Software Quality: Validation and Verification

V & V

- Validation:
  - Are we building the right product?
    The software should do what the user really requires
- Verification
  - Are we building the product right?
    The software should conform to its specification

V & V must be applied at each stage in the software process

- Objectives
  - discover defects in a system;
  - assess whether the system is useful and usable in an operational situation

V & V Goals

- should establish *confidence* that the software is fit for purpose

does NOT mean completely free of defects

  - it must be *good enough* for its intended use
  - the type of use will determine the degree of *confidence* that is needed

Approaches to V & V

- Software inspections
  - analysis of the static system representation to discover problems *(static verification)*
  - may be supplement by tool-based document and code analysis
- Software testing
  - exercising and observing product behaviour *(dynamic verification)*
  - the system is executed with test data and its operational behaviour is observed

Inspection vs. Testing

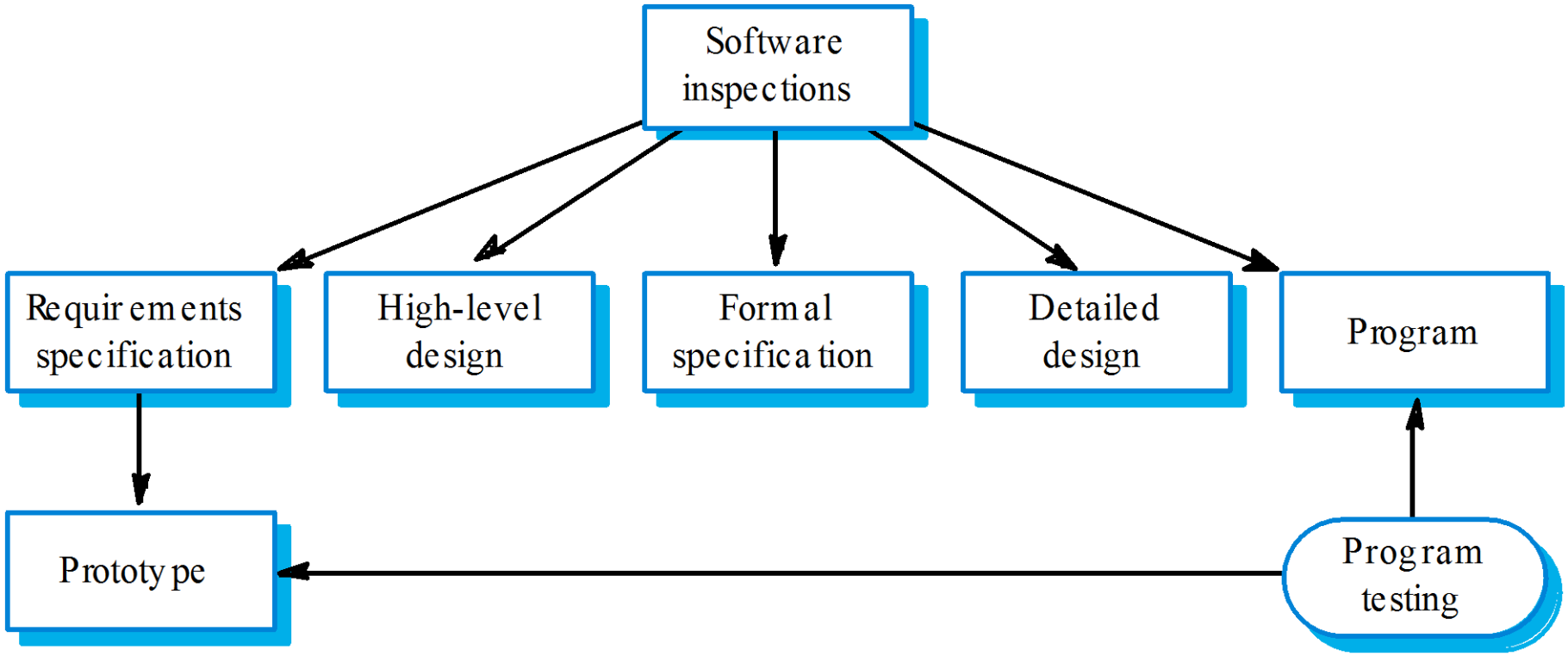
- Inspections and testing are *complementary*, not *opposing* verification techniques

- Both should be used during the V & V process.

  - Inspection:
    - checks conformance with a *specification* but not conformance with the customer's real requirements.
    - cannot check non-functional characteristics such as performance, usability, etc.

  - Testing:
    - can reveal the *presence* of errors NOT their *absence*.
    - *the only validation technique for non-functional requirements*
      → the software has to be executed to see how it behaves

Inspection vs. Testing



Inspection

Software inspection

- people examine the source representation to discover anomalies and defects
- does not require execution of a system:
  - may be used before implementation
- may be applied to any representation of the system
  - requirements,
  - design,
  - configuration data,
  - test data

effective technique for discovering program errors

Inspection checks (1)

Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a de limiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?

Inspection checks (2)

Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Automated static analysis

- Static analysers
  - software tools for source text processing
  - parse the program text and try to discover potentially erroneous conditions
  - very effective as an aid to inspections
  - supplement to but not a replacement for inspections

currently included in most popular IDEs

Static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Testing

Software testing

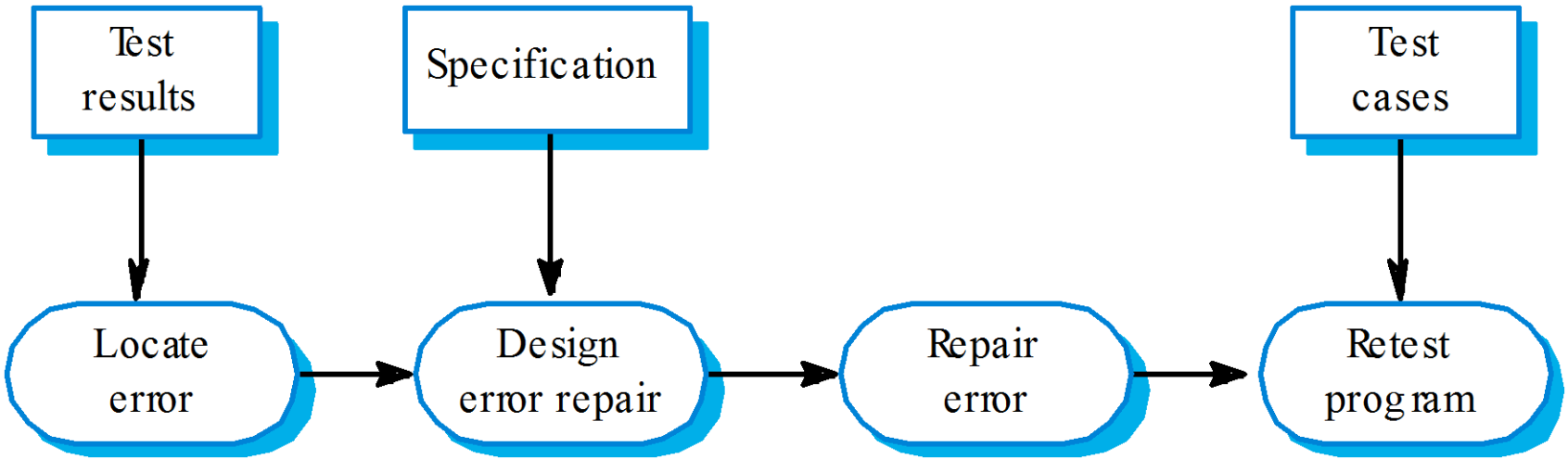
- Types of testing:
  - Defect testing
    - Tests designed to discover system defects
    - A *successful* defect test is one which reveals the presence of defects in a system.
  - Validation testing
    - Intended to show that the software meets its requirements
    - A *successful* test is one that shows that a requirements has been properly implemented.
- Defects analysis:

debugging

Testing vs. debugging

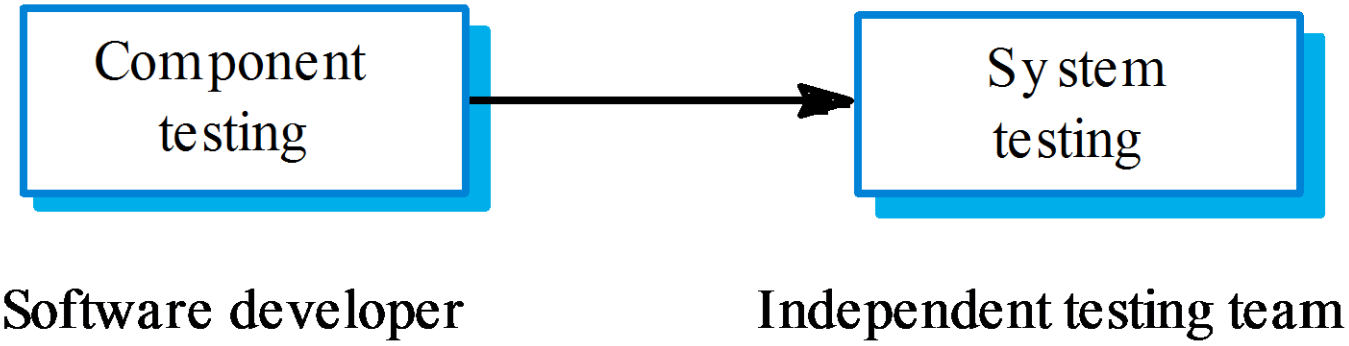
- Testing and debugging are distinct processes:
  - Testing, as operation in V & V, is concerned with establishing the existence of defects in a program.
  - Debugging is concerned with locating and repairing these errors.
- Debugging means:
  1. *formulating a hypothesis* about program behaviour
  2. *testing this hypothesis* to find the error

Debugging process

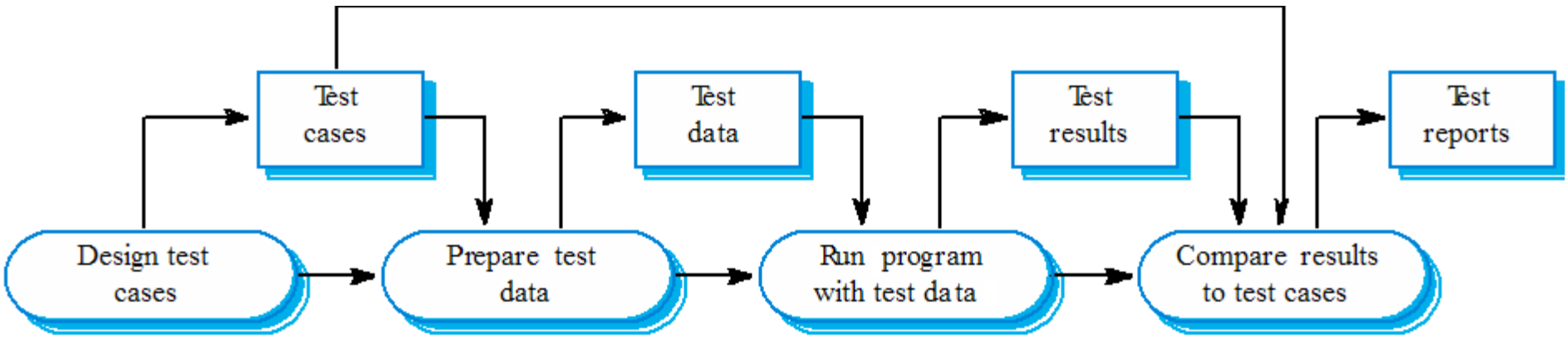


Testing process

- Component testing
  - Testing of individual program components
  - Done by the component developer
  - Tests – derived from the developer's experience.
- System testing
  - Testing groups of components, integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification.



Testing process



Testing policies

- Only exhaustive testing can show a program is free from defects.

Exhaustive testing is impossible

- Testing policies define the approach to be used in selecting system tests.

For instance:

- All functions accessed through menus should be tested;
- Combinations of functions accessed through the same menu should be tested;
- All functions must be tested with correct and incorrect user input.

Component testing

- Also called **unit testing**
  - Done by the programmer herself
  - Based on the experience of the programmer
- Goal: expose faults in components
- Unit tests can be small functions
  - that call functions from components' API
  - that are customarily written *before* the API is implemented.

System testing

- Integration testing
  - The test team has access to the system source code
  - The system is tested as components are integrated.
- Release testing
  - The test team test the complete system to be delivered as a black-box.

Other types of testing

- Regression testing
  - Phase in integration testing
  - After new components have been integrated, all tests must be run again
  - New code is not accepted until all tests run successfully.
  - May be demanded after bug fixes, extensions, etc.
- Performance testing
  - Proving that the system meets its requirements (especially non-functional)
  - Sub-step: **stress testing**
    - The system is run with tests that go beyond the specified limits.

Other types of testing

- Smoke testing
  - Non-exhaustive set of tests
  - For proving that *the most important functions work*.
  - Any failed test case is a stopper
    - The software is sent back to developers.
  - If no smoke test fails, the testing team proceeds with further testing.

Testing guidelines

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause buffers to overflow;
- Repeat the same input several times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

Management
in Software Processes
(Introduction)

Management areas

- **Managing people**
- Managing costs
- Managing quality
- Managing configurations

What is most important…

- …for a software company?
  - the team (employees, workers)
  - the profit
  - the clients
  - the projects
  - the corporate culture

What is most important…

- …for a software company?
  - **the team (employees, workers)**
  - the profit
  - the clients
  - the projects
  - the corporate culture

Managing people

People in SE

- *they are the most important assets*
- People management factors:
  - Consistency
    - Team members should all be treated in a comparable way without favourites or discrimination.
  - Respect
    - Different team members have different skills and these differences should be respected.
  - Inclusion
    - Involve all team members and make sure that people's views are considered.
  - Honesty
    - You should always be honest about what is going well and what is going badly in a project.

Topics in people management

- Selecting staff
- Motivating people
- Managing groups

Selecting staff

- Information on selection comes from:
  - Information provided by the candidates
  - Information gained by interviewing and talking with candidates
  - Recommendations and comments from other people who know or who have worked with the candidates

Considerations

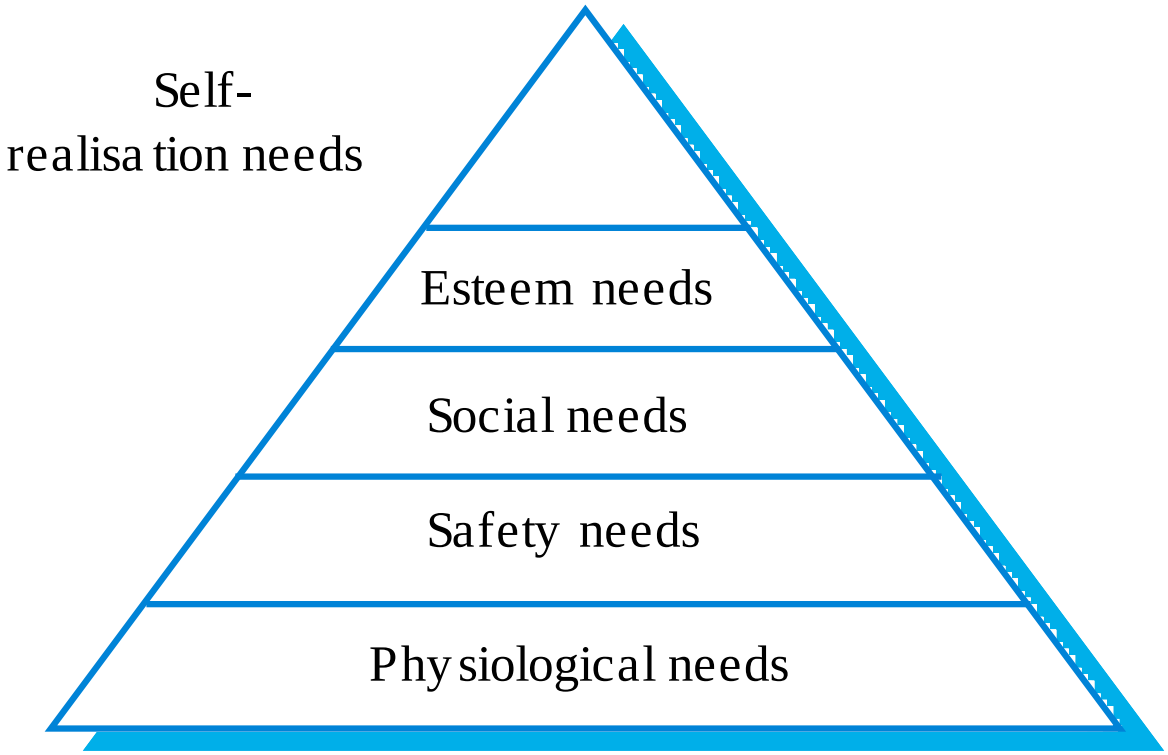
- Part-time involvement may be inevitable.
- UI design and hardware interfacing skills are in short supply.
- Recent graduates may not have specific skills but may be a way of introducing new skills.
- Technical proficiency may be less important than social skills.

Factors in selecting staff

- Experience:
  - Application domain
  - Platform
  - Programming language
- Abilities
  - Problem solving
  - Communication
- Educational background
- Adaptability
- Attitude
- Personality

Motivating people

- Motivation is based on people's needs



Needs and their satisfaction

- Social
  - Provide communal facilities;
  - Allow informal communications.
- Esteem
  - Recognition of achievements;
  - Appropriate rewards.
- Self-realization
  - Training - people want to learn more;
  - Responsibility.

Managing groups

- Software engineering is a group activity
  - non-trivial software projects cannot be completed by one person working alone.
- Group interaction – key determinant of group performance.
- Flexibility in group composition is limited
  - Managers must do the best they can with available people.

Factors

- Group composition.
- Group cohesiveness.
- Group communications.
- Group organization.

Homework

- Your team is responsible with developing the communication modules of ADMSys (see slide 20, lecture 6). Many features of this module are triggered by events or combination of events on the GUI, which you have as a black-box.
  - What are the difficulties of defining and ***automatically*** running series of test cases for these features?
  - How can the communication with the devices' PLCs be tested?