Validation:
- It is ensuring that the **specification** is correct.
- It is determining that the software to be built is actually what the user wants.
- It is building the right software.
- It is making sure that the end software is what the user wants.

Verification:
- It is ensuring that the software runs correctly.
- It is building the software right.
- It is making sure the software works.

Specification:
- **Functional specifications** define the actions and operations of a system and can be verified by software tests.
- Non-functional specifications deal with performance and usability. They require special tests.

Testing:
- The aim of testing is to locate and repair bugs.
- **Note:** Testing only reveals the presence of defects, it never proves their absence. No matter how much testing you do, you can't be sure that there isn't an error in your program.
- An alternative to testing is **formal verification**. Formal verification is using formal logic to prove that software is correct.
  Currently, formal verification:
  - Is prohibitively expensive
  - Has little automated support
  - Is mainly manual techniques
  - Is error prone

  Formal verification is only feasible when cost of failure is extreme, usually when failure leads to loss of life.
  Examples of when we use formal verification are:
  - Air and spacecraft control
  - Medical systems
  - Nuclear plants
- Testing is the least glamorous part of software development and possibly the most expensive.
- If testing is not done thoroughly, it could be very disastrous. Estimates of the economic cost of software failure produce astronomic numbers (In 2002, this number was $59.5 billion USD. Furthermore, about 10% of projects are abandoned entirely.
- Inadequate design or poor coding produces many timebombs in the system.

Approach to Testing:
- Steps:
  1. Coding's finished
  2. Run a few tests
  3. System passes
  4. Release

Testing Strategies:

1. **Exhaustive Testing:**

- For exhaustive testing, we try all possible inputs.
- This approach is simple but naive.
- This approach only works for very small input sets.

2. **Efficient Testing:**

- This divides the tests into **equivalence classes** where only one representative of each class needs to be tested.
- All other tests of inputs in the same equivalence class just repeat the first one.
- Dramatic reduction in total number of tests.
- E.g. If you have something like:

```
if (x < 2):
  // do something
else:
   // do something else
```

  You don't have to test all inputs that are less than 2 or all inputs that are greater than or equal to 2. You can just use a few select cases.

3. **Black Box Testing:**

- Testing without reference to the internal structure of the component or system.
  I.e. Specification is available but no code.
- Equivalence classes are derived from rules in the specification.
- When generating the equivalent classes, make sure to test with illegal values.
- Black Box testing will not usually cover all the special cases required to test data structures.

4. **White Box Testing:**

- Code is available and can be analyzed.
- Equivalence classes are derived from rules in the specification and the code.
- When generating the equivalent classes, make sure to test with illegal values.
- **Code coverage** is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determine a quantitative measure of code coverage.
- Here are some types of code coverage:
  - **Statement:** Each statement is executed at least once.
  - **Branch:** Each branch is traversed at least once.
  - **Condition:** Each boolean expression has been evaluated to both TRUE and FALSE.
  - **Branch/Condition:** Tests both Branch and Condition coverage.
  - **Compound Condition:** All combinations of condition values at every branch statement are covered.
  - **Path:** All program paths are traversed at least once.

Kanban:

- **Kanban** literally means "visual card," "signboard," or "billboard."
- Kanban is a popular framework used to implement agile software development. It requires real-time communication of capacity and full transparency of work. Work items are represented visually on a kanban board, allowing team members to see the state of every piece of work at any time.
- Toyota originally used Kanban cards to limit the amount of inventory tied up in work in progress on a manufacturing floor.

- Inside an iteration, effort across roles is uneven. Development work often continues throughout a cycle while testing starts late and never seems to get enough time. Using a Kanban approach in software drops time-boxed iterations in favor of focusing on continuous flow.
- Here are some challenges that time-boxed iterative development has:
    - Short time-boxes give more frequent opportunity to measure progress and inspect software but force development items to be smaller.
    - Smaller development items are often too small to be valuable and difficult to identify.
    - Quality of requirements suffers as analysts rush to prepare for upcoming cycles.
    - Quality of current development suffers when busy analysts are unable to inspect software or answer questions during development.
    - Quality often suffers as testers race to complete work late in the development time-box.
- Benefits of kanban:
    - Planning flexibility
    - Shortened time cycles
    - Fewer bottlenecks
    - Visual metrics
    - Continuous delivery
- How to set up a kanban board:
    1. Define a work process flow.
    2. Lay out a visual kanban board.
    3. Decide on limits for items in the queue and work in progress.
    4. Place prioritized goals on the left column of the board.
    5. Start the board by placing stories or features in a queue.
    6. Move features through the process flow as work is completed.
    7. Use the dates on the cards to calculate cycle time.