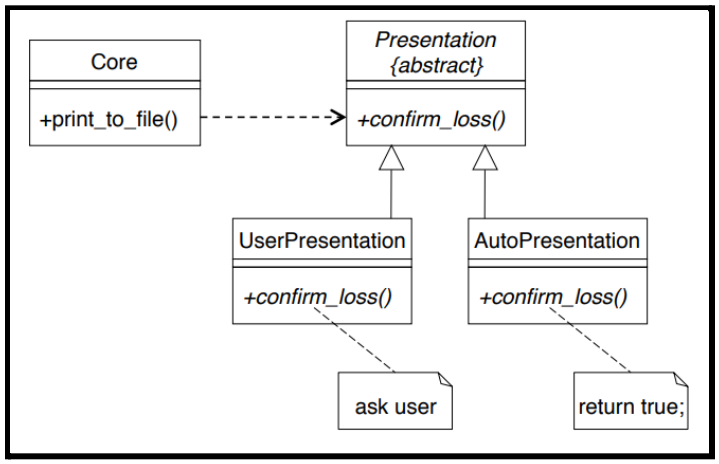


- **JUnit:**
- JUnit is a unit testing framework for Java.
- Assertion methods in JUnit:
    - **Single-Outcome Assertions:**
      E.g. fail;
    - **Stated Outcome Assertions:**
      E.g. assertNotNull(anObjectReference);
      E.g. assertTrue(booleanExpression);
    - **Expected Exception Assertions:**
      E.g. assert_raises(expectedError) {codeToExecute };
    - **Equality Assertions:**
      E.g. assertEqual(expected, actual);
    - **Fuzzy Equality Assertions:**
      E.g. assertEqual(expected, actual, tolerance);
- **Principles of Automated Testing:**
    - Write the test cases first.
    - Design for testability.
    - Use the front door first. This means test using public interfaces and avoid creating backdoor manipulations.
    - Communicate intent. Treat tests as documentation and make it clear what each test does.
    - Don't modify the UUT. Avoid test doubles and test-specific subclasses unless absolutely necessary.
    - Keep tests independent.
    - Isolate the UUT.
    - Minimize rest overlap.
    - Check one condition per test.
    - Test different concerns separately.
    - Minimize untestable code.
    - Keep test logic out of production code.
- **Challenges for automated testing:**
    - **Synchronization** - How do we know a window popped open that we can click in?
    - **Abstraction** - How do we know it's the right window?
    - **Portability** - What happens on a display with different resolution/size?
- **Techniques for testing the presentation layer:**
    - **Script the mouse and keyboard events:**
        - We can write a script that sends mouse and keyboard events. (E.g. "send_xevents @400,100")
        - However, this is not good practice/design because the script is write-only and fragile.
    - **Script at the application function level:**
        - E.g. Applescript: tell application "UMLet" to activate.
        - This is robust against size and position changes but fragile against widget renamings, and layout changes. Hence, this is still not good practice/design.
    - **Write an API for your application:**
        - We can use these APIs for testing.
        - E.g. Allow an automated test to create a window and interact with widgets.

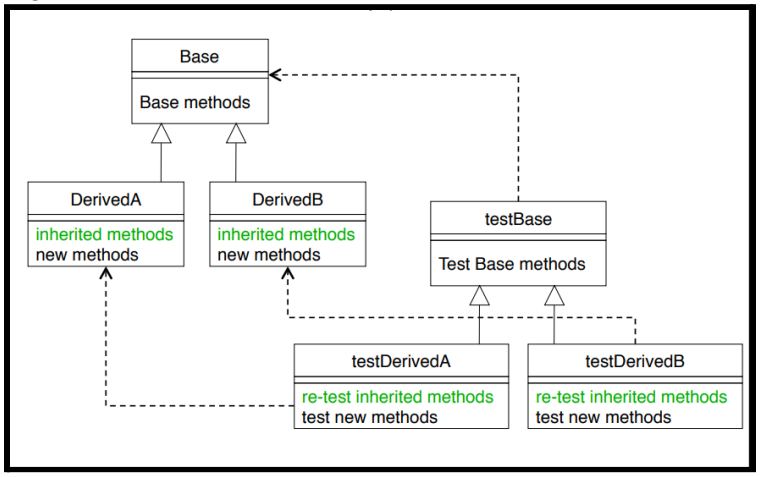
- **Circular Dependencies:**
- If you have circular dependencies in your code, you should refactor your code to remove them.
- E.g.
  Here, we have a circular dependency.



Because we have a circular dependency, we need to refactor the code. Here is the new code.



- **Testing Object Oriented Code:**
- Object oriented code can be hard to test. The best/most efficient way to test object oriented code is to have a parent test class for the parent class and to extend the parent test class for the subclasses.
  E.g.



E.g.
The test class for the parent class.

```
class FooTest {
        @Test
        public void testSomeMethodBar() {
         ...
        }

        @Test public void void someOtherMethodBaz(Baz baz) {
        ...
        }
}
```

The test class for the subclass class. Notice that this test class inherits from FooTest.

```
class EnhancedFooTest extends FooTest {
        @Test
        public void testSomeMethodBar() {
         ...
        }
}
```

- **When to stop testing:**
  - **Motorola's Zero-failure** testing model predicts how much more testing is needed to establish a given reliability goal.
  - **Failures = $ae^{-b(t)}$** where "a" and "b" are constants and "t" is the testing time.
  - The **reliability estimation process** gives the number of further failure free hours of testing needed to establish the desired failure density.
    Note: If a failure is detected during this time, you stop the clock and recalculate.
    Note: This model ignores operational profiles.
    Inputs needed:
    - **fd** = target failure density (e.g. 0.03 failures per 1000 LOC)
    - **tf** = total test failures observed so far
    - **th** = total testing hours up to the last failure

    Formula:

$$\frac{ln(fd/(0.5+fd)) * xh}{ln((0.5+fd)/(tf+fd))}$$

- **Fault Seeding:**
- **Fault seeding** is a technique for evaluating the effectiveness of a testing process. One or more faults are deliberately introduced into a code base, without informing the testers. The discovery of seeded faults during testing can be used to calibrate the effectiveness of the test process.
- The idea is that

$$\frac{\text{Detected seeded faults}}{\text{Total seeded faults}} = \frac{\text{Detected nonseeded faults}}{\text{Total nonseeded faults}}$$

and we can use this data to estimate test efficiency and to estimate the number of remaining faults.

Acceptance Testing:

- **Introduction to Acceptance Testing:**
- **Acceptance testing** is testing conducted to determine whether a system satisfies its acceptance criteria.
- There are two categories of acceptance testing:
    1. **User Acceptance Testing (UAT):** It is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed-off as meeting user needs. This is the final test performed. The main purpose of this testing is to validate the software against the business requirements. This validation is carried out by the end-users who are familiar with the business requirements.
    2. **Business Acceptance Testing (BAT):** It is undertaken within the development organization of the supplier to ensure that the system will eventually pass the user acceptance testing. This is to assess whether the product meets the business goals and purposes or not.
- **The 3 main goals of accepting testing are:**
    1. Confirm that the system meets the agreed upon criteria.
    2. Identify and resolve discrepancies, if there are any.
    3. Determine the readiness of the system for cut-over to live operations.
- **The acceptance criteria are defined on the basis of the following attributes:**
    1. Functional Correctness and Completeness
    2. Accuracy
    3. Data Integrity
    4. Data Conversion
    5. Backup and Recovery
    6. Competitive Edge
    7. Usability
    8. Performance
    9. Start-up Time
    10. Stress
    11. Reliability and Availability
    12. Maintainability and Serviceability
    13. Robustness
    14. Timeliness
    15. Confidentiality and Availability
    16. Compliance
    17. Installability and Upgradability
    18. Scalability
    19. Documentation
- **Selection of Acceptance Criteria:**
- The acceptance criteria discussed are usually too general, so the customer needs to select a subset of the quality attributes.
- The quality attributes are then prioritized to the specific situation.
- Ultimately, the acceptance criteria must be related to the business goals of the customer's organization.

- Example of an acceptance test plan

1. Introduction
2. Acceptance test category For each category of acceptance criteria (a) Operation environment (b) Test case specification (i) Test case Id# (ii) Test title (iii)Test objective (iv) Test procedure
3. Schedule
4. Human resources

- **Acceptance Test Execution:**
- The acceptance test cases are divided into two subgroups:
  - The first subgroup consists of basic test cases.
  - The second subgroup consists of test cases that are more complex to execute.
- The acceptance tests are executed in two phases:
  - In the first phase, the test cases from the basic test group are executed.
  - If the test results are satisfactory then the second phase, in which the complex test cases are executed, is taken up.
  - In addition to the basic test cases, a subset of the system-level test cases are executed by the acceptance test engineers to independently confirm the test results.
- Acceptance test execution activity includes the following detailed actions:
  - The developers train the customer on the usage of the system.
  - The developers and the customer coordinate the fixing of any problem discovered during acceptance testing.
  - The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy.
- The acceptance test engineer may create an Acceptance Criteria Change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier.
- An ACC report is generally given to the supplier's marketing department through the on-site system test engineers.
- E.g. of an ACC document

1. ACC Number:	A unique number
2. Acceptance Criteria Affected:	The existing acceptance criteria
3. Problem/Issue Description:	Brief description of the issue
4. Description of Change Required:	Description of the changes needed to be done to the original acceptance criterion
5. Secondary Technical Impacts:	Description of the impact it will have on the system
6. Customer Impacts:	What impact it will have on the end user
7. Change Recommended by:	Name of the acceptance test engineer(s)
8. Change Approved by:	Name of the approver(s) from both the parties

- **Acceptance Test Report:**
- The acceptance test activities are designed to reach at a conclusion:
  - Accept the system as delivered.
  - Accept the system after the requested modifications have been made.
  - Do not accept the system.
- Usually some useful intermediate decisions are made before making the final decision.
- A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing is not promising. If the test results are unsatisfactory, changes are made to the system before acceptance testing can proceed to the next phase.
- During the execution of acceptance tests, the acceptance team prepares a test report on a daily basis.
  I.e. During the execution of acceptance tests, a daily acceptance test report is made.
- E.g. of a daily acceptance test report

1. Date:	Acceptance report date
2. Test case execution status:	Number of test cases executed today Number of test cases passing Number of test cases failing
3. Defect identifier:	Submitted defect number Brief description of the issue
4. ACC number(s):	Acceptance criteria change document number(s), if any
5. Cumulative test execution status:	Total number of test cases executed Total number of test cases passing Total number of test cases failing Total number of test cases not executed yet

- At the end of the first and the second phases of acceptance testing an acceptance test report is generated.
- E.g. of a finalized acceptance test report

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval

- **Acceptance Testing in Extreme Programming:**
- In the XP framework, the user stories are used as the acceptance criteria.
- The user stories are written by the customer as things that the system needs to do for them.
- Several acceptance tests are created to verify the user story has been correctly implemented.
- The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results.
- A story is incomplete until it passes its associated acceptance tests.
- Ideally, acceptance tests should be automated, either using the unit testing framework, before coding.
- The acceptance tests take on the role of regression tests.

Static Analysis:

- Static analysis is a method of computer program debugging that is done by examining the code without executing the program.
- This process provides an understanding of the code structure and can help ensure that the code adheres to industry standards.
- Automated tools can assist programmers and developers in carrying out static analysis. The software will scan all code in a project to check for vulnerabilities while validating the code.
- Static analysis is generally good at finding coding issues such as:
    - Programming errors
    - Coding standard violations
    - Undefined values
    - Syntax violations
    - Security vulnerabilities
- Once the code is written, a static code analyzer should be run to look over the code. It will check against defined coding rules from standards or custom predefined rules. Once the code is run through the static code analyzer, the analyzer will have identified whether or not the code complies with the set rules. It is sometimes possible for the software to flag false positives, so it is important for someone to go through and dismiss any. Once false positives are waived, developers can begin to fix any apparent mistakes, generally starting from the most critical ones. Once the code issues are resolved, the code can move on to testing through execution.
- Example of static analysis tools include:
    - FindBugs
    - JLint
    - JSHint
- Different tools find different bugs.
- Benefits of using static analysis include:
    - It can evaluate all the code in an application, increasing code quality.
    - Automated tools are less prone to human error and are faster.
    - It will increase the likelihood of finding vulnerabilities in the code, increasing web or application security.
    - It can be done in an offline development environment.
- Drawbacks of using static analysis include:
    - False positives can be detected.
    - It will detect harmless bugs that may not be worth fixing.
    - A tool might not indicate what the defect is if there is a defect in the code.
    - Static analysis can't detect how a function will execute.