

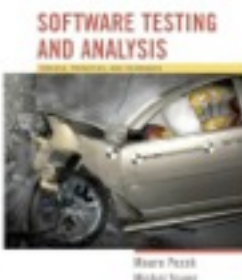
Test case selection and adequacy criteria

Automated testing and
verification

J.P. Galeotti - Alessandra Gorla

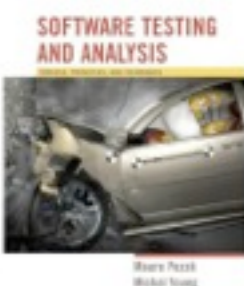
Adequacy: We can't get what we want

- What we would like:
 - A real way of measuring effective testing
If the system passes an adequate suite of test cases, then it must be correct (or dependable)
- But that's impossible!
 - Adequacy of test suites, in the sense above, is provably undecidable.
- So we'll have to settle on weaker proxies for adequacy
 - Design rules to highlight inadequacy of test suites



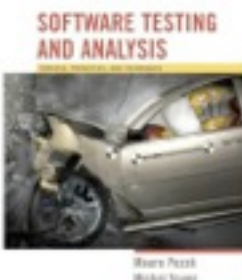
Adequacy Criteria as Design Rules

- Many design disciplines employ *design rules*
 - E.g.: “traces (on a chip, on a circuit board) must be at least ____ wide and separated by at least ____”
 - “The roof must have a pitch of at least ____ to shed snow”
 - “Interstate highways must not have a grade greater than 6% without special review and approval”
- Design rules do not guarantee good designs
 - Good design depends on talented, creative, disciplined designers; design rules help them avoid or spot flaws
 - Test design is no different



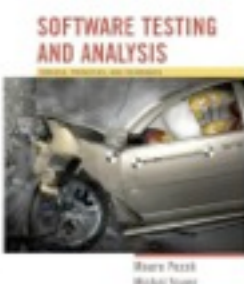
Practical (in)Adequacy Criteria

- Criteria that identify inadequacies in test suites.
 - Examples
 - *if the specification describes different treatment in two cases, but the test suite does not check that the two cases are in fact treated differently, we may conclude that the test suite is inadequate to guard against faults in the program logic.*
 - *If no test in the test suite executes a particular program statement, the test suite is inadequate to guard against faults in that statement.*
- If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite.
- If a test suite satisfies all the obligations by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness.



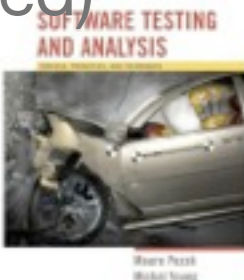
Analogy: Building Codes

- Building codes are sets of design rules
 - Maximum span between beams in ceiling, floor, and walls; acceptable materials; wiring insulation; ...
 - Minimum standards, subject to judgment of building inspector who interprets the code
- You wouldn't buy a house just because it's "up to code"
 - It could be ugly, badly designed, inadequate for your needs
- But you might avoid a house because it isn't
 - Building codes are adequacy criteria, like practical test "adequacy" criteria



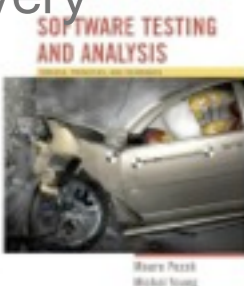
Some useful terminology

- **Test case:** a set of inputs, execution conditions, and a pass/fail criterion.
- **Test case specification:** a requirement to be satisfied by one or more test cases.
- **Test obligation:** a partial test case specification, requiring some property deemed important to thorough testing.
- **Test suite:** a set of test cases.
- **Test or test execution:** the activity of executing test cases and evaluating their results.
- **Adequacy criterion:** a predicate that is true (satisfied) or false (not satisfied) of a \langle program, test suite \rangle pair.



Where do test obligations come from?

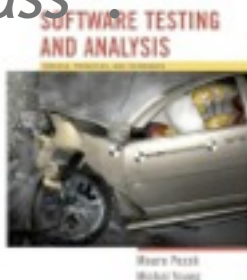
- Functional (black box, specification-based): from software specifications
 - Example: If spec requires robust recovery from power failure, test obligations should include simulated power failure
- Structural (white or glass box): from code
 - Example: Traverse each program loop one or more times.
- Model-based: from model of system
 - Models used in specification or design, or derived from code
 - Example: Exercise all transitions in communication protocol model
- Fault-based: from hypothesized faults (common bugs)
 - Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs



Adequacy criteria

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if
 - all the tests succeed (pass)
 - every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
- Example:

the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S , and the outcome of each test execution was “pass”.



Satisfiability

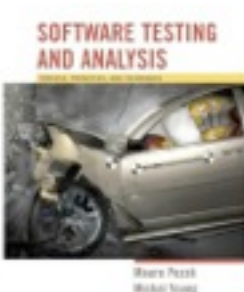
- Sometimes *no* test suite can satisfy a criterion for a given program
 - Example: Defensive programming style includes “can’t happen” sanity checks

```
if (z < 0) {  
    throw new LogicError(  
        "z must be positive here!")  
}
```

No test suite can satisfy statement coverage for this program (if it's correct)

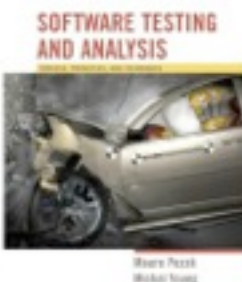
Coping with Unsatisfiability

- Approach A: exclude any unsatisfiable obligation from the criterion.
 - Example: modify statement coverage to require execution only of statements that can be executed.
 - But we can't know for sure which are executable!
- Approach B: measure the extent to which a test suite approaches an adequacy criterion.
 - Example: if a test suite satisfies 85 of 100 obligations, we have reached 85% *coverage*.
 - Terms: An adequacy criterion is satisfied or not, a coverage measure is the fraction of satisfied obligations



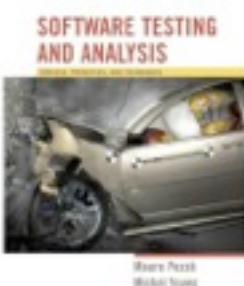
Coverage: Useful or Harmful?

- Measuring coverage (% of satisfied test obligations) can be a useful indicator ...
 - Of progress toward a thorough test suite, of trouble spots requiring more attention
- ... or a dangerous seduction
 - Coverage is only a proxy for thoroughness or adequacy
 - It's easy to improve coverage without improving a test suite (much easier than designing good test cases)
 - The only measure that really matters is (cost-)effectiveness



Comparing Criteria

- Can we distinguish stronger from weaker adequacy criteria?
- Empirical approach: Study the effectiveness of different approaches to testing in industrial practice
 - What we really care about, but ...
 - Depends on the setting; may not generalize from one organization or project to another
- Analytical approach: Describe conditions under which one adequacy criterion is provably stronger than another
 - Stronger = gives stronger guarantees
 - One piece of the overall “effectiveness” question



The subsumes relation

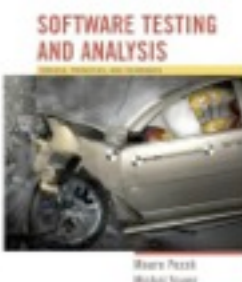
Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.

- Example:

Exercising all program branches (branch coverage) *subsumes* exercising all program statements

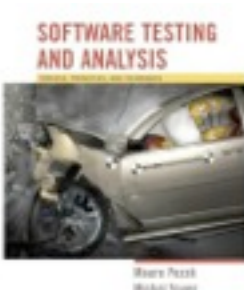
- A common analytical comparison of closely related criteria

- Useful for working from easier to harder levels of coverage, but not a direct indication of quality



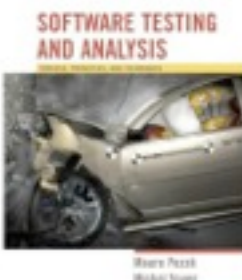
Uses of Adequacy Criteria

- Test selection approaches
 - Guidance in devising a thorough test suite
 - Example: A specification-based criterion may suggest test cases covering representative combinations of values
- Revealing missing tests
 - Post hoc analysis: What might I have missed with this test suite?
- Often in combination
 - Example: Design test suite from specifications, then use structural criterion (e.g., coverage of all branches) to highlight missed logic



Summary

- Adequacy criteria provide a way to define a notion of “thoroughness” in a test suite
 - But they don’t offer guarantees; more like *design rules to highlight inadequacy*
- Defined in terms of “covering” some information
 - Derived from many sources: Specs, code, models, ...
- May be used for selection as well as measurement
 - With caution! An aid to thoughtful test design, not a substitute



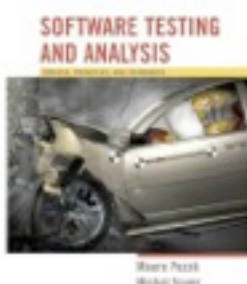
Functional Testing

Automated testing and
verification

J.P. Galeotti - Alessandra Gorla

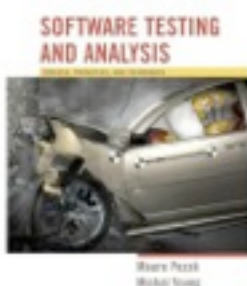
Functional testing

- **Functional testing:** Deriving test cases from program specifications
 - *Functional* refers to the source of information used in test case design, not to what is tested
- *Also known as:*
 - **specification-based testing** (from specifications)
 - **black-box testing** (no view of the code)
- Functional specification = description of intended program behavior
 - either formal or informal



Systematic vs Random Testing

- **Random** (uniform):
 - Pick possible inputs uniformly
 - Avoids designer bias
 - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
 - But treats all inputs as equally valuable
- **Systematic** (non-uniform):
 - Try to select inputs that are especially valuable
 - Usually by choosing representatives of classes that are supposed to fail *often* or *not at all*
- Functional testing is systematic testing



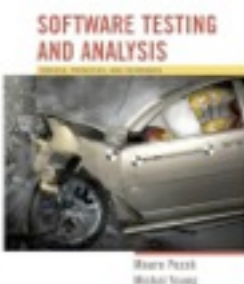
Why Not Random?

- Non-uniform distribution of faults
- *Example:* Java class “roots” applies quadratic equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

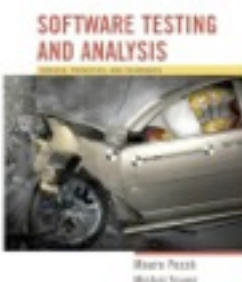
Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$

Failing values are *sparse* in the input space — needles in a very big haystack.
Random sampling is unlikely to choose $a=0.0$ and $b=0.0$



Consider the purpose of testing ...

- To estimate the proportion of needles to hay, sample randomly
 - Reliability estimation requires unbiased samples for valid statistics. *But that's not our goal!*
- To find needles and remove them from hay, look systematically (non-uniformly) for needles
 - Unless there are a *lot* of needles in the haystack, a random sample will not be effective at finding them
 - We need to use everything we know about needles, e.g., are they heavier than hay? Do they sift to the bottom?



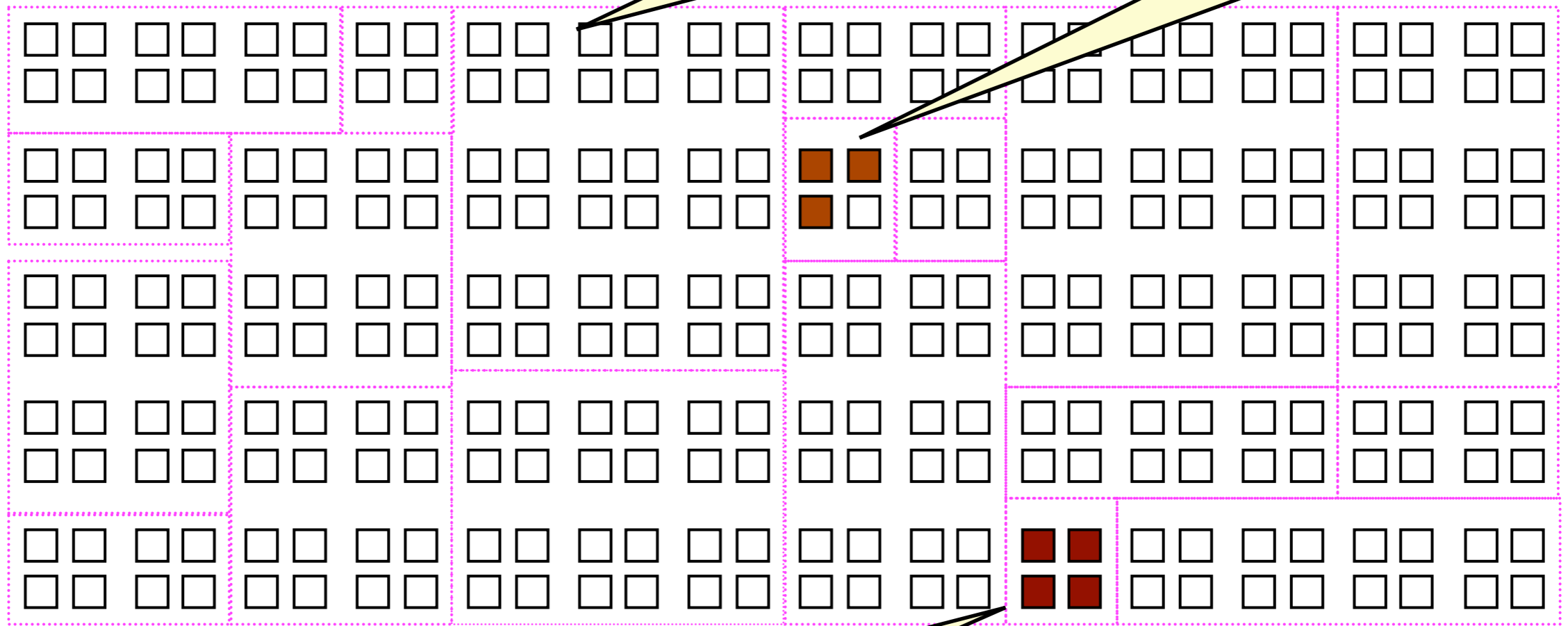
Systematic Partition Testing

- Failure (valuable test case)
- No failure

Failures are sparse in the space of possible inputs ...

... but dense in some parts of the space

The space of possible input values
(the haystack)



If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of drawing pink lines to isolate regions with likely failures

SOFTWARE TESTING AND ANALYSIS

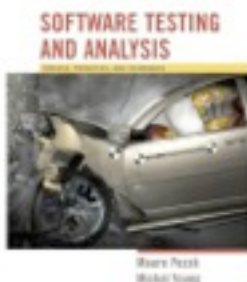


(c) 2007 Mauro Pezzè & Michal Young

Mauro Pezzè
Michal Young

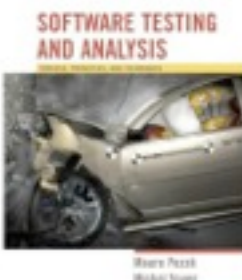
The partition principle

- Exploit some knowledge to choose samples that are more likely to include “special” or trouble-prone regions of the input space
 - Failures are sparse in the whole input space ...
 - ... but we may find regions in which they are dense
- (Quasi*-)Partition testing: separates the input space into classes whose union is the entire space
 - *Quasi because: The classes may overlap
- Desirable case: Each fault leads to failures that are dense (easy to find) in some class of inputs
 - sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault
 - seldom guaranteed; we depend on experience-based heuristics



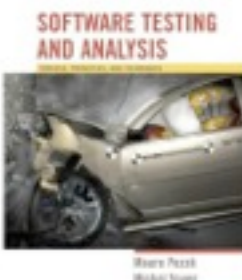
Functional testing: exploiting the specification

- Functional testing uses the specification (formal or informal) to partition the input space
 - E.g., specification of “roots” program suggests division between cases with zero, one, and two real roots
- Test each category, and boundaries between categories
 - No guarantees, but experience suggests failures often lie at the boundaries (as in the “roots” program)



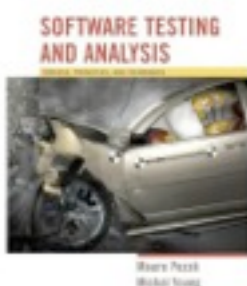
Why functional testing?

- The base-line technique for designing test cases
 - **Timely**
 - Often useful in refining specifications and assessing testability *before* code is written
 - **Effective**
 - finds some classes of fault (e.g., missing logic) that can elude other approaches
 - **Widely applicable**
 - to any description of program behavior serving as spec
 - at any level of granularity from module to system testing.
 - **Economical**
 - typically less expensive to design and execute than structural (code-based) test cases



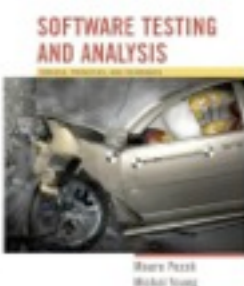
Early functional test design

- Program code is not necessary
 - Only a description of intended behavior is needed
 - Even incomplete and informal specifications can be used
 - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
 - Often reveals ambiguities and inconsistency in spec
 - Useful for assessing testability
 - And improving test schedule and budget by improving spec
 - Useful explanation of specification
 - or in the extreme case (as in XP), test cases are the spec



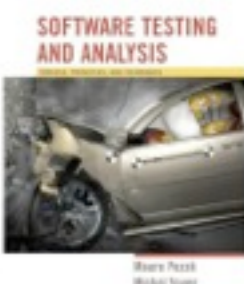
Functional versus Structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults
- Functional testing is best for *missing logic* faults
 - A common problem: Some program logic was simply forgotten
 - Structural (code-based) testing will never focus on code that isn't there!



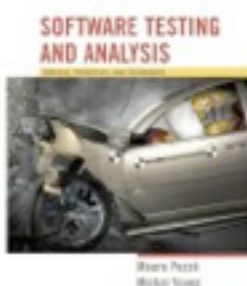
Functional vs structural test: granularity levels

- Functional test applies at all granularity levels:
 - Unit (from module interface spec)
 - Integration (from API or subsystem spec)
 - System (from system requirements spec)
 - Regression (from system requirements + bug history)
- Structural (code-based) test design applies to relatively small parts of a system:
 - Unit
 - Integration

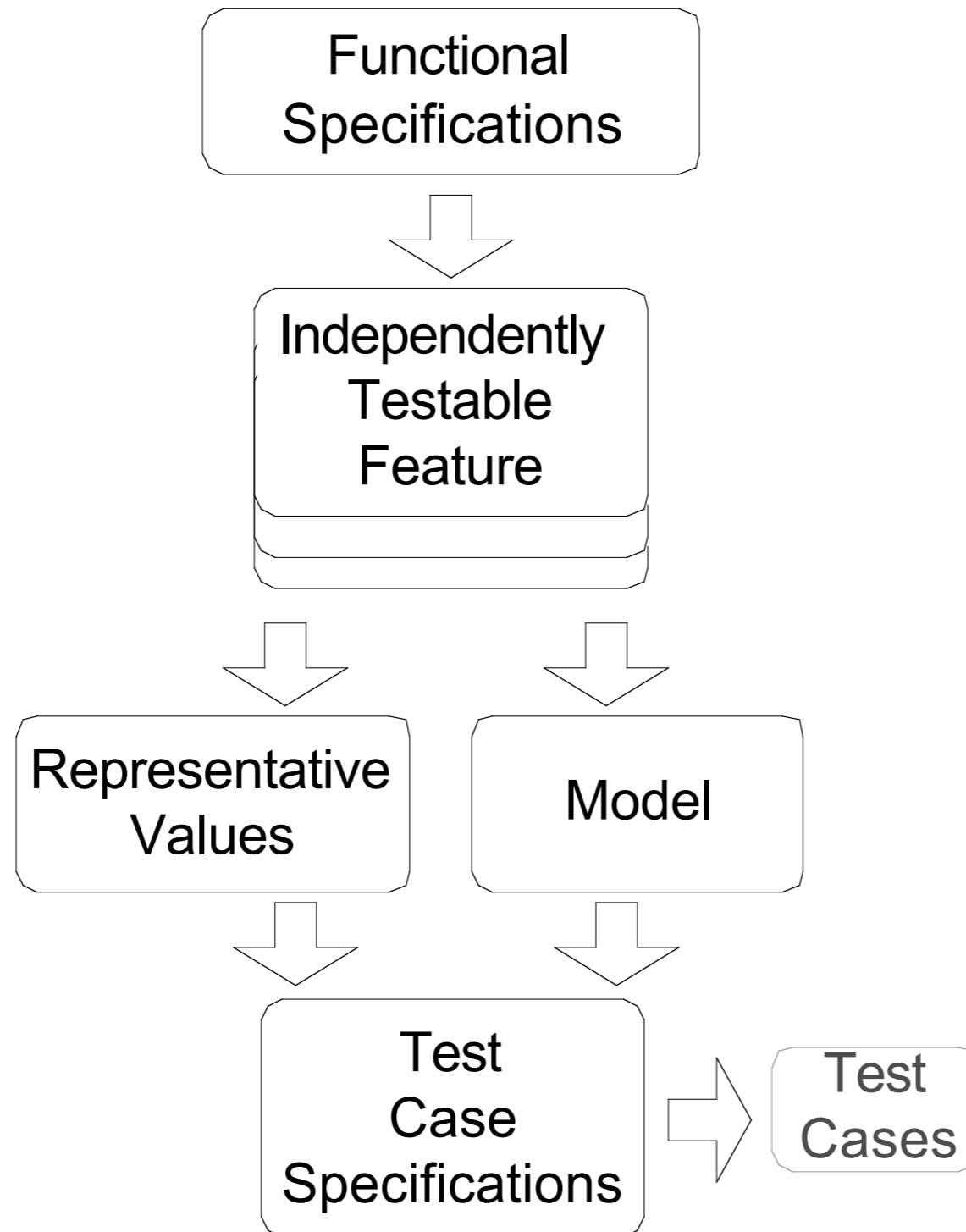


Steps: From specification to test cases

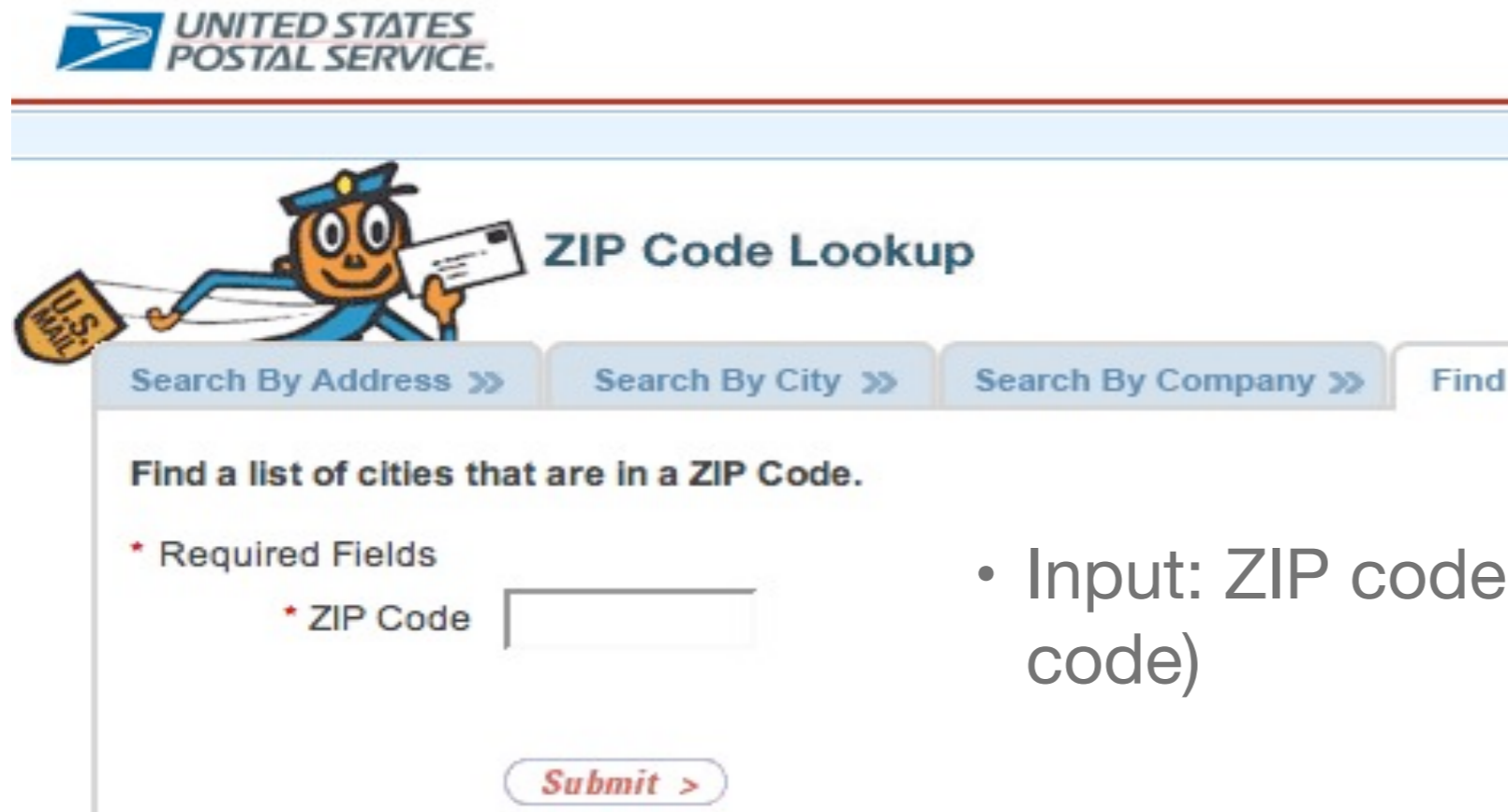
- 1. Decompose the specification
 - If the specification is large, break it into *independently testable features* to be considered in testing
- 2. Select representatives
 - Representative values of each input, or
 - Representative behaviors of a *model*
 - Often simple input/output transformations don't describe a system. We use models in program specification, in program design, and in test design
- 3. Form test specifications
 - Typically: combinations of input values, or model behaviors
- 4. Produce and execute actual tests



From specification to test cases



Simple example: Postal code lookup



The screenshot shows the United States Postal Service logo at the top left. Below it is a blue horizontal bar. Underneath the bar is a cartoon mail carrier holding a letter, with the text "ZIP Code Lookup" to its right. Below the mail carrier are three search options: "Search By Address >>", "Search By City >>", and "Search By Company >>". To the right of these options is a "Find" button. Below the search options is a section titled "Find a list of cities that are in a ZIP Code." Under this title, there is a "Required Fields" section with a red asterisk. Below this, there is a red asterisk followed by "ZIP Code" and an empty text input field. Below the input field is a "Submit >" button.

- Input: ZIP code (5-digit US Postal code)
- Output: List of cities
- What are some representative values (or classes of value) to test?

Example: Representative values

Simple example with one input, one output

- Correct zip code
 - With 0, 1, or many cities
- Malformed zip code
 - Empty; 1-4 characters; 6 characters; very long
 - Non-digit characters
 - Non-character data



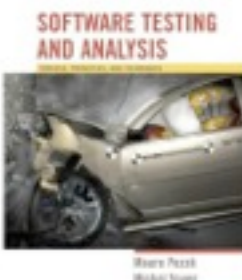
The screenshot shows the United States Postal Service logo at the top. Below it is a cartoon mail carrier holding a letter. The main heading is "ZIP Code Lookup". There are three search options: "Search By Address >>", "Search By City >>", and "Search By Company >>". A "Find" button is on the right. Below the search options, it says "Find a list of cities that are in a ZIP Code." Underneath, there is a "Required Fields" section with a "ZIP Code" label and an empty input field. A "Submit >" button is at the bottom.

Note prevalence of boundary values (0 cities, 6 characters) and error cases



Summary

- Functional testing, i.e., generation of test cases from specifications is a valuable and flexible approach to software testing
 - Applicable from very early system specs right through module specifications
- (quasi-)Partition testing suggests dividing the input space into (quasi-)equivalent classes
 - Systematic testing is intentionally non-uniform to address special cases, error conditions, and other small places
 - Dividing a big haystack into small, hopefully uniform piles where the needles might be concentrated



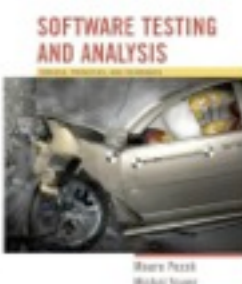
Combinatorial Testing

Automated testing and
verification

J.P. Galeotti - Alessandra Gorla

Combinatorial testing: Basic idea

- Identify distinct attributes that can be varied
 - In the data, environment, or configuration
 - Example: browser could be “IE” or “Firefox”, operating system could be “Vista”, “XP”, or “OSX”
- Systematically generate combinations to be tested
 - Example: IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX, ...
- Rationale: Test cases should be varied and include possible “corner cases”



Key ideas in combinatorial approaches

- **Category-partition testing**

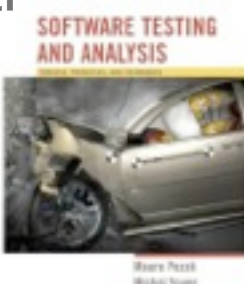
- separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases

- **Pairwise testing**

- systematically test interactions among attributes of the program input space with a relatively small number of test cases

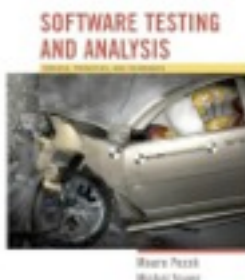
- **Catalog-based testing**

- aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values



Category partition (manual steps)

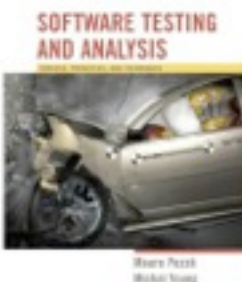
1. Decompose the specification into independently testable features
 - for each feature identify
 - parameters
 - environment elements
 - for each parameter and environment element identify elementary characteristics (categories)
2. Identify relevant values
 - for each characteristic (category) identify (classes of) values
 - normal values
 - boundary values
 - special values
 - error values
3. Introduce constraints



An informal specification: check configuration

Check Configuration

- Check the validity of a computer configuration
- The parameters of check-configuration are:
 - Model
 - Set of components



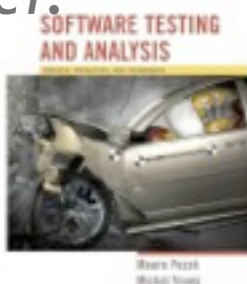
An informal specification: parameter model

Model

- A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs

Example:

The required “slots” of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.



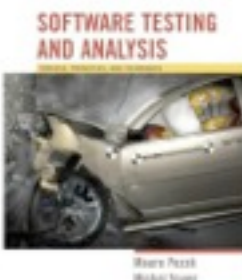
An informal specification of parameter **set of components**

Set of Components

- A set of *(slot, component)* pairs, corresponding to the required and optional slots of the model. A *component* is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value *empty* is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

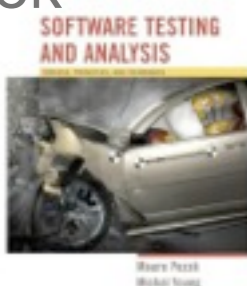
Example:

The default configuration of the Chipmunk C20 includes 100 gigabytes of hard disk; 200 and 500 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 200 gigabytes of hard disk.



Step 1: Identify independently testable units and categories

- Choosing categories
 - no hard-and-fast rules for choosing categories
 - not a trivial task!
- Categories reflect test designer's judgment
 - regarding which classes of values may be treated differently by an implementation
- Choosing categories well requires experience and knowledge
 - of the application domain and product architecture. The test designer must look under the surface of the specification and identify hidden characteristics



Step 1: Identify parameters and environment

Parameter *Model*

- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

Parameter *Components*

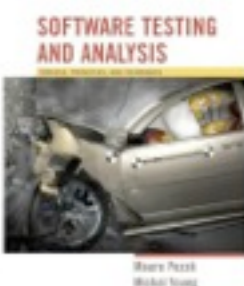
- Correspondence of selection with model slots
- Number of required components with selection \neq empty
- Required component selection
- Number of optional components with selection \neq empty
- Optional component selection

Environment element: *Product database*

- Number of models in database (#DBM)
- Number of components in database (#DBC)

Step 2: Identify relevant values

- Identify (list) representative classes of values for each of the categories
 - Ignore interactions among values for different categories (considered in the next step)
- Representative values may be identified by applying
 - Boundary value testing
 - select extreme values within a class
 - select values outside but as close as possible to the class
 - select interior (non-extreme) values of the class
 - Erroneous condition testing
 - select values outside the normal domain of the program



Step 2: Identify relevant values: Model

Model number

Malformed

Not in database

Valid

Number of required slots for selected model (#SMRS)

0

1

Many

Number of optional slots for selected model (#SMOS)

0

1

Many

Step 2: Identify relevant values: Component

Correspondence of selection with model slots

Omitted slots

Extra slots

Mismatched slots

Complete correspondence

Number of required components with non empty selection

0

< #SMRS

= #SMRS

Required component selection

Some defaults

All valid

≥ 1 incompatible with slots

≥ 1 incompatible with another selection

≥ 1 incompatible with model

≥ 1 not in database

Number of optional components with non empty selection

0

< #SMOS

= #SMOS

Optional component selection

Some defaults

All valid

≥ 1 incompatible with slots

≥ 1 incompatible with another selection

≥ 1 incompatible with model

≥ 1 not in database

SOFTWARE TESTING
AND ANALYSIS



(c) 2007 Mauro Pezzè & Michal Young

Step 2: Identify relevant values: Database

Number of models in database (#DBM)

0

1

Many

Number of components in database (#DBC)

0

1

Many

Note 0 and 1 are unusual (special) values. They might cause unanticipated behavior alone or in combination with particular values of other parameters.



Step 3: Introduce constraints

- A combination of values for each category corresponds to a test case specification
 - in the example we have 314.928 test cases
 - most of which are impossible!
 - example
zero slots and *at least one incompatible slot*
- Introduce constraints to
 - rule out impossible combinations
 - reduce the size of the test suite if too large



Step 3: error constraint

[error] indicates a value class that

- corresponds to a erroneous values
- need be tried only once

Example

Model number: Malformed and Not in database

error value classes

- No need to test all possible combinations of errors
- One test is enough (we assume that handling an error case bypasses other program logic)



Example - Step 3: error constraint

Model number

Malformed [error]

Not in database [error]

Valid

Correspondence of selection with model slots

Omitted slots [error]

Extra slots [error]

Mismatched slots [error]

Complete correspondence

Number of required comp. with non empty selection

0 [error]

< number of required slots [error]

Required comp. selection

≥ 1 not in database [error]

Number of models in database (#DBM)

0 [error]

Number of components in database (#DBC)

0 [error]

**Error constraints
reduce test suite
from 314.928 to
2.711 test cases**



Step 3: property constraints

constraint **[property]** **[if-property]** rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

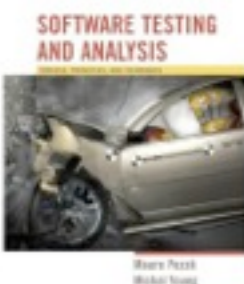
Example

combine

Number of required comp. with non empty selection = number required slots *[if RSMANY]*

only with

Number of required slots for selected model (#SMRS) = Many *[RSMANY]*



Example - Step 3: property constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

< number required slots	[if OSNE]
= number required slots	[if OSMANY]

from 2.711 to 908
test cases



Step 3 (cont): **single** constraints

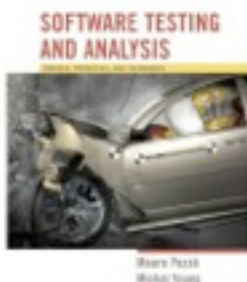
[**single**] indicates a value class that test designers choose to test only once to reduce the number of test cases

Example

*value some default for **required component selection** and **optional component selection** may be tested only once despite not being an erroneous condition*

note -

single and error have the same effect but differ in rationale. Keeping them distinct is important for documentation and regression testing



Example - Step 3: **single** constraints

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]
--------------	----------

Optional component selection

Some default	[single]
--------------	----------

Number of models in database (#DBM)

1	[single]
---	----------

Number of components in database (#DBC)

1	[single]
---	----------

**from 908 to 69
test cases**

Check configuration – Summary

Parameter Model

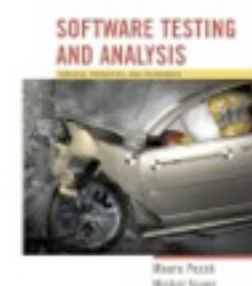
- Model number
 - Malformed [error]
 - Not in database [error]
 - Valid
- Number of required slots for selected model (#SMRS)
 - 0 [single]
 - 1 [property RSNE] [single]
 - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
 - 0 [single]
 - 1 [property OSNE] [single]
 - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
 - 0 [error]
 - 1 [single]
 - Many
- Number of components in database (#DBC)
 - 0 [error]
 - 1 [single]
 - Many

Parameter Component

- Correspondence of selection with model slots
 - Omitted slots [error]
 - Extra slots [error]
 - Mismatched slots [error]
 - Complete correspondence
- # of required components (selection \neq empty)
 - 0 [if RSNE] [error]
 - < number required slots [if RSNE] [error]
 - = number required slots [if RSMANY]
- Required component selection
 - Some defaults [single]
 - All valid
 - ≥ 1 incompatible with slots
 - ≥ 1 incompatible with another selection
 - ≥ 1 incompatible with model
 - ≥ 1 not in database [error]
- # of optional components (selection \neq empty)
 - 0
 - < #SMOS [if OSNE]
 - = #SMOS [if OSMANY]
- Optional component selection
 - Some defaults [single]
 - All valid
 - ≥ 1 incompatible with slots
 - ≥ 1 incompatible with another selection
 - ≥ 1 incompatible with model
 - ≥ 1 not in database [error]



Next ...

- Category partition testing gave us
 - Systematic approach: Identify characteristics and values (the creative step), generate combinations (the mechanical step)
- But ...
 - Test suite size grows very rapidly with number of categories. Can we use a non-exhaustive approach?
- Pairwise (and n-way) combinatorial testing do
 - Combine values systematically but not exhaustively
 - Rationale: Most unplanned interactions are among just two or a few parameters or parameter characteristics

