TVVS – Test, Verification and Validation of Software

Integration and System Testing

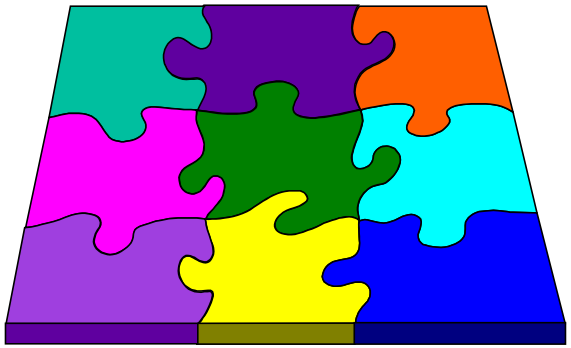
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

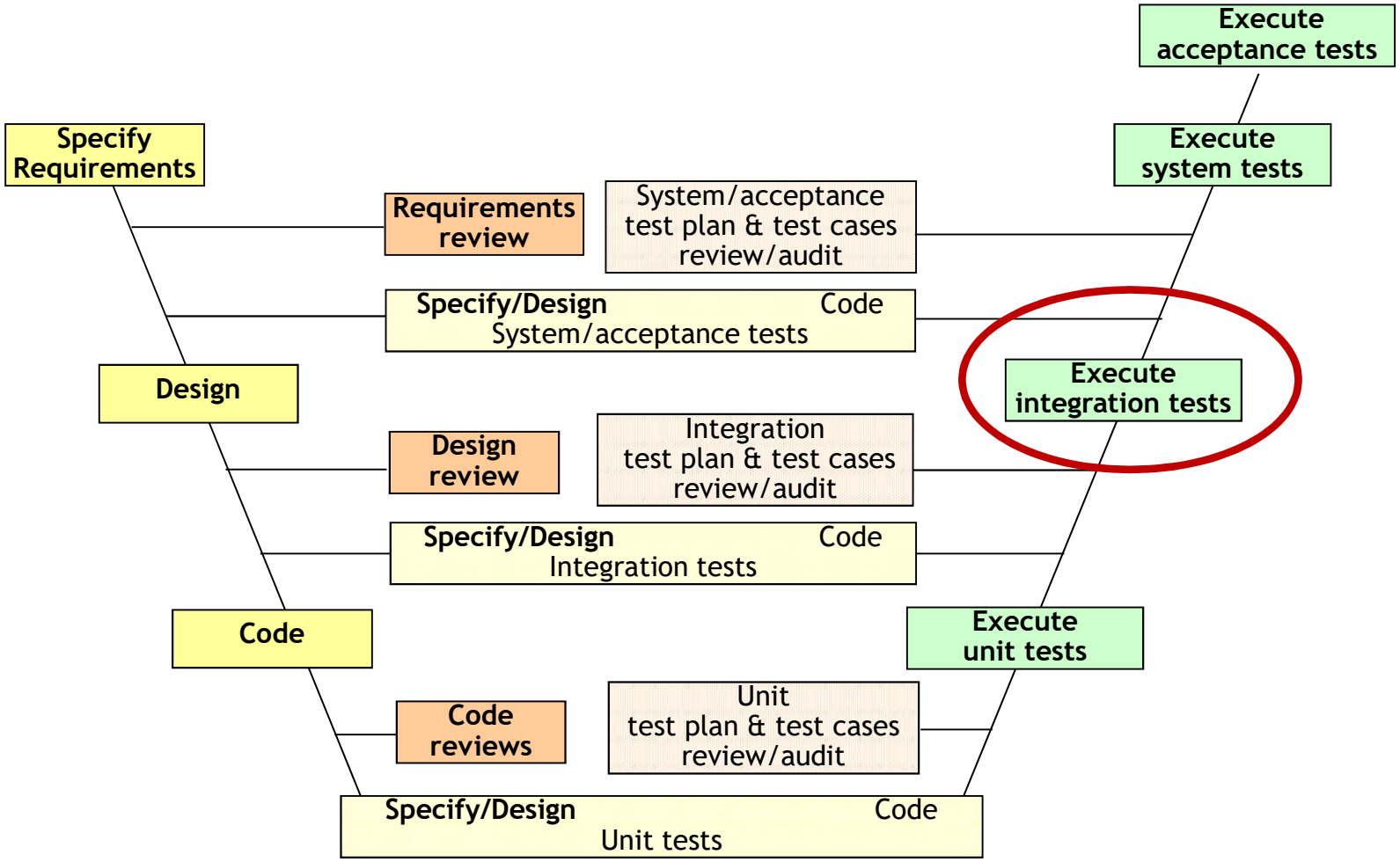
Agenda

- Integration testing
- System testing
- Regression testing

Integration testing



V-model



The extended V-model of software development [I.Burnstein]

Integration testing

- Testing of groups of components integrated to create a sub-system. Components should be tested previously.
- Usually the responsibility of an independent testing team (except sometimes in small projects)
- Integration testing should be black-box testing with tests derived from the technical specification
- A principal goal is to detect defects that occur on the interfaces of units
- Main difficulty is localising errors
- *Incremental* integration testing (as opposed to *big-bang* integration testing) reduces this difficulty

Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces
  - Sub-systems request services from other sub-systems

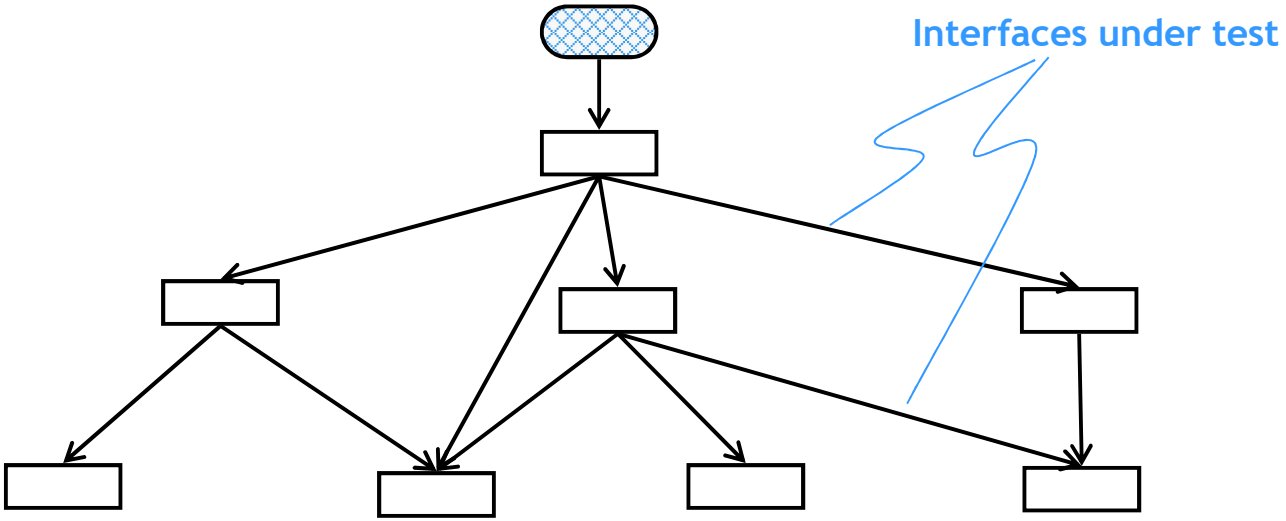
Interface errors

- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated
- …

Big-bang integration testing



Incremental integration testing

- Top-down integration testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up integration testing
  - Integrate individual components in levels until the complete system is created
- Sandwich Testing
  - Combination of Top-down with Bottom-up testing
- Collaboration integration testing
  - Appropriate for iterative development strategies where software components are created and fatten as new use cases are implemented (through a collaboration of objects and components)
  - Scenario based testing
- The integration testing strategy must follow the software construction strategy

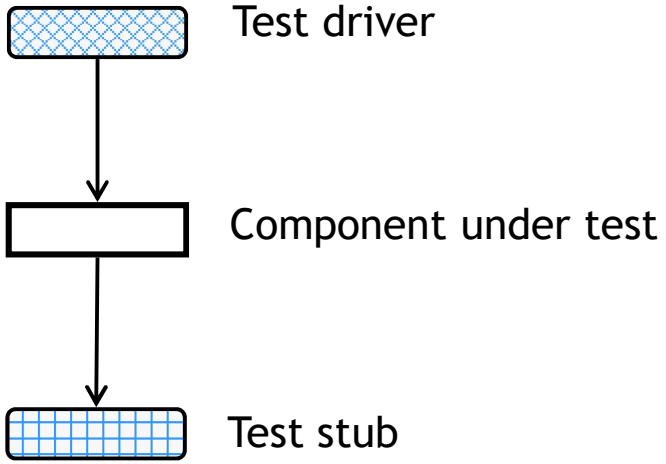
Integration testing – More strategies

- **thread testing**
  - An approach to component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy.
- **functional integration**
  - An integration approach that combines the components or systems for the purpose of getting a basic functionality working early.
- **neighborhood integration testing**
  - A form of integration testing where all of the nodes that connect to a given node are the basis for the integration testing.
- **pairwise integration testing**
  - A form of integration testing that targets pairs of components that work together, as shown in a call graph.

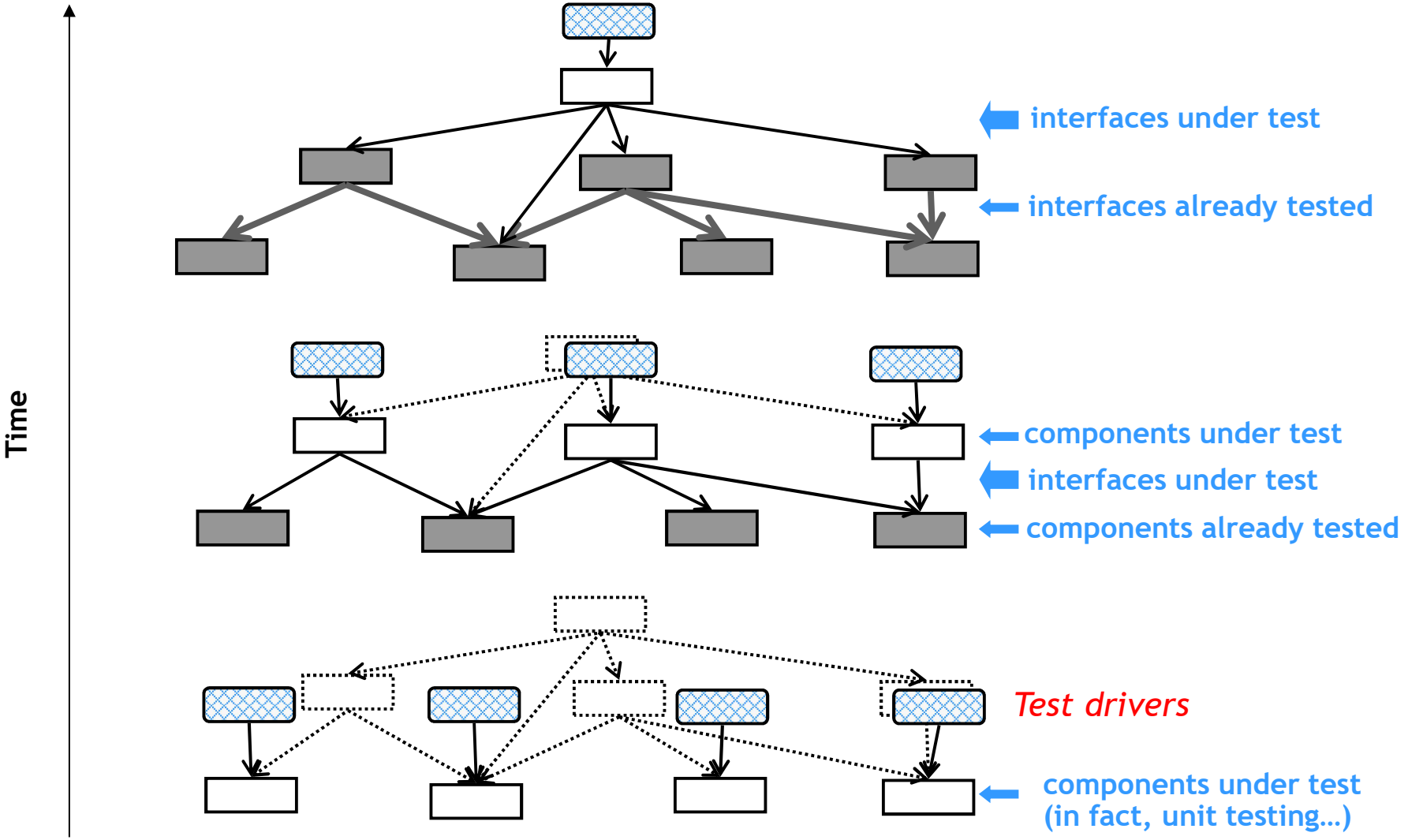
[ISTQB]

Test harness: drivers and stubs

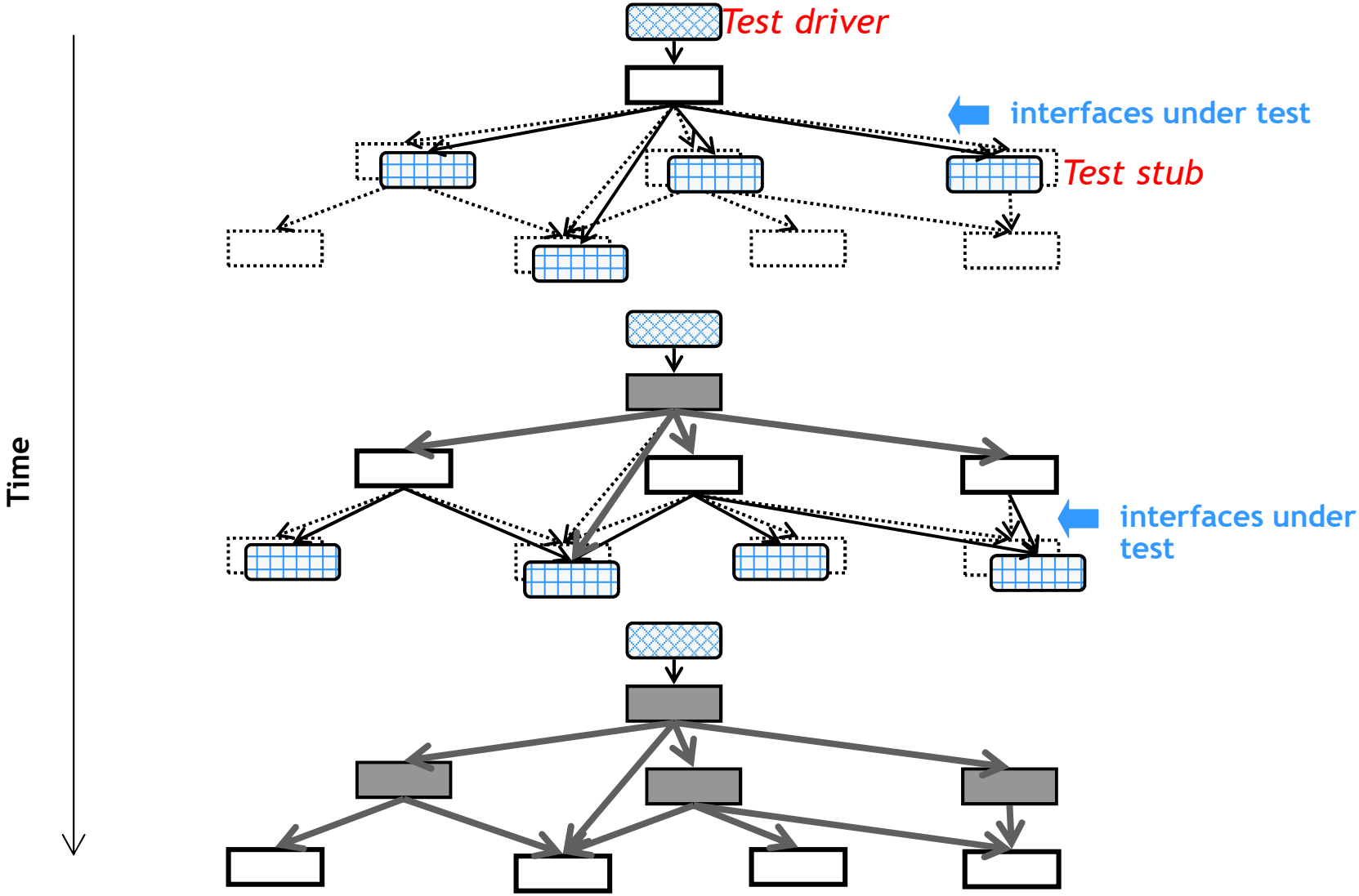
- Test harness: auxiliary code developed to support testing
- Test drivers
  - Call the target code, simulating calling units or a user
  - In automatic testing: implementation of test cases and procedures
- Test stubs
  - Simulate modules/units/systems called by the target code
  - Mock objects can be used for this purpose



Bottom-up integration testing



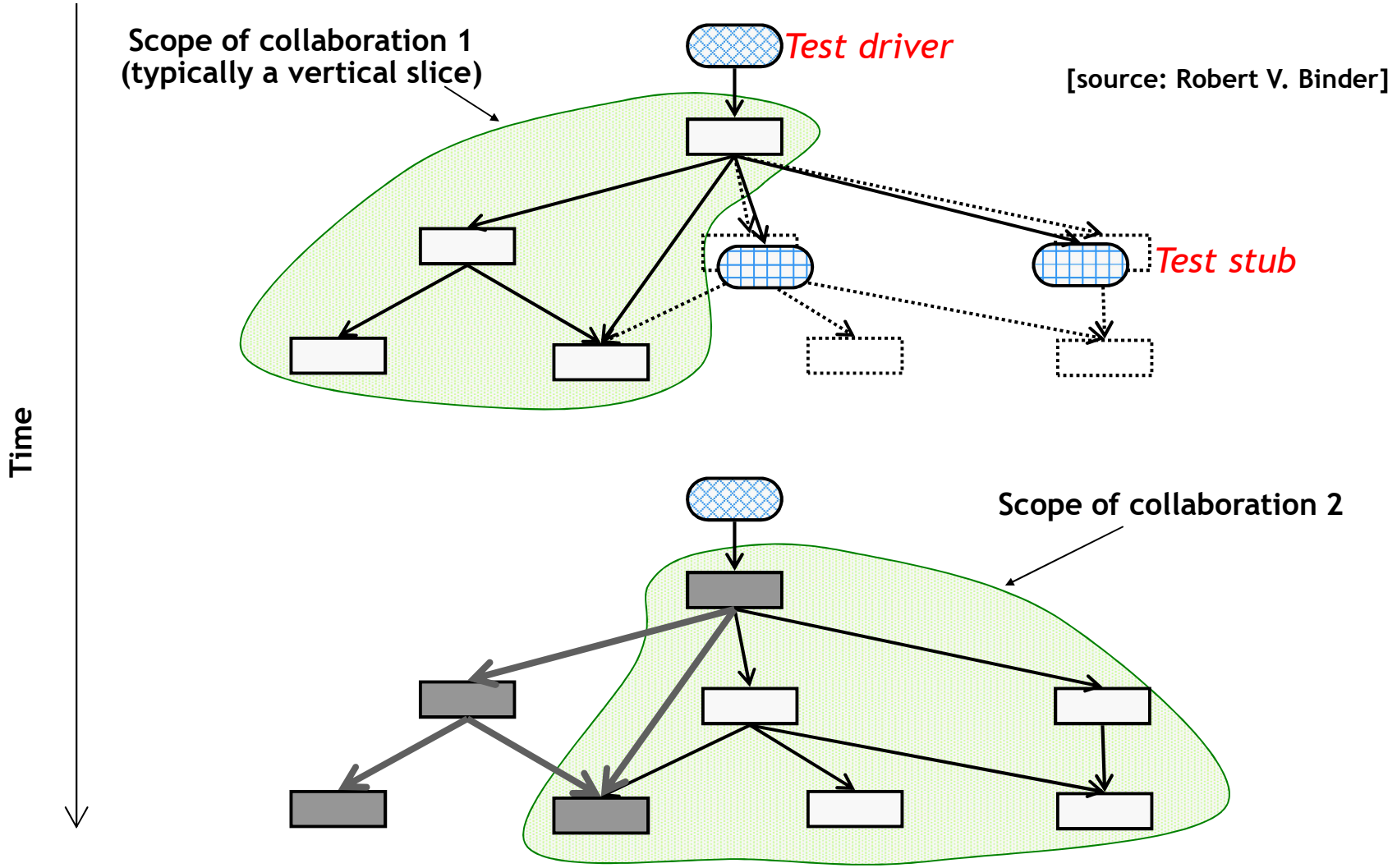
Top-down integration testing



Top-down versus bottom-up

- Design validation
  - Top-down integration testing is better at discovering design flaws that can be fixed earlier
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Top-down integration requires the development of complex stubs to drive significant data upward while bottom-up integration requires drivers. Often a combination of approaches known as sandwich
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Collaboration integration testing



Integration testing

- Top-down approach
  - Requires the highest-level modules be test and integrated first.
  - This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers.
  - Getting I/O functions early can ease test writing.
  - Early demonstration of the main functionality.
  - However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle.
  - Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.

Integration testing

- Bottom-up approach
  - Requires the lowest-level units be tested and integrated first.
  - Units are tested early in the development process and the need for stubs is minimized.
  - Test conditions may be easier to create.
  - The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late.
  - Major faults may be detected towards the end of the integration process.
  - Can not observe a functional system till the end of the process.
  - Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.

Integration testing

- Mixed approach
  - Requires testing along functional data and control-flow paths.
  - First, the inputs for functions are integrated in the bottom-up pattern discussed before.
  - The outputs for each function are then integrated in the top-down manner.
  - The primary advantage of this approach is the degree of support for early release of limited functionality.
  - It also helps minimize the need for stubs and drivers.
  - The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

Integration Testing - common problems

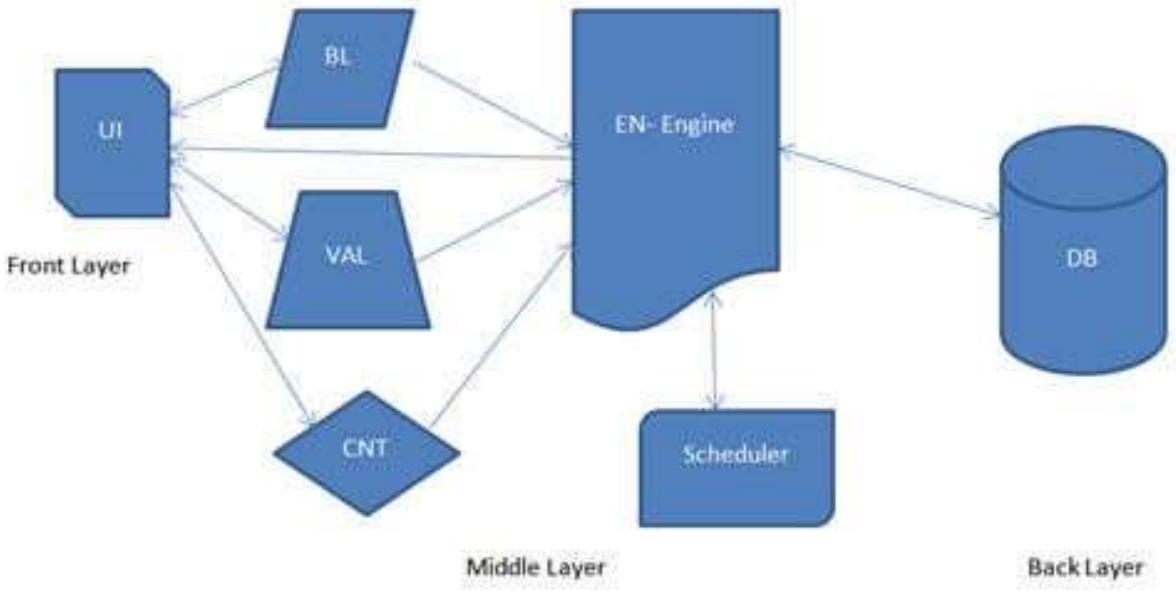
Problem	Description
Insufficient integration test environments (test beds)	Lack of sufficient test beds due to cost or hardware needed elsewhere causes competition for limited resources among test teams
Insufficient schedule	Insufficient time scheduled for adequate testing
Poor test bed fidelity	Insufficient sw, hw, and system fidelity to actual system (e.g., due to inaedquate sw simulations/drivers, prototype or wrong version)
Unavailable components	Not all the components to be integrated are ready at the proper time
Poor code quality	Code contains many defects that should have been found during lower-level testing, which unnecessarily slows integration testing
...	

Integration testing vs unit testing

Unit test	Integration test
Depends on code	Depends on external systems
Easy to write and verify	Setup of integration test might be complicated
Units tested in isolation	More than one component are tested
Dependencies are mocked if needed	No mocking (only unrelated componenents are mocked)
Verifies implementation code	Verifies interconection behaviour when components are used together
Most used by developers	Also useful to QA, DevOps, Help Desk
Failed test is a code problem (if the requirements didn't change)	Failed test may mean that the code is correct but the environment changed
Should be fast	May take long

Example

- **UI** – User Interface module, which is visible to the end user, where all the inputs are given.
  BL – Is the Business Logic module, which has all the all the calculations and business specific methods.
  VAL – Is the Validation module, which has all the validations of the correctness of the input.
  CNT – Is the content module which has all the static contents, specific to the inputs entered by the user. These contents are displayed in the reports.
  EN – Is the Engine module, this module reads all the data that comes from BL, VAL and CNT module and extracts the SQL query and triggers it to the database.
  Scheduler – Is a module which schedules all the reports based on the user selection (monthly, quarterly, semiannually & annually)
  DB – Is the Database.



https://www.softwaretestinghelp.com/what-is-integration-testing/

Example

Integration testing focuses on the flow of data between the modules

- **The questions here are:**
  - **How** the BL, VAL and the CNT module **will read and interpret** the data entered in the UI module?
  - Is BL, VAL and CNT module **receiving the correct** data from UI?
  - In **which format the data** from BL, VAL and CNT is transferred to the EQ module?
  - **How** will the EQ **read the data** and extract the query?
  - Is the query **extracted correctly**?
  - Is the Scheduler **getting the correct data** for reports?
  - Is the **result set** received by the EN, from the database is **correct** and as expected?
  - Is EN **able to send the response back** to the BL, VAL and CNT module?
  - Is UI module **able to read** the data and display it appropriately to the interface?

Example

- **Few other sample test conditions can be as follows:**
  - Are the menu options generating the correct window?
  - Are the windows able to invoke the window under test?
  - For every window, identify the function calls for the window that the application should allow.
  - Identify all calls from the window to other features that the application should allow
  - Identify reversible calls: closing a called window should return to the calling window.
  - Identify irreversible calls: calling windows closes before called window appears.
  - Test the different ways of executing calls to another window e.g. – menus, buttons, keywords.

Example

- **Steps to Kick off Integration Tests**
  - Understand the architecture of your application.
  - Identify the modules
  - Understand what each module does
  - Understand how the data is transferred from one module to another.
  - Understand how the data is entered and received into the system (entry point and exit point of the application)
  - Segregate the application to suit your testing needs.
  - Identify and create the test conditions
  - Take one condition at a time and write down the test cases.

System testing

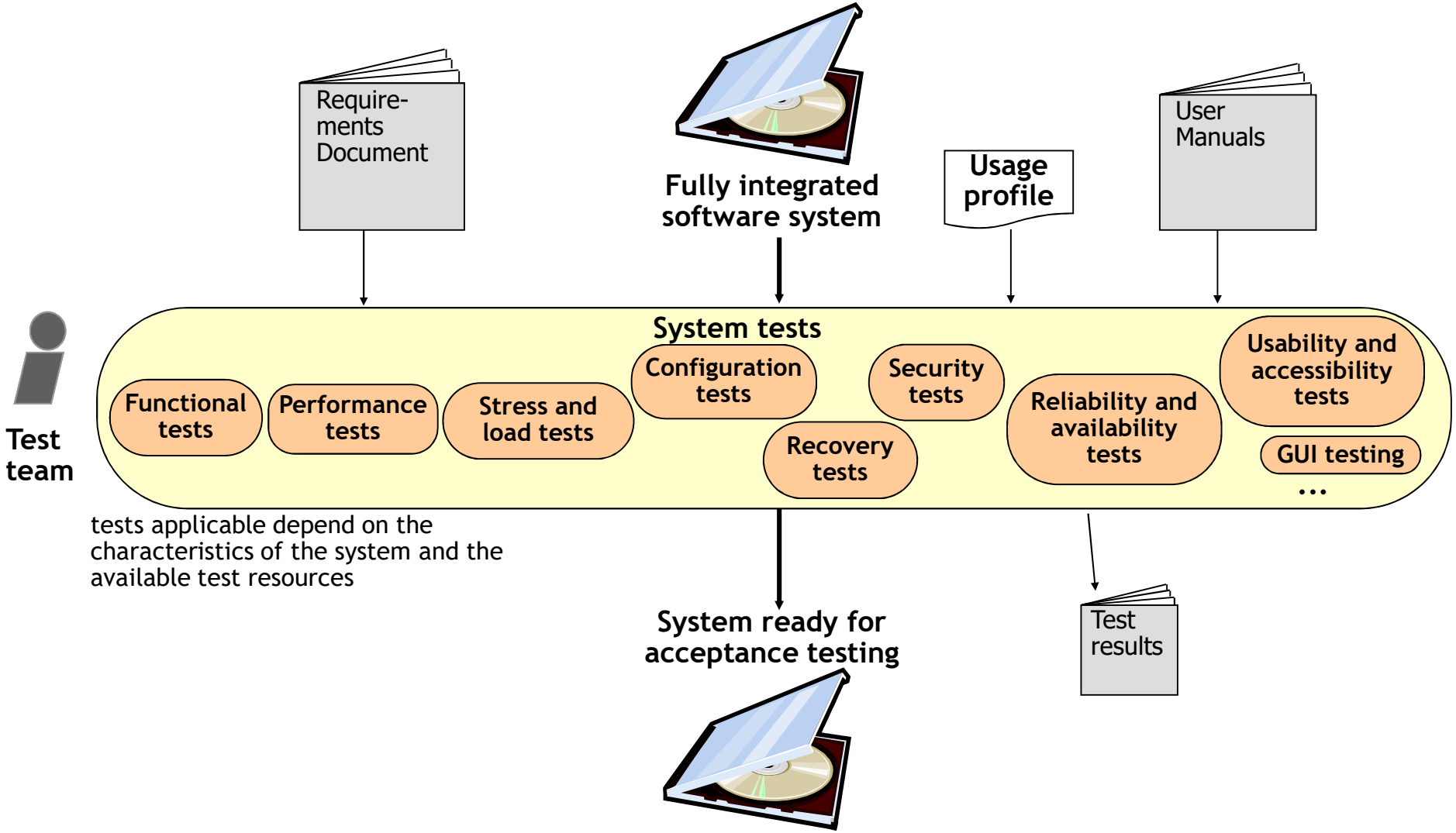
System testing

- Testing the system as a whole by an independent testing team
- Often requires many resources: laboratory equipment, long test times, etc.
- Usually based on a requirements document, specifying both functional and non-functional (quality) requirements
- Preparation should begin at the requirements phase with the development of a master test plan and requirements-based tests (black-box tests)

System testing

- The goal is to ensure that the system performs according to its requirements, by evaluating both functional behavior and quality requirements such as reliability, usability and performance.
- Especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage
- Tests implemented on the parts and subsystems may be reused/repeated, and additional tests for the system as a whole may be designed

Types of system testing



Functional testing

- Ensure that the behavior of the system adheres to the requirements specification
- Black-box in nature
- Equivalence class partitioning, boundary-value analysis and state-based testing are valuable techniques
- Document and track test coverage with a (tests to requirements) traceability matrix
- A defined and documented form should be used for recording test results from functional and other system tests
- Failures should be reported in test incident reports
  - Useful for developers (together with test logs)
  - Useful for managers for progress tracking and quality assurance purposes

Performance testing

- Goals:
  - See if the software meets the performance requirements
  - See whether there are any hardware or software factors that impact on the system's performance
  - Provide valuable information to tune the system
  - Predict the system's future performance levels
- Results of performance tests should be quantified, and the corresponding environmental conditions should be recorded

Performance testing

- Resources usually needed
  - a source of transactions to drive the experiments, typically a load generator
  - an experimental test bed that includes hardware and software the system under test interacts with
  - instrumentation of probes that help to collect the performance data (event logging, counting, sampling, memory allocation counters, etc.)
  - a set of tools to collect, store, process and interpret data from probes

Stress and load testing

- Load testing – maximize the load imposed on the system (volume of data, number of users, ...)
  - Examples:
    - a system is required to handle 10 interrupts / second and the load causes 20 interrupts/second
    - a suitcase being tested for strength and endurance is stomped by a multi tonne elephant
    - testing a word processor by editing a very large document

Stress and load testing

- Stress testing – minimize the resources available to the system (processor, memory, disk space, ...). Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation that are not revealed under normal testing conditions
  - Examples:
    - run processes that consume resources (CPU, memory, disk, network) on the Web and database servers
    - take the database offline, then restart it
- The goal is to try to break the system, find the circumstances under which it will crash, and provide confidence that the system will continue to operate correctly (possibly with bad performance but with correct functional behavior) under conditions of stress
- Supported by many of the resources used for performance testing

Configuration testing

- Configuration testing checks for failure of the system to perform under all of the combinations of hw and sw configurations. Typical sw systems interact with multiple hw devices such as disc drives, tape drives, and printers.
- Objectives [Beizer]:
  - show that all the configuration changing commands and menus work properly
  - show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions
  - show that the systems' performance level is maintained when the devices are interchanged, or when they fail

Configuration testing

- Types of test to be performed:
  - rotate and permute the positions of devices to ensure physical/logical device permutations work for each device
  - induce malfunctions in each device, to see if the system properly handles the malfunction
  - induce multiple device malfunctions to see how the system reacts

Security testing

- Evaluates system characteristics that relate to the availability, integrity and confidentiality of system data and services
- Computer software and data can be compromised by
  - criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy
  - errors on the part of honest developers/maintainers (and users?) who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge
- Both can be perpetuated by those inside and outside on an organization
- Areas to focus: password checking, legal and illegal entry with passwords, password expiration, encryption, browsing, trap doors, viruses, ...
- Usually the responsibility of a security specialist

Recovery testing

- Drive a system to losses of resources in order to determine if it can recover properly from these losses
- Especially important for transaction systems
- Example: loss of a device during a transaction
- Tests would determine if the system could return to a well-known state, and that no transactions have been compromised
  - Systems with automated recovery are designed for this purpose
- Areas to focus [Beizer]:
  - **Restart** – the ability of the system to restart properly on the last checkpoint after a loss of a device
  - **Switchover** – the ability of the system to switch to a new processor, as a result of a command or a detection of a faulty processor by a monitor

Reliability and availability testing

- **Software reliability** is the probability that a software system will operate without failure under given conditions for a given interval
  - May be measured by the mean time between failures (MTBF)
  - MTBF = MTTF (mean time to failure) + MTTR (mean time to repair)
- **Software availability** is the probability that a software system will be available for use
  - May be measured by the percentage of time the system is on or **uptime** (example: 99,9%)
  - A = MTTR / MTBF
- Low reliability is compatible with high availability in case of low MTTR
- Requires statistical testing based on usage characteristics/profile
  - During testing, the system is loaded according to the usage profile
- More information: Ilene Burnstein, section 12.5
- Usually evaluated only by high maturity organizations

Usability testing

- Usability tests are concerned with external Properties: The user's perspective:
  - **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
  - **Learnability.** degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
  - **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
  - **User error protection.** Degree to which a system protects users against making errors.
  - **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
  - **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Accessibility testing

- Accessibility testing is the technique of making sure that your product is accessibility compliant.
- Typical accessibility problems can be classified into following four groups, each of them with different access difficulties and issues:
  - **Visual impairments** such as blindness, low or restricted vision, or color blindness. User with visual impairments uses assistive technology software that reads content loud. User with weak vision can also make text larger with browser setting or magnificent setting of operating system
  - **Motor skills** such as the inability to use a keyboard or mouse, or to make fine movements
  - **Hearing impairments** such as reduced or total loss of hearing
  - **Cognitive abilities** such as reading difficulties, dyslexia or memory loss

GUI testing

- Raises specific challenges
  - GUI test automation is harder than API test automation
  - Observing visible GUI state is difficult
  - Observing invisible GUI state is tricky almost impossible
  - Controlling GUI actions is difficult
    - Event based
    - Non-solicited events
  - State space and test case explosion
    - Multiple ways (mouse, keyboard,…) to achieve the same goal
    - Almost all user actions are enabled most of the time

GUI errors – concrete examples

- Correct functioning
- Missing commands
- Correct window modality
- Mandatory fields
- Incorrect field defaults
- Data validation
- Error handling (messages to the user)
- Wrong values retrieved by queries
- Filling order
- (…)

GUI testing: manual techniques

- Heuristic Methods
  - A group of specialists studies the interface in order to find problems that they can identify.
- Guidelines
  - Recommendations about user interfaces. E.g.: how to organize the display and the menu structure.
- Cognitive walkthrough
  - The developers walk through the interface in the context of core tasks a typical user will need to accomplish. The actions and the feedback of the interface are compared to the user's goals and knowledge, and discrepancies between user's expectations and the steps required by the interface are noted.
- Usability tests
  - The interface is studied under real-world or controlled conditions (real users), with evaluators gathering data on problems that arise during its use.

GUI testing: manual techniques

- Advantages
  - More bugs found per test cases executed (good specialist) (not necessarily by time or money invested)
  - Adaptability: bugs found provide hints to find other bugs
  - Can find bugs (e.g., usability problems) that are difficult to find with automated tests (the converse is also true)
  - Can be supported / made more systematic/repeatable by checklists of standard tests and application specific tests
  - Good for exploratory / initial testing
- Disadvantages
  - Regression testing
  - Effort required
  - Weak coverage
  - Repeatability / reproducibility
  - Good test specialists are difficult to find
  - Depends on the capabilities of the tester

Automated GUI testing approaches

- Capture-Replay tools
  - WinRunner, Rational Robot, Android
- Random input testing tools
  - Rational's TestFactory uses dumb monkey method

- Unit testing frameworks
  - JUnit, NUnit
- Model-based testing tools
  - Spec Explorer (API testing)
  - Spec Explorer with GUI testing extensions
  - Guitar (Atif Memon)

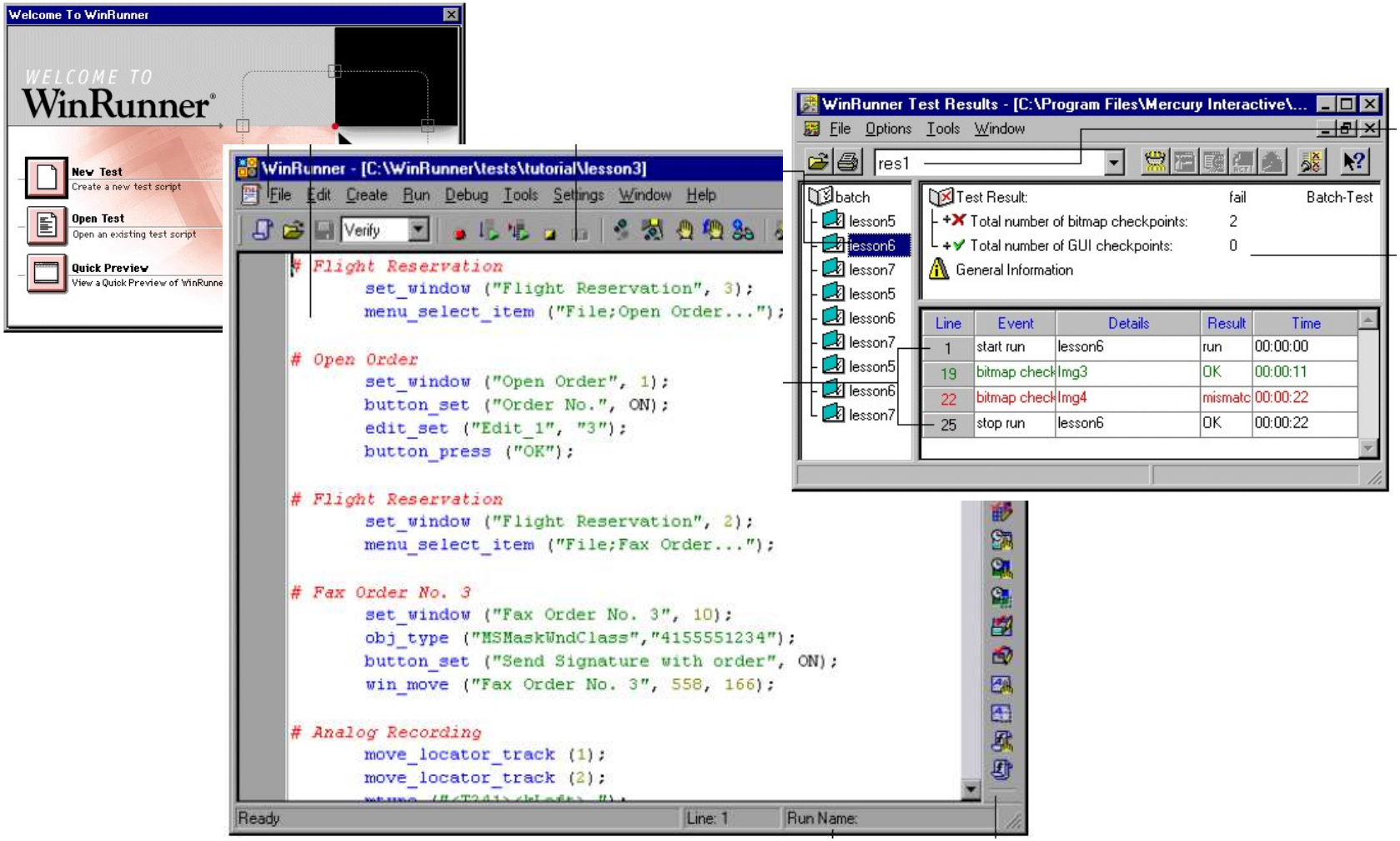
Capture replay tools

- How it works
  - Two different interacting modes
    - Capture
      - The user/tester interacts with the tool
      - The tools saves user actions and output
    - Replay
      - Saved user actions are reproduced
      - The output obtained is compared with the one expected
  - Three different execution modes
    - Position based
    - Object based
    - Mixed

Capture replay tools

- Output verification
  - Property/content comparison
  - Bitmap comparison
  - Optical character recognition
- Scripts
  - Saved user actions
  - From scratch
  - Mixed
  - A higher level of abstraction is needed to reach independence of GUI updates

Capture replay tools



Capture replay tools

- Advantages
  - Test case recording (capture) (similar to macro recording in Excel)
  - Recorded test scripts can be made more generic by programming
    - E.g., use logical names in test script and map logical names to physical objects in a separate file
  - Automatic test case execution (replay)
  - Output checking
  - Good for regression testing
- Disadvantages
  - Can be used only when the software application is working correctly
  - Sensitivity to physical details
  - Do not support automatic generation of test cases

Random testing tools

- Actions performed randomly without knowledge of how humans use the application
- Microsoft says that 10 to 20% of the bugs in Microsoft projects are found by these tools
- Two kinds of tools
  - Dumb monkeys – low IQ; they can't recognize an error when they see one
  - Smart monkeys – generate inputs with some knowledge to reflect expected usage; get knowledge from state table or model of the AUT.

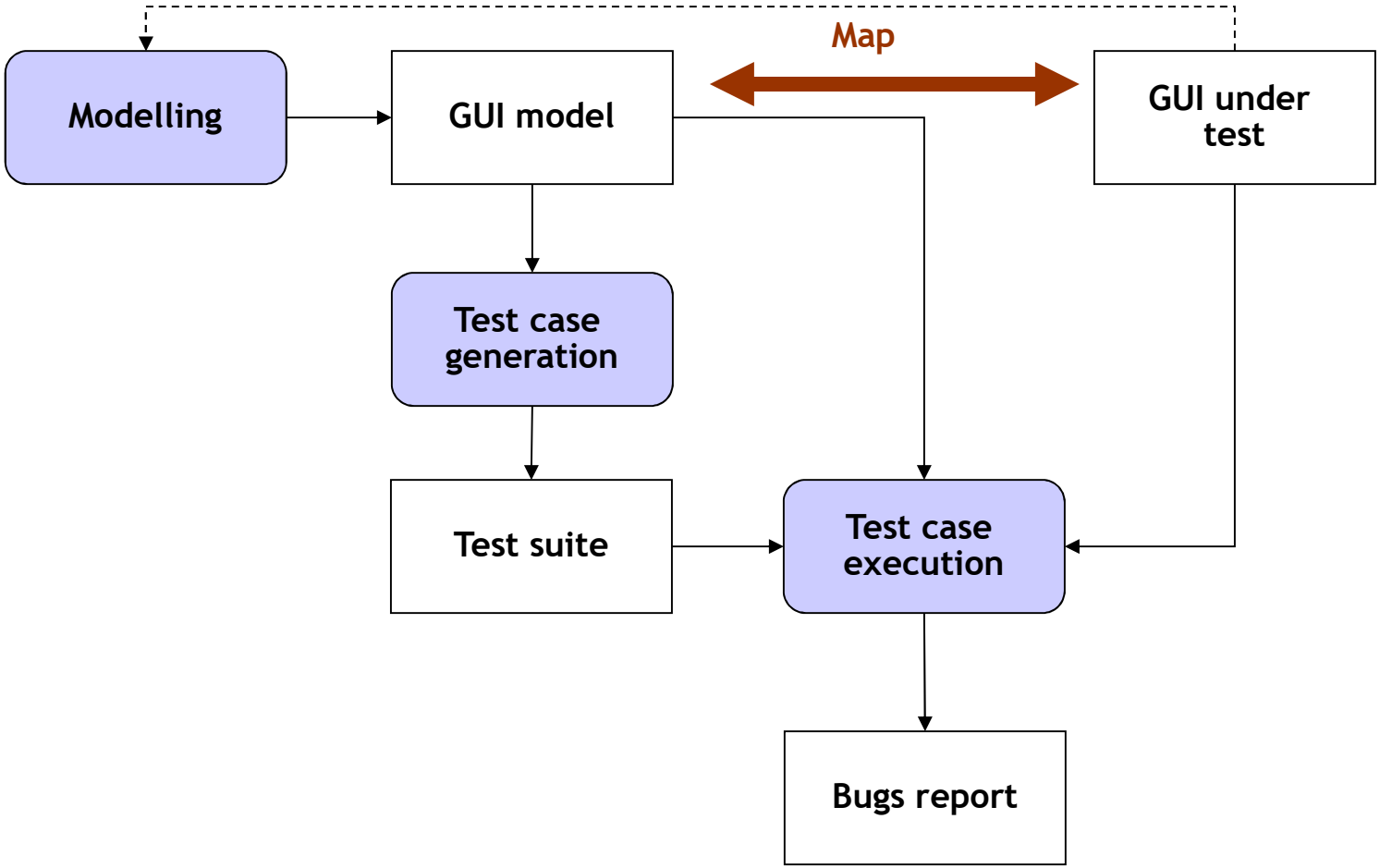
Random input testing tools

- Advantages
  - Good for finding system crashes
  - No effort in generating test cases
  - Independent of GUI updates
  - Increase confidence on the sw when running several hours without finding errors
  - “Easy” to implement
- Disadvantages
  - Not good for finding other kinds of errors
  - Difficult to reproduce the errors
  - Unpredictable

Unit testing frameworks

- GUI testing transformed into API testing
  - E.g., JUnit, NUnit
- Advantages
  - Used in combination with appropriate GUI test libraries (e.g. Maui for windows, Jemmy and Abbot for Java GUI applications), essentially reduce GUI testing to API testing
  - Flexibility
  - Automatic test execution
  - Support TDD (Test Driven Development)
- Disadvantages
  - Test cases are programmed manually
  - Appropriate GUI test libraries not always available

Model-based testing tools



Model-based testing tools

- Advantages
  - Higher degree of automation (test case generation)
  - Allows more exhaustive testing
  - Good for correctness/functional testing
  - Model can be easily adapted to changes
- Disadvantages
  - Requires a formal specification/model
  - Test case explosion
  - Test case generation has to be controlled appropriately to generate a test case of manageable size
  - Small changes to the model can result in a totally different test suite

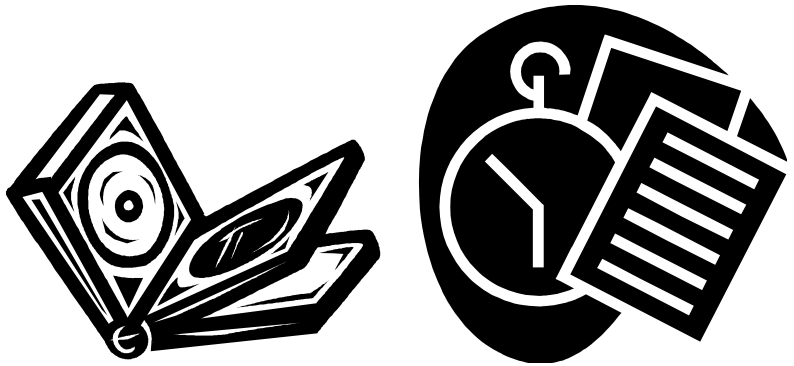
Automated GUI testing - vision

- Combination of approaches
  - ***Abstract layer*** separates logical names from physical properties of GUI objects and can be used in several approaches
  - ***GUI instrumentation library*** is reused by capture-replay tools, unit testing frameworks and model-based testing tools
  - ***Recording techniques*** describe how high-level user actions described in a model are mapped to concrete actions in the application (to see next)
  - ***Integrated environment*** supports all the testing automation approaches (also supporting automated static analysis)

GUI testing tools

- In the Marketplace:
  - CompuWare TestPartner - www.compuware.com
  - Rational Robot
  - Rational Visual Test - www.rational.com
  - WinRunner
  - LoadRunner - www.mercuryinteractive.com
  - Segue's SilkTest - www.segue.com
- Open Source
  - Abbot
  - GUITAR
  - Pounder
  - qftestJUI
  - Android - www.wildopensource.com/activities/larry-projects/android.php
  - GUI test drivers – www.testingfaqs.org/t-gui.html

Regression testing



Regression testing

- **regression testing:** Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed. [ISTQB]

- Note: Regression testing is not a testing level

Regression testing

- Regression testing is not a level of testing, but it is the *retesting of software* that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes **[Source: Burstein]**
- Regression tests are especially important when multiple software releases are developed **[Source: Burstein]**
- Sometimes the execution of all tests is not feasible so there is the need to select a subset of those tests in order to reduce the time for regression testing. Some techniques to support such selection are execution trace, execution slice and test prioritization **[source: Aditya P. Mathur]**

Regression testing

- Advantages
  - It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code.
  - It ensures that the bugs found earlier are NOT creatable.
  - Regression testing can be done by using the automation tools
  - It helps in improving the quality of the product.
- Disadvantages
  - If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again.
  - Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality.

Regression testing vs retesting

- **Regression testing** is carried out to ensure that the existing functionality is working fine and there are no side effects of any new change or enhancements done in the application. In other words, Regression Testing checks to see if new defects were introduced in previously existing functionality.
- **Retesting** is carried out in software testing to ensure that a particular defect has been fixed and it's the functionality working as expected.

Regression testing vs retesting

Regression testing	Retesting
To find issues which may be introduced because of any change or modification in the application	To confirm whether a failed test is working fine after being fixed
To check if changes did not introduce new bugs in existing functionality	To ensure that a particular bug was resolved and the functionality is working as exepected
Even passed tests are executed	Only failed test cases are reexecuted
To check for unexpected side effects	To ensure original issue is working as expected
Verification of bugs are not included in the regression testing	Verification of bugs are included in the retesting
Based on the project and availability of resourced, Regression testing can be carried out parallel with Retesting	Priority of retesting is higher than regression testing, so it is carried out before regression testing
...	

References and further reading

- R. Jeffries, J. R. Miller, C. Wharton, and K. M. Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques," 1991
- M. Y. Ivory and M. A. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces," *ACM Computing Surveys*, vol. 33, pp. 470-516, 2001.
- http://www.geocities.com/model_based_testing/
- Model-based testing tools
  - http://www-verimag.imag.fr/~async/TGV
  - http://research.microsoft.com/fse/AsmL
  - http://research.microsoft.com/SpecExplorer/
  - A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," Pittsburgh, 2001.
- N. Nyman, "Using Monkey Test Tools," in *STQE - Software Testing and Quality Engineering Magazine*, 2000.
- Unit testing frameworks - www.nunit.org; www.junit.org
- http://www.sqa-test.com/White_Paper.doc
- J. Edvardsson, "A Survey on Automatic Test Data Generation," 1999.
- http://research.microsoft.com/projects/specsharp/
- K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition / Decision Coverage," NASA/TM-2001-210876, 2001.

References and further reading

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- Software testing techniques (2nd ed.), by Boris Beizer, ISBN:0-442-20672-0, 1990
- A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing," presented at ICFEM'05, 2005.
- A. C. R. Paiva, J. C. P. Faria, and R. M. Vidal, "Automated Specification-based Testing of Interactive Components with AsmL," presented at 5th edition of the Quatic (Quality: the bridge to the future in ICT) international conference, Porto, 2004.
- A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal, "Modeling and Testing Hierarchical GUIs," presented at ASM 2005 - 12th International Workshop on Abstract State Machines, Paris - France, 2005.