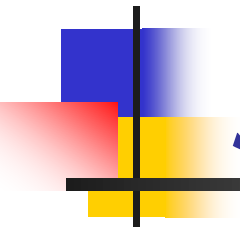
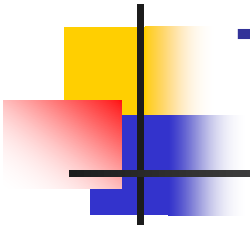


COMP 354: Introduction to Software Engineering

Software Integration Testing

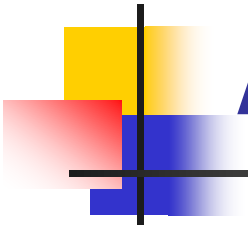
Based on Chapter 20 of the textbook



Testing Fundamentals

Attributes of a good test:

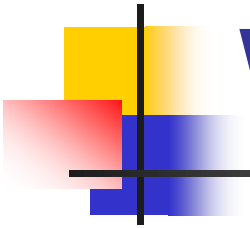
- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be "best of breed."
- A good test should be neither too simple nor too complex.



Approaches to Testing

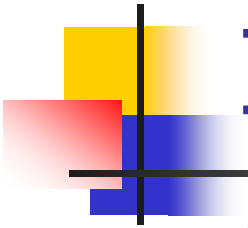
Any engineered product can be tested in one of two ways:

1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.
2. Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.



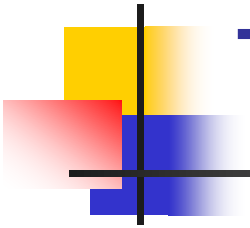
White Box Integration Testing

- White-box testing, is an integration testing philosophy that uses implementation knowledge of the control structures described as part of component-level design to derive test cases.
- White-box tests can be only be designed after source code exists and program logic details are known.
- Logical paths through the software and collaborations between components are the focus of white-box integration testing.
- Important data structures should also be tested for validity after component integration.



Integration Testing

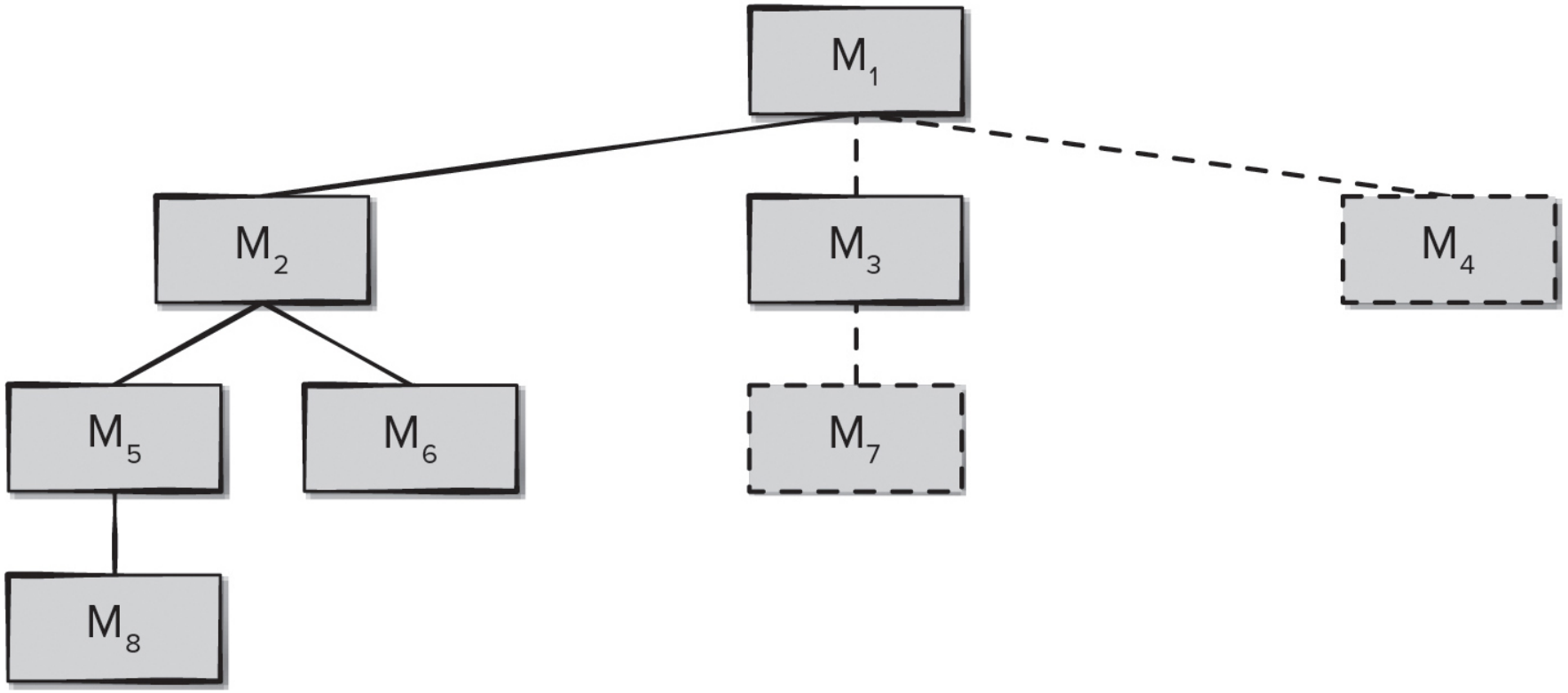
- Integration testing is a systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that matches the design.
- In the big bang approach, all components are combined at once and the entire program is tested as a whole. Chaos usually results!
- In incremental integration a program is constructed and tested in small increments, making errors easier to isolate and correct. Far more cost-effective!

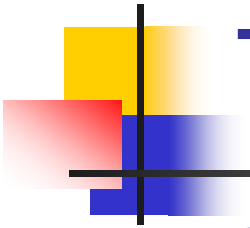


Top-Down Integration

- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate to the main control module are incorporated into the structure followed by their subordinates.
- Depth-first integration integrates all components on a major control path of the program structure before starting another major control path.
- Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally before moving down to the next level of subordinates.

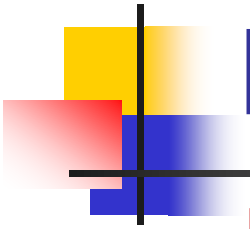
Top-Down Integration





Top-Down Integration Testing

- The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (for example, depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real component.
- Regression testing may be conducted to ensure that new errors have not been introduced.

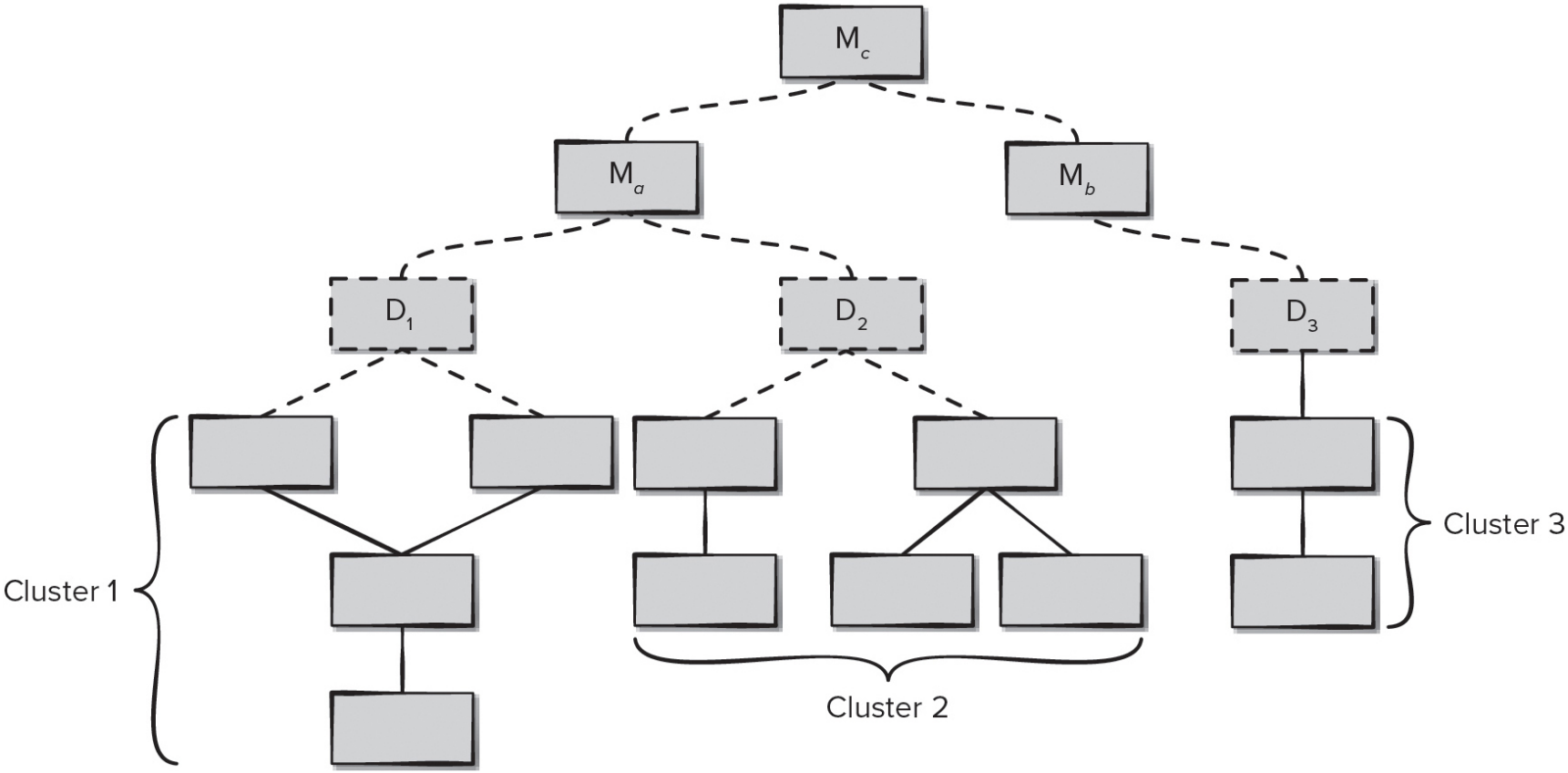


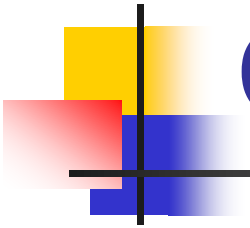
Bottom-Up Integration Testing

Bottom-up integration testing, begins construction and testing with atomic modules components at the lowest levels in the program structure.

- Low-level components are combined into clusters (builds) that perform a specific software subfunction.
- A driver (a control program for testing) is written to coordinate test-case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined, moving upward in the program structure.

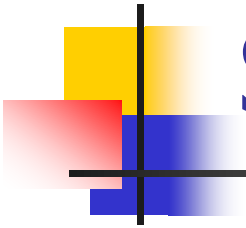
Bottom-Up Integration





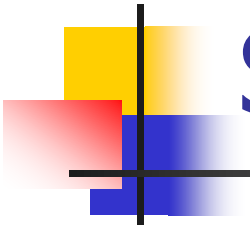
Continuous Integration

- Continuous integration is the practice of merging components into the evolving software increment at least once a day.
- This is a common practice for teams following agile development practices such as X P or DevOps. Integration testing must take place quickly and efficiently if a team is attempting to always have a working program in place as part of continuous delivery.
- Smoke testing is an integration testing approach that can be used when software is developed by an agile team using short increment build times.



Smoke Testing Integration

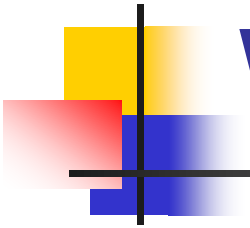
- Software components that have been translated into code are integrated into a build. – that includes all data files, libraries, reusable modules, and components required to implement one or more product functions.
- A series of tests is designed to expose "show-stopper" errors that will keep the build from properly performing its function cause the project to fall behind schedule.
- The build is integrated (either top-down or bottom-up) with other builds, and the entire product (in its current form) is smoke tested daily.



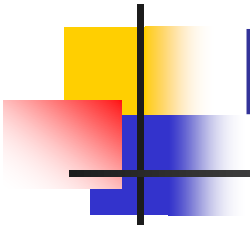
Smoke Testing Advantages

- Integration risk is minimized, since smoke tests are run daily.
- Quality of the end product is improved, functional and architectural problems are uncovered early.
- Error diagnosis and correction are simplified, errors are most likely in (or caused by) the new build.
- Progress is easier to assess, each day more of the final product is complete.
- Smoke testing resembles regression testing by ensuring newly added components do not interfere with the behaviors of existing components.

Integration Testing Work Products

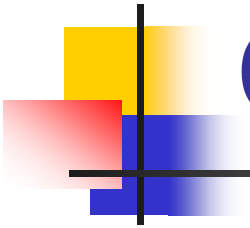


- An overall plan for integration of the software and a description of specific tests is documented in a test specification.
- Test specification incorporates a test plan and a test procedure and becomes part of the software configuration.
- Testing is divided into phases and incremental builds that address specific functional and behavioral characteristics of the software.
- Time and resources must be allocated to each increment build along with the test cases needed.
- A history of actual test results, problems, or peculiarities is recorded in a test report and may be appended to the test specification.
- It is often best to implement the test report as a shared Web document to allow all stakeholders access to the latest test results and the current state of the software increment.



Regression Testing

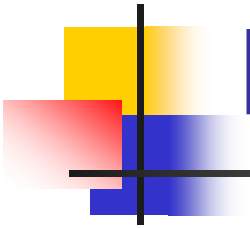
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.
- AI tools may be able to help select the best subset of test cases to use in regression automatically based on previous experiences of the developers with the evolving software product.



OO Integration Testing

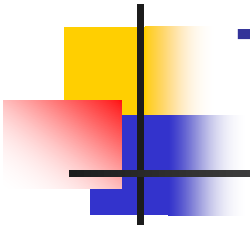
- Thread-based testing, integrates the set of classes required to respond to one input or event for the system.
  - Each thread is integrated and tested individually.
  - Regression testing is applied to ensure no side effects occur.
- Use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few server classes.
  - The next layer classes, (called dependent classes) use the independent classes are tested next.
  - This sequence of testing layers of dependent classes continues until the entire system is constructed.

OO Testing – Fault-Based Test Case Design



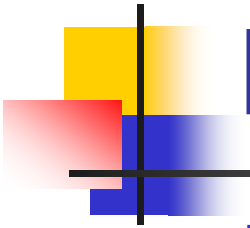
- The object of fault-based testing is to design tests that have a high likelihood of uncovering plausible faults.
- Because the product or system must conform to customer requirements, fault-based testing begins with the analysis model.
- The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.
- To determine whether these faults exist, test cases are designed to exercise the design or code.

Fault-Based OO Integration Testing



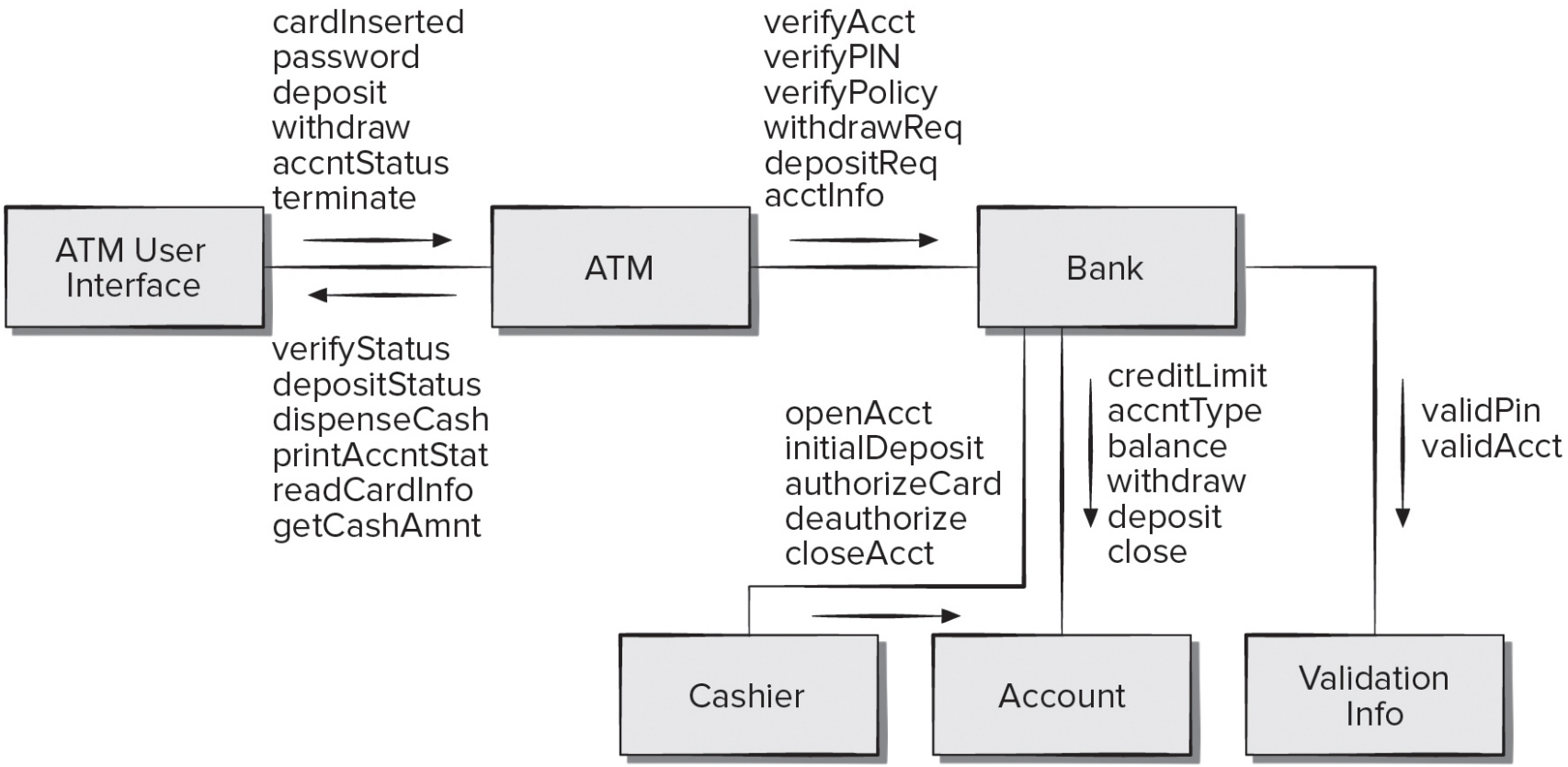
- Fault-based integration testing looks for plausible faults in operation calls or message connections:
  - unexpected result
  - wrong operation/message used
  - incorrect invocation
- Integration testing applies to attributes and operations – class behaviors are defined by the attributes.
- Focus of integration testing is to determine whether errors exist in the calling (client) code, not the called (server) code.

OO Testing – Fault-Based Test Case Design



- Scenario-based testing uncovers errors that occur when any actor interacts with the software.
- Scenario-based testing concentrates on what the user does, not what the product does.
- This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes must begin.

Collaboration Diagram for Banking Application

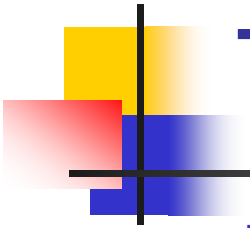


OO Testing – Random Test Case Design

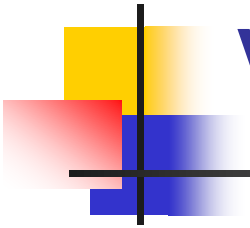
- For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
- For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
- For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
- For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.
- A random test case for the Bank class might be:

  Test case r3 = verifyAcct·verifyPIN·depositReq

OO Testing – Scenario-Based Test Case Design

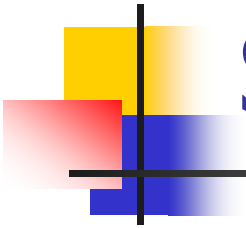


- Scenario-based testing uncovers errors that occur when any actor interacts with the software.
- Scenario-based testing concentrates on what the user does, not what the product does.
- This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes must begin.



Validation Testing

- Validation testing tries to uncover errors, but the focus is at the requirements level - on user visible actions and user-recognizable output from the system.
- Validation testing begins at the culmination of integration testing, the software is completely assembled as a package and errors have been corrected.
- Each user story has user-visible attributes, and the customer's acceptance criteria which forms the basis for the test cases used in validation-testing.
- A deficiency list is created when a deviation from a specification is uncovered and their resolution is negotiated with all stakeholders.
- An important element of the validation process is a configuration review (audit) that ensures the complete system was built properly.



Software Testing Patterns

- Testing patterns are described in much the same way as design patterns.
- Example:

  Pattern name: **ScenarioTesting**

  Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The ScenarioTesting pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement.