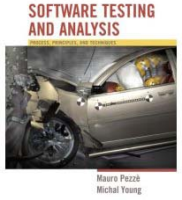
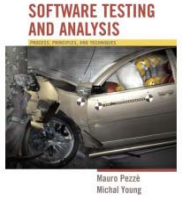
Automating Analysis and Test



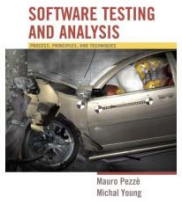
Learning objectives

- Understand the main purposes of automating software analysis and testing
- Identify activities that can be fully or partially automated
- Understand cost and benefit trade-offs in automation
- Separate publicity from important features in descriptions of commercial A&T tools



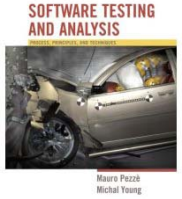
Three Potential Roles of Automation

- Necessary for introducing a task
  - example: coverage tools enable measuring structural coverage of test suites
- Useful to reduce cost
  - example: capture and replay tools reduce the costs of reexecuting test suites
- Useful to increase (human) productivity
  - example: software inspection is a manual activity, but tools to organize and present information and manage communication increase the productivity of people



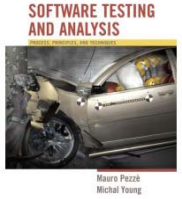
Approaching Automation

- Prioritize automation steps based on
  - variations in impact, maturity, cost, scope of the technology
  - fit and impact on the organization and process
- Three (non-orthogonal) dimensions for automation
  - value and current cost of the activity
  - extent to which the activity requires or is made less expensive by automation
  - cost of obtaining or constructing tool support



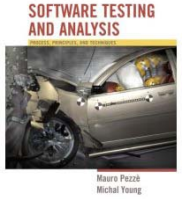
Automation Costs Vary Enormously

- Some tools are so simple to develop that they are justifiable even if their benefits are modest
  - example: generate test cases from finite state machine models
- Some tools that would be enormously valuable are simply impossible
  - example: identify exactly which parts of a program can never be executed (a provably undecidable problem)



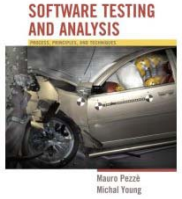
Costs May Depend on Scope

- Sometimes a general-purpose tool is only marginally more difficult to produce than a tool specialized for one project
  - example: general capture and replay for Windows applications vs capture and replay for a specific Windows application
  - Investment in the general-purpose tool, whether to build it or to buy it, can be amortized across projects
- In other cases, simple, project-specific tools may be more cost effective
  - Tool construction is often a good investment in a large project
  - example: simulators to permit independent subsystem testing



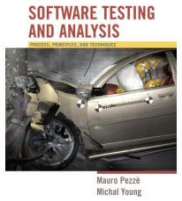
Focusing Where Automation Pays

- Simple repetitive tasks are often straightforward to automate
  - humans are slow and make errors in repetitive tasks
- But …judgment and creative problem solving remain outside the domain of automation
- Example: Humans are
  - Very good at identifying relevant execution scenarios that correspond to test case specifications
  - Very inefficient at generating large volumes of test cases or identifying erroneous results within a large set of outputs from regression tests
- Automating the repetitive portions of the task reduces costs, and improves accuracy as well



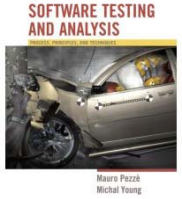
Planning: The Strategy Level

- Prescribes tools for key elements of the quality process
- Can include detailed process and tool prescriptions
- Recommends different tools contingent on aspects of a project
  - (application domain, development languages, size, overall quality,…)
- Often included in the A&T strategy: tools for
  - Organizing test design and execution
  - Generating quality documents
  - Collecting metrics
  - Managing regression test suites
- Less often included: tools for
  - Generating test cases
  - Dynamic analysis

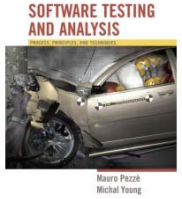


Planning: The Project Level

- The A&T Plan Indicates
  - Tools inherited from the strategy
  - Additional tools selected for that project
    For new or customized tools, the A&T plan must include
    - Costs (including training)
    - Implied activities
    - Potential risks
- The plan positions tools within the development process and the analysis and test methodology
  - Avoid waste of cost and effort from lack of contextualization of the tools
  - Example: tools for measuring code coverage
    - simple and inexpensive
    - (if not properly contextualized) an annoyance, producing data not put to productive use

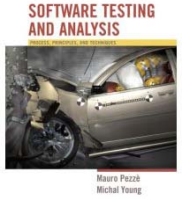


Process Support: Planning & Monitoring



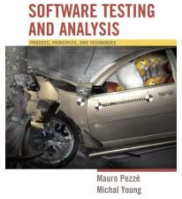
Automation in Process Management

- Managing a process involves …
  - planning a set of activities with appropriate cost and quality trade-offs
  - monitoring progress to identify risks as early as possible
  - avoiding delays by adjusting the plan as needed
- … and requires …
  - human creativity and insight for which no tool can substitute
- Tools can *support* process management and improve decision making by
  - organizing and monitoring activities and results
  - facilitating group interaction
  - managing quality documents
  - tracking costs



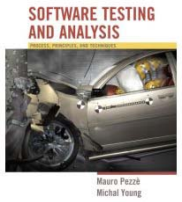
Classic Planning Tools

- Facilitate task scheduling, resource allocation, and cost estimation by arranging tasks according to resource and time constraints
- Can be specialized to A&T management with features for deriving relations among tasks, launching tasks, and monitoring completion of activities
- Examples: tools to
  - recognize delivery of a given artifact
  - schedule execution of a corresponding test suite
  - notify test designer of test results
  - record the actual execution time of the activity
  - signal schedule deviations to the quality manager
- Most useful when integrated in the analysis and test environment



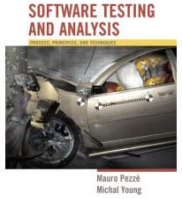
Version and Configuration Control Tools

- Analysis and testing involve complex relations among a large number of artifacts
- Version and configuration management tools
  - relate versions of software artifacts
  - trigger consistency checks and other activities
  - support analysis and testing activities like they control assembly and compilation of related modules
    - example: trigger execution of the appropriate test suites for each software modification
- Improve efficiency in well-organized processes
  - not a substitute for organization



Monitoring

- Integrated quality tracking
  - improves efficiency in a well-structured process,
  - does not by itself bring order out of chaos
- Progress must be monitored in terms of
  - schedule (actual effort and completion times vs plan)
  - level of quality
- Quality of the final product
  - cannot be directly measured before its completion
  - but we can derive useful indications
    - example: orthogonal defect classification [see chapter 20]

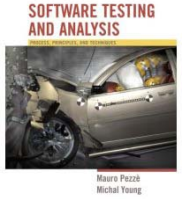


Quality Tacking

- Essential function: recognize deviations from expectation as early as possible to reduce consequences
- Proxy measures
  - must be computed early
  - must be interpreted in a way that avoids misleading conclusions or distorted incentives
- Example: lines of code
    - useful as a simple proxy for productivity
    - must be carefully interpreted to avoid creating both an incentive for verbosity and a disincentive for effective reuse
- Example: number of faults detected
    - useful to detect deviations from the norm
    - one should be as concerned about the causes of abnormally low numbers as high
- Collection, summary, and presentation of data can be automated
- Design and interpretation cannot be automated

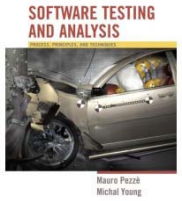
Managing People

- People may work
  - in different groups
  - in different companies
  - distributed across time zones and continents
- A large proportion of a software engineer's time is devoted to communication
- We need to
  - facilitate effective communication
  - limit disruptions and distractions of unmanaged communication

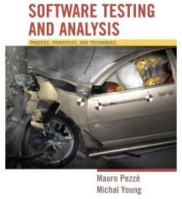


Managing Communication

- Simple general-purpose tools (e-mail, chat, forum, ...)
  - balance synchronous with asynchronous communication
  - examples
    - When excessive interruptions slow progress, we may replace synchronous with asynchronous communication
    - Conversely, when communication is splintered into many small exchanges punctuated by waits for reply, we may replace asynchronous with synchronous communication
- Communication is most effective when all parties have immediate access to relevant information
  - Task-specific tools can improve on general-purpose support
  - Example: tools for distributed software inspections
    - Extend chat interfaces or forum with
      - Managed presentation of the artifact to be inspected
      - Appropriate portions of checklists and automated analysis results

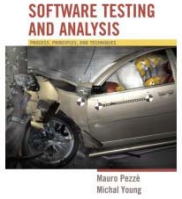


Measurement



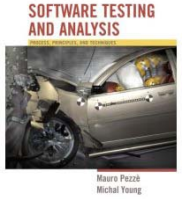
Metrics

- Measuring progress & results is necessary for managing processes
- … but often we cannot measure what we really care about
  - e.g., actual progress toward goals or effort remaining; projected reliability; …
- Metrics are *proxy* measures (rough guides) based on what we can measure
  - Anything that is *correlated* with the real measure of interest under typical conditions
  - Usually require calibration to local conditions



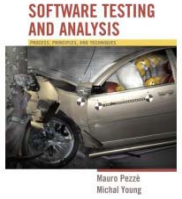
Static Metrics: Size

- Static metrics measure some software properties, often to estimate other properties (i.e., as proxies for things we can't measure)
- Size is the most basic property
  - strongly correlated with schedule and cost
  - several possible variations, depending on white space, comments, programming style
- Course measures include counts of modules or interfaces
  - functions, methods, formal parameters, etc
- Many more complex measures …
  - but lines of code is about as good (or bad) as complex measures for judging effort



Measuring Complexity

- Intuitive rationale: *If* we could measure how complicated a program or its parts were, we could …
  - Focus test & analysis on most error-prone parts of a system
  - Make better plans and schedules
  - Consider redesign of excessively complex subsystems
- But we can't measure true (logical) complexity directly.
- Control flow complexity is a proxy.



Cyclomatic complexity

- Among attempts to measure complexity, only cyclomatic complexity is still commonly collected

cyclomatic complexity V(g)

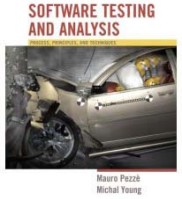
=

number of *independent paths* through the control flow graph

=

e - n + 2

(edges - nodes + 2)



Cyclomatic metrics and complexity

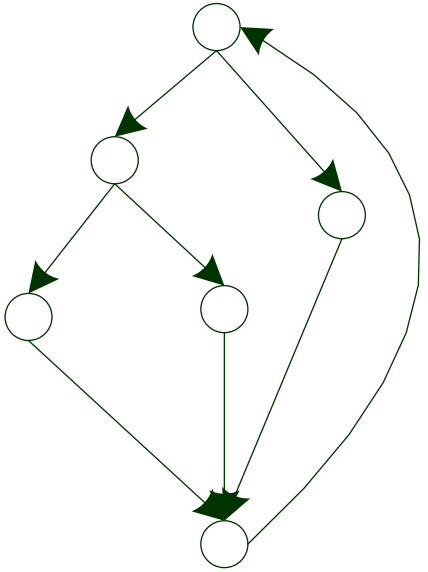
CFG1

V(g) = 1 - 2 + 2 = 1

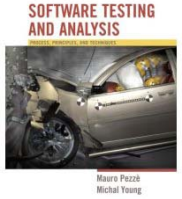
CFG2

V(g) = 5 - 6 + 2 = 1

CFG3

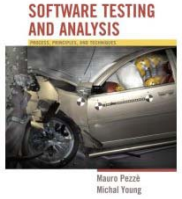


V(g) = 8 - 6 + 2 = 4



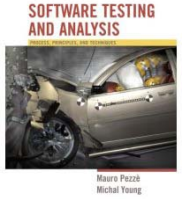
Interpreting Cyclomatic Complexity

- V(g) < 20
  - Low to moderate cyclomatic complexity
  - simple program
- V(g) > 20
  - high cyclomatic complexity
  - complex programs
- V(g) > 50
  - very high cyclomatic complexity
  - programs very difficult or impossible to thoroughly test
- Cyclomatic vs logical complexity
  - sign of complex control flow structure
  - does not capture other aspects of logical complexity that can lead to difficulty in testing



Metrics & Quality Standards

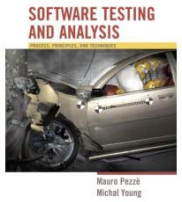
- Quality standards
  - May be prescribed (e.g., by contract)
  - May be adopted voluntarily as guidance
- A quality standard like ISO/IEC 9126 requires measurement of user-perceived quality
  - but doesn't say *how* to measure it
- To implement …
  We must find objective indicators (metrics) for each required quality



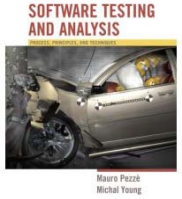
ISO/IEC 9126 Metrics (level 1)

Functionality	Ability to meet explicit and implicit functional requirements
Reliability	Ability to provide the required level of service when the software is used under appropriate conditions
Usability	Ease of understanding, teaching, and using
Efficiency	Ability to guarantee required performance under given conditions
Maintainability	Ability to be updated, corrected, and modified
Portability	Ability to be executed in different environments and interoperate with other software

Broad qualities require refinement and mapping to objectively measurable properties

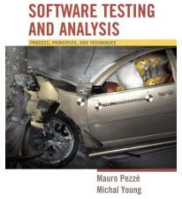


Automating Program Analysis, Test Case Generation, and Test Execution



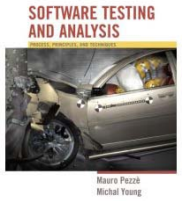
Test Case Generation and Execution

- Automation is important because
  - It is large fraction of overall test and analysis costs
  - can become a scheduling bottleneck near product delivery deadlines
- Designing a test suite
  - involves human creativity
- Instantiating and executing test cases
  - is a repetitive and tedious task
  - can be largely automated to
    - reduce costs
    - accelerate the test cycle



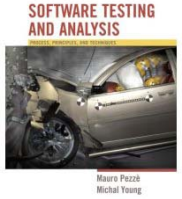
Automated Testing - Stages

- Push the creative work as far forward as possible
  - E.g., designing functional test suites is part of the specification process
  - At each level, from systems requirements through architectural interfaces and detailed module interfaces
- Construct scaffolding with the product
- Automate instantiation and execution
  - So they are not a bottleneck
  - So they can be repeated many times



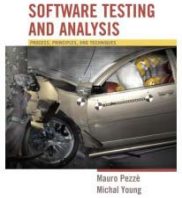
Static Analysis and Proof

- Effective for
  - Quick and cheap checks of simple properties
    - Example: simple data flow analyses can identify anomalous patterns
  - Expensive checks necessary for critical properties
    - Example: finite state verification tool to find synchronization faults



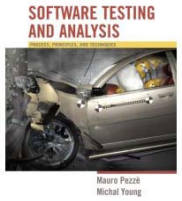
Design for Verification

- Decompose Verification Problems
  - Design: enforce design rules to accommodate analysis
    - example: encapsulate safety-critical properties into a safety kernel
  - Verification: focus on encapsulated or simplified property
    - example:
    - prove safety properties of the (small) kernel
    - check (cheaply, automatically) that all safety-related actions are mediated by the kernel



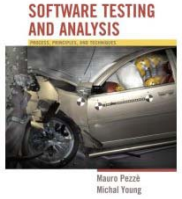
Undecidability and Automated Analysis

- Some tools report false alarms in addition to real violations of the properties they check
  - example: data flow analyzers
- Some tools avoid false alarms but may also fail to detect all violations
  - example: bug finders
- Some tools are heavyweight with respect to requirement for skilled human interaction and guidance to provide strong assurance of important general properties
  - examples
  - Finite state verification systems (model checkers)
    - can verify conformance between a model of a system and a specified property
    - require construction of the model and careful statement of the property
  - Theorem provers
    - execute with interactive guidance
    - requires specialists with a strong mathematical background to formulate the problem and the property interactively select proof strategies



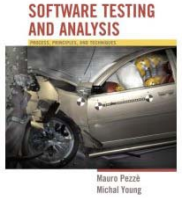
Complex analysis tools

- Verifiers based on theorem proving
  - verify a wide class of properties
  - require extensive human interaction and guidance
- Finite state verification tools
  - restricted focus
  - execute completely automatically
  - almost always require several rounds of revision to properly formalize a model and property to be checked



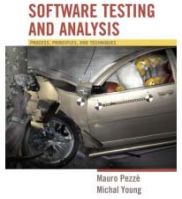
Simple analysis tools

- Restricted to checking a fixed set of simple properties
  - do not require any additional effort for specification
- Type checkers
  - typically applied to properties that are syntactic = enforce a simple well-formedness rule
  - violations are easy to diagnose and repair
  - Often rules are stricter than one would like
- Data flow analyzers
  - sensitive to program control and data flow
  - often used to identify anomalies rather than simple, unambiguous faults
- Checkers of domain specific properties
  - Web site link checkers
  - ...



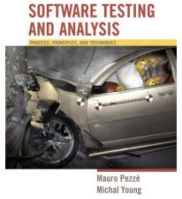
Cognitive Aids

Supporting creative, human processes



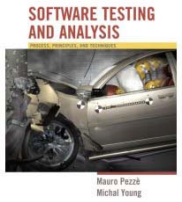
Cognitive Aids: Problems to Address

- Nonlocality
  - Information that requires a shift of attention
  - Example: following a reference in one file or page to a definition on another
  - creates opportunities for human error
- Information clutter
  - Information obscured by a mass of distracting irrelevant detail

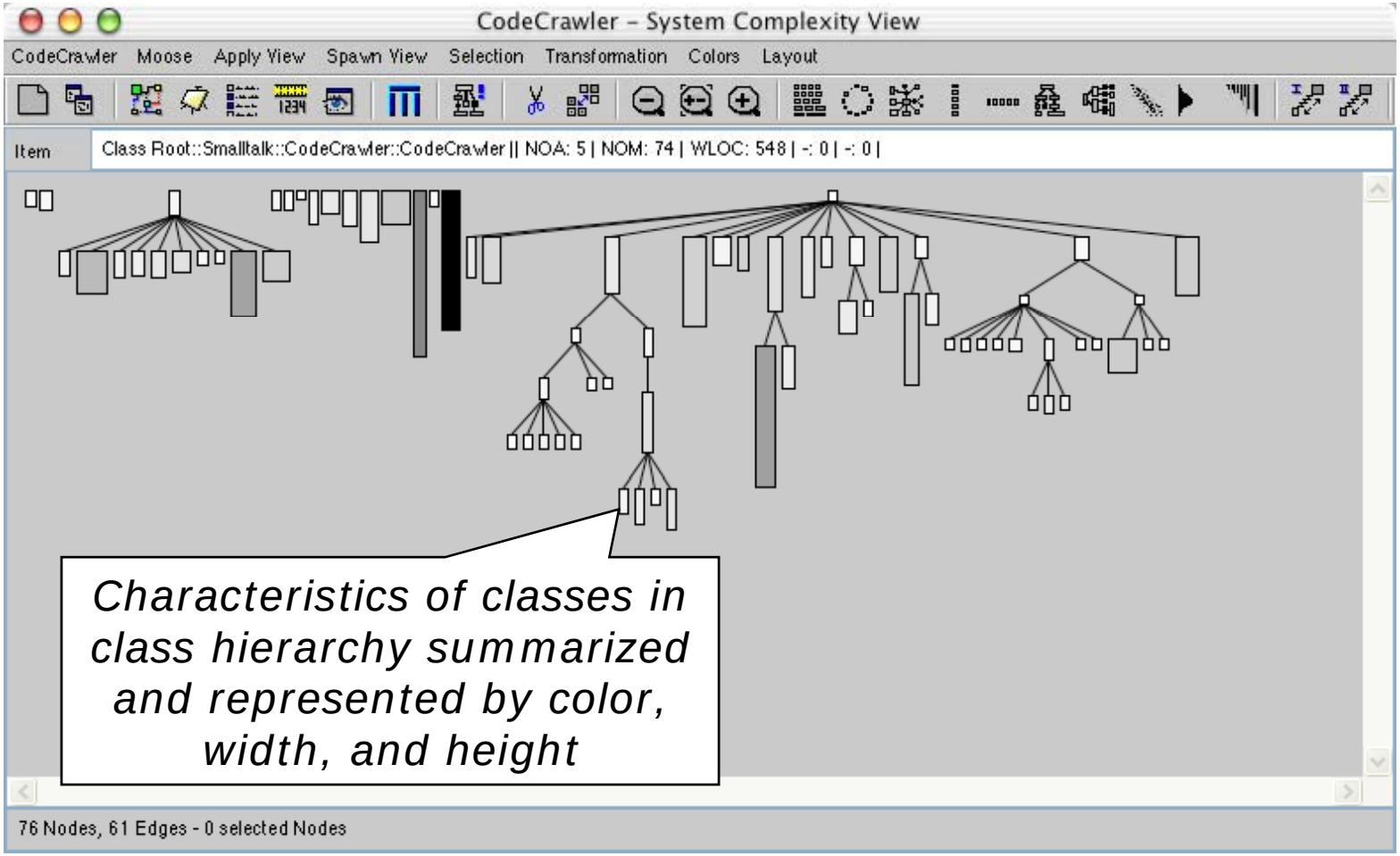


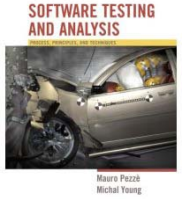
Cognitive Aids: Approaches

- Nonlocality and clutter
  - increase the cognitive burden of inspecting complex artifacts (requirements statements, program code, test logs,…)
  - decrease effectiveness and efficiency
- Can be reduced by automatically focusing and abstracting from irrelevant detail
- Browsing and visualization aids
  - Often embedded in other tools and customized to support particular tasks
    - Pretty-printing and program slicing
    - Diagrammatic representations

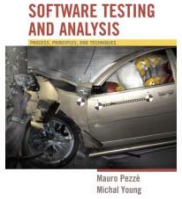


Diagrammatic Representations Example: Code Crawler



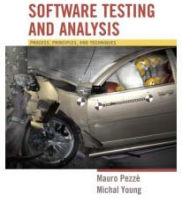


Related Tools: Version control, Debugging



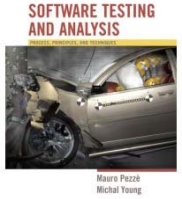
Version Control

- Record versions and releases of each part of an evolving software system
  - From very simple version management (CVS, SVN) to very complex configuration management systems
- Useful for maintaining test artifacts (plans, test cases, logs, etc.)
  - Test artifacts are versioned with the product
- Integrate with process support
  - E.g., it is possible to trigger re-testing on changes, or require successful test before committing to baseline
- Provide historical information for tracing faults across versions and collecting data for improving the process



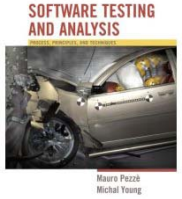
Debugging ≠ Testing

- Testing = detecting the presence of software faults
- Debugging = locating, diagnosing, and repairing faults
- Responsibility for testing and debugging typically fall to different individuals
- Debugging starts with a set of test cases
  - A small, simple test case that invariably fails is far more valuable in debugging than a complex scenario, particularly one that may fail or succeed depending on unspecified conditions
  - larger suites of single-purpose test cases are better than a small number of comprehensive test cases



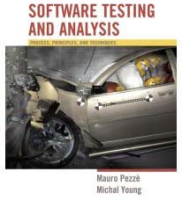
Run-time Debugging Tools

- All modern deguggers …
  - Allow inspection of program state
  - Pause execution
    - at selected points (breakpoints)
    - when certain conditions occur (watchpoints)
    - after a fixed number of execution steps
  - Provide display and control at the level of program source code
- Specialized debugging support may include
  - Visualization (e.g., for performance debugging)
  - Animation of data structures
  - Differential debugging compares a set of failing executions to other executions that do not fail



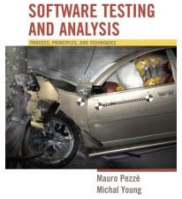
Automation Strategy

(summary)



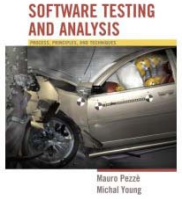
Choosing and Integrating tools

- Tools and approaches must fit …
  - development organization, process, and application domain
- Simple rule: Identify significant costs (money or schedule) for automation
  - Example: automated module testing
    - of little use for organizations using the Cleanroom process
    - essential for organizations using XP
  - Example:
    - organizations building safety-critical software can justify investment in sophisticated tools for verifying the properties of specifications and design organization that builds rapidly evolving mass market applications is more likely to benefit from good support for automated regression testing
- Also consider activities that *require* automation
  - Missed by analysis of current testing & analysis costs



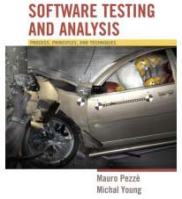
Think Strategically

- Evaluate investments in automation beyond a single project and beyond the quality team
- Reusing common tools across projects reduces
  - cost of acquiring and installing tools
  - cost of learning to use them effectively
  - impact on project schedule
- Think globally
  - Often quality tools have costs and benefits for other parts of the software organization



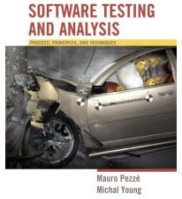
Summary

- Automation
  - Can improve the efficiency of some quality activities
  - Is a necessity for implementing others
  - Os never a substitute for a rational, well-organized quality process
  - Can incrementally improve processes that makes the best use of human resources
  - Must be carefully evaluated to balance costs and benefits



Optional Slides

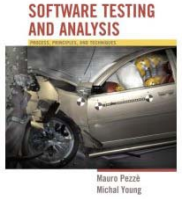
The following slides are not crucial to understanding the core ideas of the chapter, but may be used by instructors who wish to cover some details in more detail.



Static Metrics: Size

- Static metrics measure some software properties, often to estimate other properties (i.e., as proxies for things we can't measure)
- Size
  - most basic property of software
  - strongly correlated with schedule and cost
  - several possible variations, depending on white space, comments, programming style

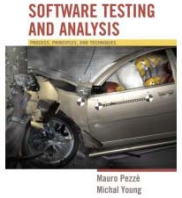
Size	Size of the source file, measured in bytes
Lines	All-inclusive count of lines in source code file
LOC	Lines of code, excluding comment and blank lines
eLOC	Effective lines of code, excluding comments, blank lines, and stand-alone braces or parenthesis
lLOC	Logical lines of code, that is, statements as identified by logical separators such as semicolons



Static Metrics: Complexity and Readability

- Complexity may be as important as sheer size
- Example proxy measures for complexity and readability:

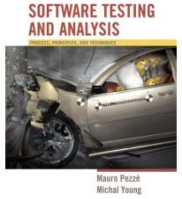
CDENS	Comment density: comment lines/eLOC
Blocks	Number of basic blocks: sequences of statements with one entry point, one exit point, and no internal branches
AveBlockL	Average number of lines per basic block
NEST	Control structure nesting level: minimum, maximum, and average
Loops	Number of loops
Int1	Number of first order intervals
MOI	Maximum order of intervals
LCSAJ	Number of linear code sequences [see chapter 5]
BRANCH	Number of branches in the control flow graph



Coarse Measurements of Size

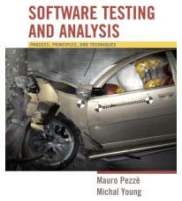
- Consider interfaces between units
- Examples

Functions	Number of defined functions (or methods, procedures, ...)
FPar	Number of formal parameters of functions
FRet	Number of return points of functions
IComplex	Interface complexity: FPar + FRet



Complexity of Object-Oriented Code

WMC	Weighted methods per class sum of the complexities of methods in all classes, divided by the number of classes parametric with respect to a measure of complexity in methods
DIT	Depth of the inheritance tree of a class
NOC	Number of children (subclasses) of a class
RFC	Response for a class number of methods that may be executed in response to a method call to an object of the class (transitive closure of the calling relation)
CBO	Coupling between object classes number of classes with which the class is coupled
LCOM	Lack of cohesion in methods number of methods with pairwise disjoint sets of instance variables referenced within their respective method bodies



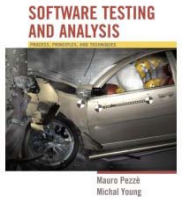
ISO/IEC 9126 Metrics - level 2 (1/3)

Functionality

Suitability	Ability to provide the required functionality
Accuracy	Ability to provide correct results
Interoperability	Ability to interact with other products
Security	Ability to protect access to private data and guarantee a level of service

Reliability

Maturity	Ability to avoid failures
Fault Tolerance	Ability to maintain a suitable level of functionality even in the presence of external failures
Recoverability	Ability to recover data and resume function after a failure



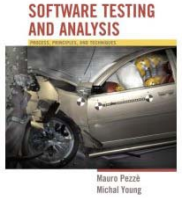
ISO/IEC 9126 Metrics - level 2 (2/3)

Usability

Understandability	Ease of understanding the product
Learnability	Ease of training users
Operability	Ease of working with the product
Attractiveness	Degree of appreciation by users

Efficiency

Time Behavior	Ability to satisfy average and maximum response time requirements
Resource Utilization	Amount of resources needed for executing the software



ISO/IEC 9126 Metrics - level 2 (3/3)

Maintainability

Analyzability	Ease of analyzing the software to reveal faults
Changeability	Ease of changing the software
Interoperability	Ability to interact with other products
Stability	Ability to minimize the effects of changes
Testability	Ease of testing the software

Portability

Adaptability	Ability to be adapted to new environments
Installability	Ease of installing the software
Co-existence	Ability to share resources with other products
Replaceability	Ability to be replaced by other products