LECTURE 02.
COVERAGE-BASED TECHNIQUES. PART I

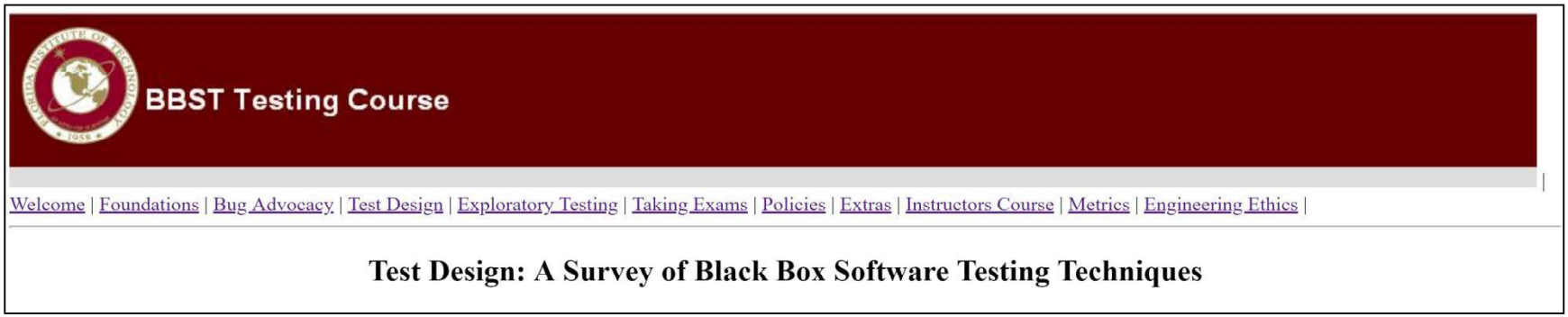
Test Design Techniques
[02 March 2022]

Elective Course, Spring Semester, 2021-2022

Camelia Chisăliţă-Creţu, Lecturer PhD
Babeş-Bolyai University

Acknowledgements

The course Test Design Techniques is based on the Test Design course available on the **BBST Testing Course** platform.



The BBST Courses are created and developed by **Cem Kaner, J.D., Ph.D.., Professor of Software Engineering at Florida Institute of Technology.**

Contents

- Coverage Measurement
- Coverage-based techniques
  - Definition. Focus
  - Techniques
    - Part I
      1. Tours
      2. Function. Function testing
      3. Feature or function integration testing
      4. Equivalence class analysis
      5. Boundary testing
      6. Best representative testing
      7. Domain testing
      8. Test idea catalogues
    - Part II
      9. Logical expressions
      10. Multivariable testing
      11. State transitions
      12. User interface testing
      13. Specification-based testing
      14. Requirements-based testing
      15. Compliance-driven testing
      16. Configuration testing
      17. Localization testing

Coverage Measurement

- **Coverage assesses the extent (or proportion) of testing of a given type that has been completed, compared to the population of possible tests of this type.**

$$\text{Coverage of items} = \frac{\text{number of items exercised}}{\text{total number of items}} \times 100$$

- **Coverage measures the amount of testing done of a certain type.**

Complete Coverage vs. Complete Testing

- **Complete testing means:**
  - **to run all distinct tests, i.e., exhaustive testing;**
  - **to test so thoroughly that there are no bugs left in the software.**
- **Coverage measures the amount of testing done of a certain type.**

Complete coverage does not mean complete testing.

Complete coverage ≠ complete testing.

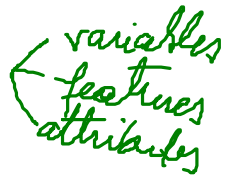
Complete coverage $\not\Rightarrow$ complete testing.

Complete testing ⇒ complete coverage.

- *Question:* **What do we have to do, to achieve complete testing?**
- *Answer:* **We cannot achieve complete testing. We might be able to achieve *adequate testing*…**

Coverage-based Test Techniques

- A coverage-based technique means
  - the tester should run every test of a given type;



- in practice, the tester probably will not run every test of each type, i.e., 100% coverage, but the tester might measure the coverage of that type of testing.
  - Coverage provides the confidence degree in using the product for that type of testing.

Coverage-based test techniques. Focus

Coverage-based techniques focus *how much* it is tested.

- a technique may be classified depending on what the tester has in mind when he uses it.

- E.g.: **Feature integration testing**
  - is **coverage-oriented** if the tester is checking whether every function behaves well when used with other functions;
  - is **risk-oriented** if the tester has a theory of error for interactions between functions.

Coverage-based Test Design Techniques

- **Coverage-based techniques:**

Part I

1. Tours (24 tour types);
2. Function testing;
3. Feature/function integration testing;
4. Equivalence class analysis;
5. Boundary testing;
6. Best representative testing;
7. Domain testing;
8. Test idea catalogues;

Part II

9. Multivariable testing;
10. Logical expressions;
11. State transitions;
12. User interface testing;
13. Specification-based testing;
14. Requirements-based testing;
15. Compliance-driven testing;
16. Configuration testing;
17. Localization testing.

COVERAGE TECHNIQUES. PART I

Tours (24 tour types), Function testing, Feature/function integration testing

Equivalence class analysis, Boundary testing, Best representative testing

Domain testing

Test idea catalogues

Tours. Example

- Consider the **Microsoft Office Word** application**;**
- Perform a tour based on basic editing aspects:
  - enter some text;
  - change several characteristics (font, size, colour, etc.);
- Build a **mind map** consisting of your **remarks** (notes) gathered during your investigation.

Tours. Definition

- **A tour is**
  - **an exploration of a product that is organized around a theme.**
  - a search through the product performed with the aim to create a collection of related information about the program.

- touring is functionally similar to a structured *brainstorming approach*; it is used for surfacing a collection of ideas, that are explored in depth, later, one at a time.

- **the core of exploration by touring is learning;**

> **Coverage: A specific type of tour creates a list of things to test, e.g., the function list, the list of error messages, then each member of the list get tested.**

Tours. Details. Outcome

- Touring is one of several starting points for **exploratory testing** as:
  - the tester might discover bugs or other quality-relevant information during a tour;
  - the tester might discover bugs when he follows up a tour with a deeper set of tests guided by his tour results.

- **The result of a tour is an inventory: a list of things of the type that the tester is interested in.**
  - a "complete" tour yields an exhaustive list; but **most tours are incomplete yet useful;**
  - later, the tester can test everything on the list, to some intentional degree of thoroughness;
    - E.g.: for the **Editing menu**; we may create an inventory of the program's functions; furthermore, **testing from that inventory is called function testing**.

Tours. Guidelines

- **Testers can perform touring together:**
  - touring in pairs is often *more productive* than touring for twice as long, alone.
  - new testers will benefit from a tour guide.
- **Testers can split tours across many sessions:**
  - there is no need to complete a tour in one day or one week.
  - It is recommended to perform tour for an hour or two, do some other tasks, then pick up the tour a few days later.

Tours. Classification

- There are over 20 types of tours:
  1. *Feature/Function Tour;
  2. *Menus and Windows Tour;
  3. *Mouse and Keyboard Tour;
  4. *Transactions Tour;
  5. *Error Message Tour;
  6. *Variables Tour (Variability Tour);
  7. Data Tour;
  8. Sample Data Tour;
  9. File Tour;
  10. Structure Tour;
  11. Operational modes Tour;
  12. Sequence Tour (Scenario Tour);
  13. Claims Tour;
  14. Benefits Tour;
  15. Market Context Tour;
  16. *User Tour;
  17. *Life history Tour;
  18. Configuration Tour;
  19. Interoperability Tour;
  20. Compatibility Tour;
  21. Testability Tour;
  22. *Specified-risk Tour;
  23. *Extreme Value Tour;
  24. Complexity Tour.

*Tours. Types (1)

- **Feature/Function Tour:**
  - **goal:** to find what the program can do;
    - it finds all the features, controls and command line options.
  - **result:** the list of all features, controls and command line options;
  - for each function the followings can be included:
    - input (params), output;
    - scope of the function;
    - options of the function;
    - error cases;
    - circumstances under which the function behaves differently.

Function List. Guidelines (1)

- Category of functions, e.g., Editing;
  - for individual functions, e.g., Cut, Paste, Delete,
    1. **inputs to the function**
       - *parameters*
         - maximum value;
         - minimum value;
         - other special cases;
    2. **outputs of the function**
    3. **possible scope of the function**, e.g., Delete word, Delete paragraph;
    4. **options of the function**, e.g. configure the program to Delete the contents of a row of a table, leaving a blank row or Delete the row along with its contents;
    5. **error cases that might be reached while using this function**;
    6. **circumstances under which the function behaves differently**, e.g. deleting from a word processor configured to track and display changes or not to track changes;

Function List. Guidelines (2)

- usually, testers add notes in order to describe:
  1. how he can decide
     - if the function works;
     - if the function does not work.
  2. how the function is used
     - what is the result (the consequence) of running this function?
  3. environmental variables that may constrain the function under test.

*Tours. Types (2)

- **Menus and Windows Tour:**
  - **goal:** to find all the menus (main and context menus), menu items, windows, toolbars, icons, and other controls.

- **Mouse and Keyboard Tour:**
  - **goal:** to find all the things that can be done with the *mouse* and the *keyboard*.
  - **tips:** perform click on everything, try all the keys on the keyboard, including F-keys, Enter, Tab, Escape, Backspace and combinations with Shift, Ctrl, and Alt.

*Tours. Types (3)

- **A transaction** is
  - a complete task; typically it involves a sequence of many functions
- **Transactions Tour:**
  - **goal:** to find all the transactions the program allows to perform.
  - E.g.: go to the store, buy something (including paying for it) is a transaction, i.e., a complete task;

- **transaction vs feature/function:**
  - *the mindset of the tester while he performs the tour is different*:
    - *transaction tour:* **What can I do with this program?**
    - *feature/function tour:* **What are the program's commands?**

*Tours. Types (4)

- **Error Message Tour:**
  - **goal:** to find all the error messages the program may produce;
  - **tips:**
    - **list every error message**; ask programmers for a list; look for a text file that stores program strings (including error messages);
      - E.g.: Process Explorer is a System Internals tool available on Microsoft's Web site that includes functions for dumping a program's strings;
    - **list every condition** that may throw an error message; highlight cases that should yield an error message but do not.

*Tours. Types (5)

- **A variable** is
  - a **program entity** that may have different values during program execution;
- **Variables Tour:**
  - **goal:** to find all the variables the user can change in the program;
  - **tips:** *questions to ask for each variable*:
    - What values can each variable take?
    - What are the ranges (domains) and range boundaries of each variable?
    - Some variables (such as many variables that are set with check boxes) enable, disable or constrain the values of other variables, i.e., *dependent variables*:
      - What changes in one variable will cause changes in other variables?

Tours. Types (6)

- **The data** elements of the application are
  - *variables* and
  - *constants*, or information the program reads from disk or obtains from other applications.
- **Data Tour:**
  - **goal:** to find the data that is processed by the program;

- **Variables tour vs Data tour**:
  - the tours are obviously related, but their emphasis is different;
  - the *variables tour* identifies **the variables in the program**;
  - the *data tour* considers **what values are fed to those variables and where those values come from**.

Tours. Types (7)

- **Sample Data Tour:**
  - **goal:** to find the sample data available in the program;
  - **tips:**
    - the tester should consider data from using or testing a previous version of the application or an application that should interoperate with the tested one.
    - the tester should check whether the program handles these data correctly?

Tours. Types (8)

- **File Tour:**
  - **goal:** to find the files used by this program and where are they located;
  - **tips:** *questions to ask*:
    - What's in the folder where the program's ***.exe*** file is found?
    - What other folders contain application data? The tester should check out the system's directory structure, including folders that might be available only to the system administrator.
    - Look for ***readme*** files, ***help*** files, *log* files, installation scripts, ***.cfg***, ***.ini*** files;
    - Look at the names of ***.dll*** files and extrapolate on the functions that they might contain or the ways in which their absence might undermine the application.

Tours. Types (9)

- **Structure Tour:**
  - **goal:** to find the **items** included in the complete product and where are they located;
  - **an item is anything that may be tested**:
    - E.g.: code, data, interfaces, documentation, hardware (security services required for access, cables), packaging, associated web sites or online services, etc.;

Tours. Types (10)

- **An operational mode is**
  - a formal characterization of the **status** of one or more internal data objects that affect system behavior." [Whittaker1997]
- in **state-model based testing**, **an operational mode is a visible state of the program**;
  - identifying all the operational modes (states) is one of the core tasks of state-model based testing; this is followed by identifying the transitions (from State X, we can go directly to States Y and Z) and then testing all the transitions.

- **Operational modes Tour:**
  - **goal:** to find all the operational modes (states) of the program.

Tours. Types (11)

- **A sequence (or sub-path) is**
  - a set of actions that takes the user to one state to another;
  - E.g.: the sequence "open a document --> print the document --> change the document --> start to save the document" takes the user to the Save-Document state;

- **Sequence Tour:**
  - **goal:** to find all relevant sequences in the program.
  - there are several sequences that may take the user from one state to the same destination state;
  - **tips:** *questions to ask for the identified sequences:*
    - What are they?
    - Which ones are interesting?

Tours. Types (12)

- **sequence vs (state) transition**
  - **sequence** = a **set of transitions** from a state to another in order **to achieve a certain result**;
  - *sequence tour:* **What can I do with this program?**
  - *operational-modes tour:* **What are the program's commands (transitions from this state)?**

- **sequence and transactions: What can I do with the program?**
- **(state) transition and feature/function: What are the individual elements of the program?**

Tours. Types (13)

- **Claims Tour:**
  - **goal:** to find all the claims the vendor makes about the product;
  - **tips:** *questions to ask:*
    - What is it suppose to contain? (specification-based testing)
    - What is it suppose to do? (specification-based testing)
    - Find published claims in manuals, help, tutorials, and advertisements;
    - Find unpublished claims in internal specifications and memos that make promises or define the product's intent.

Tours. Types (14)

- **Benefits Tour:**
  - **goal:** to identify all the benefits of the application;
  - **tips:**
    - Evaluate competing programs;
    - Read textbooks or product reviews;
  - tester's task is not to find bugs; it is to discover what the program will provide, i.e., **what's valuable about this program once it is fully working**;
  - this is a useful starting point for testers; the tester can communicate much more effectively about a product if he understands **what it should be good for**.

Tours. Types (15)

- **Market Context Tour:**
  - **goal:** to find the market context for the tested product;
  - **tips:** *questions to ask:*
    - Who are the competitors?
    - Why would people use other competitors' product?
    - Among the competitors, are there cheaper ones? Less capable ones? More capable? More expensive?
    - Why would someone buy this one instead of the other ones? Why would they buy the other ones?
  - **the source of information is similar to *benefits tour*, but the goal is to understand the product's place within its market.**
    - it helps to understand the product and to prioritize testing, select interesting data for tests;
    - it helps the tester to advocate in a certain way for certain bugs to be fixed.

*Tours. Types (16)

- **User Tour:**
  - **goal:** to find user types and what would they do with the product;
  - **tips:** *questions to ask:*
    - Imagine *five* different types of users of the tested application;
    - What would they do with it?
    - What information would they want from it?
    - How would they expect its features to work? (the benefits)

*Tours. Types (17)

- **The life history of an object** consists of
  - object creation point (starting point), continues as it is used and transformed, and it finishes (ending point) when it gets destroyed or terminated.

- **Life History Tour:**
  - **goal:** to find life histories (scenarios) for the same types of objects the program works with;
  - E.g.: *see next slide*.

*Tours. Types (18)

- E.g. **1. a telephone system**; when a person makes a call this represent an object to the system;
  - the object (the call) = data and capabilities (behaviour);
  - when hang up (the call ends) the object is destroyed;
  - when the call is made from outside the system, it is still a call but it is created in a different way;
- E.g., **2. a checking account in a banking application** and consider an object in the system;
  - *questions to ask:*
    - When and how can the program create it?
    - How can the program change it?
    - How is it used?
    - What does it interact with?
    - When does the program deactivate, discard or destroy it?
    - After the object is *terminated*, is any record kept of it, for future reference?

*Tours. Types (19)

- ***the end of object's life* is an ambiguous concept;**
  - *this ambiguity or misunderstanding of specifications may cause sometimes bugs;*
- the tester can create many scenarios to reflect different potential life histories for the same types of objects;
  - the testers may create various diagrams and discuss them with the programmers in order to clarify the meaning of the "the end of object's life" [Kaner2003].

Tours. Types (20)

- **Configuration Tour:**
  - **goal:** to find all the program configurations;
  - **tips:** *questions to ask:*
    - What are the program configurations?
    - What program *settings/configurations* can you change?
    - Which will persist if you exit the program and reopen it?
    - What operating system settings will affect application performance?

Tours. Types (21)

- **Interoperability Tour:**
  - **goal:** to find all **external items** the application interacts with;
  - **tips:** *questions to ask:*
    - Find every aspect of this application that communicates with **other software**, including device drivers, competing applications, and **external** clients or servers;
      - by communication we mean any aspect of the application that
        - creates data that other software will use or
        - reads data saved by other software.

- **interoperability testing** allows to investigate how well the program under test works (inter-operates) with other programs, i.e., **software** or **hardware**.

Tours. Types (22)

- **Compatibility Tour:**
  - **goal:** to find the **connected items** the application is compatible with, i.e., other devices or platforms;
  - **tips:** *questions to ask:*
    - What devices should the program work with?
    - What platforms, i.e., operating system and other system software, should the program run on?

Compatibility Testing vs Interoperability Testing

- **Compatibility testing** assess compatibility with
  - software or hardware that **are part of the tested product**;
- **Interoperability testing** assess compatibility with
  - software or hardware **external to the tested product**.
- **compatibility tour** is similar to **interoperability tour** but it refers to devices that are part of the tested product, i.e., **software** of **hardware**;
  - sometimes testers may refer to both of these tours by using the same term, i.e., *interoperability* or *compatibility*.

Tours. Types (23)

- **Testability Tour:**
  - **goal:** to find what makes the product easier or harder to test;
  - **tips:**
    - find all the features that can be used as testability features;
    - identify available tools that can help during testing;
  - **result:** usually, a testability tour results in is a **request for more testable features**;
    - E.g.: the software under test trades messages with another program to get its data; during testability tour, the tester might realize it would be useful to see those messages; therefore, he asks for a new feature, a log file that saves the contents of every message between these programs.

*Tours. Types (24)

- **A risk catalog (failure modes)** is
  - **a list of ways the tester thinks the product could fail;**
- creating risk catalogs involves many iterations of touring, categorizing, and brainstorming.
- **Specified-risk Tour:**
  - **goal:** to find ways the program could fail considering a specified risk;
  - **steps:**
    1. Build a catalog of risks.
    2. Imagine a way that the program could fail and then search for all parts of the program that **might fail in this way**.
       - E.g., The **extreme-value tour** is an example of this type of tour.
    3. Walk through the program asking, at each point, "what could go wrong here"?
       - **Failure ideas need to be sorted into categories and for each category, try to imagine other parts of the program that could fail in the same way.**

*Tours. Types (25)

- **Extreme Value Tour:**
  - it is an example on specified-risk tour;
  - **goal:** to find ways the program could fail considering the risk related to a variable extreme values;
  - **tips:** *questions to ask:*
    - What might cause problems for the variables?
    - What are the programmers' assumptions about incoming data? (coding habits)
  - E.g.:
    - **Zero** as value;
    - Different value than the expected one:
      - **small numbers** where large ones are expected;
      - **negatives** where positive ones are expected;
      - **huge** ones where modestly-sized ones are expected;
      - **non-numbers** where numbers are expected;
      - **empty values** where data is expected.
    - **Potentially unexpected units, formats, or data types.**

Tours. Types (26)

- **Complexity Tour:**
  - **goal:** to find the most complex aspects of the application;
  - **tips:** *questions to ask:*
    - **What will confuse the user?**
    - **What would lead the user to errors?**
    - Would program interactions lead the application to a fragile program state?
    - Should be considered features and challenging data sets as well;
    - Look for anything that is hard to describe or hard to explain.
    - Build the top “N” (e.g., 5) most complex tasks or aspects of the application.

Tours. Types (27)

- complexity tour is recommended especially when:
  - The underlying hypothesis is that **complex aspects of the program are more vulnerable to failure**, malware attack, or user dissatisfaction;
  - The touring tester looks for cases in which the program requires a **long sequence of user actions or combines many functions to achieve one result, or uses many variables at once**.

- **complexity tour vs feature tour = transaction tour vs feature tour;**
  - they are not performed both or a lot of both;
  - the focus will be more on one level of generality than the other.

Tours. Diversity applied in Exploration (1)

- **Exploratory (software) testing is**
  - **a style of software testing** that
  - emphasizes the **personal freedom and responsibility of the individual tester**
  - to **continually optimize the value** of his work
  - by treating
    - **test-related learning,**
    - test design,
    - test execution, and
    - test result interpretation as
  - mutually **supportive activities that run in parallel throughout the project**.

Tours. Diversity applied in Exploration (2)

- **Exploratory Testing** allows
  - to use empirical methods (tests) **to learn new things about the software**.
- **Testers run *new types of tests* all the time to gain *new knowledge*.**
  - Each type of tour allows to explore the program from a **different perspective**, looking for **different types of information**.

Tours. Classification

- There are over 20 types of tours:

1. *Feature/Function Tour;
2. *Menus and Windows Tour;
3. *Mouse and Keyboard Tour;
4. *Transactions Tour;
5. *Error Message Tour;
6. *Variables Tour (Variability Tour);
7. Data Tour;
8. Sample Data Tour;
9. File Tour;
10. Structure Tour;
11. Operational modes Tour;
12. Sequence Tour (Scenario Tour);
13. Claims Tour;
14. Benefits Tour;
15. Market Context Tour;
16. *User Tour;
17. *Life history Tour;
18. Configuration Tour;
19. Interoperability Tour;
20. Compatibility Tour;
21. Testability Tour;
22. *Specified-risk Tour;
23. *Extreme Value Tour;
24. Complexity Tour.

Tours. Conclusions

- **No one will use every type of tour.**
- A greater diversity of available tours enables greater diversity in testing.
  - People have different interests, backgrounds and skills. Some testers will explore a program in very different ways from other testers.
  - E.g.: **complexity tour** might be rare in practice and the follow-up, i.e., risk-based testing, will be hard work. But someone who has the knowledge and skill to do this will learn things about the software's potential weaknesses (and ways to navigate the program) that **less challenging tours will miss**.
- Some tours seem more like taking a simple inventory; others use more aggressive testing.
  - What is common to the tours is that **the tester does what is necessary to identify a set of information of a desired type**.
  - Testing is achieved to the level of creativity and depth needed to uncover the required information. The tester structures deeper testing, guided by the tour's results, later.

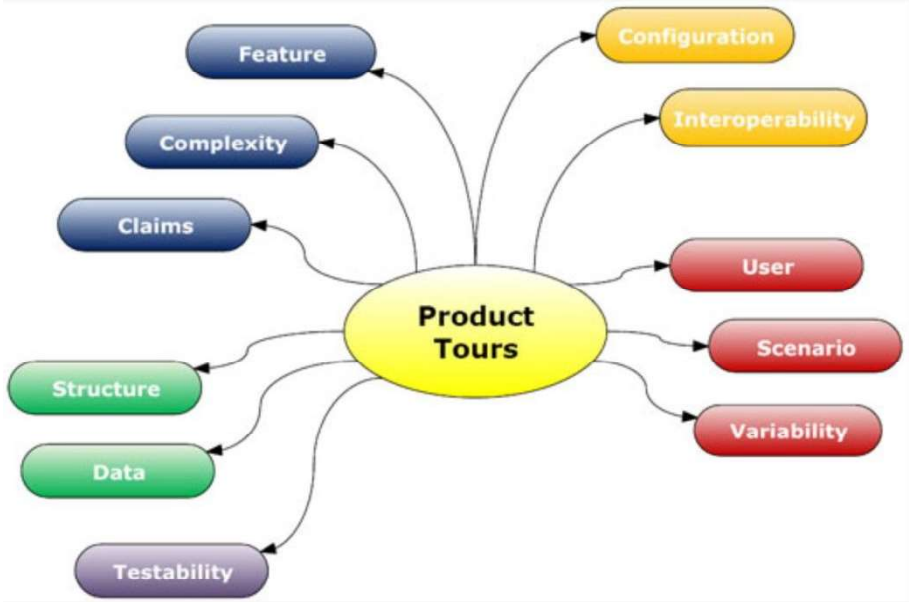
Tours. Mnemonics

- **F**eature tour
- **C**omplexity tour
- **C**laims tour

- **C**onfiguration tour
- **U**ser tour
- **T**estability tour
- **S**cenario tour (sequence tour)

- **V**ariability tour (variable tour)
- **I**nteroperability tour
- **D**ata tour
- **S**tructure tour

- **FCC CUTS VIDS**



[JonathanKohl2006], [DevelopSense2009]

Function Testing. Example

- Continue to work on the **Microsoft Office Word** application**;**
- Choose a function from the ones previously identified;
  - determine what it does and
  - if it works as expected.

Function. Definition

- **A function is**
  - **something the product can do.**
- Functions might also be called
  - **features** or
  - **commands** or
  - **capabilities.**

Function Testing. Definition

- **Function Testing tests each feature or function on its own.**
- **Objective:** to scan through the product such that:
  - cover **every** function or feature and
  - with at least enough testing to determine
    - **what it does and**
    - **whether it is (basically) working.**
- **this objective is not always achieved.**

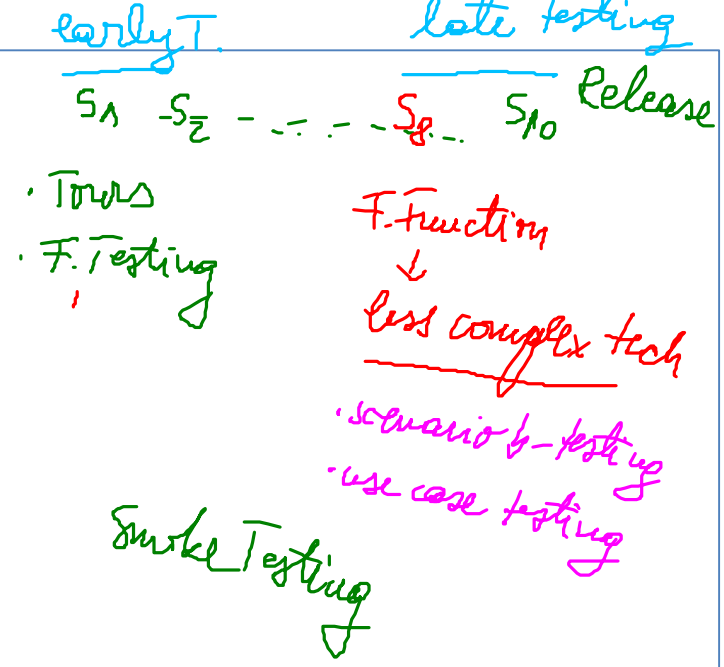
Coverage: Function testing provides the testing coverage of the features of the product.

Function Testing. Function List

- in order to perform function testing, testers need to identify program functions or features;
- there are several ways testers can learn about the program they test, i.e., to identify what they will test (the program functions) and to build a **function list**:
  1. check specifications or the draft user manual;
  2. walk the user interface, i.e., **various types of tours**;
     - the tester performs function tour and then he organizes the testing to achieve any level of *function coverage* that he chooses.
  3. try commands at the command line;
  4. search the program or resource files for command names.
- **Function list is an outline of the program's capabilities.**
- **the best tools for creating function list are mind mapping tools;**
  - E.g.: XMind, Mind Manager, Nova Mind, FreeMind.

Function Testing. Ways to use it

- Function testing *may be effectively* applied in:
  - early product testing;
  - smoke testing;
- Function testing has *several drawbacks* when applied:
  - beyond early testing;
  - as main testing technique.



Function Testing. Early Testing (1)

- Function testing applied **in early testing** means *to test sympathetically*;
- **sympathetically testing** means to understand:
  - what are the developers trying to achieve with this program?
  - what would it do if it worked properly?
  - why would people want to use this?
  - what are its strengths?

- **some testers go straight on finding bugs without considering product's benefits.**

Function Testing. Early Testing (2)

- Complex tests that use many functions in early testing:
  - running **complex tests early may block the entire test** => blocking bugs might prevent from discovering a broken feature until late in the project;
- Fast scan for serious problems
  - some aspects of the product are so poorly designed or so poorly implemented that they must be redone and
    - it is better to realize this early,
    - it is pointless to test an area in detail if it will be replaced (**redesigned**, redone).

Function Testing. Smoke Testing

- **smoke testing = build-verification testing;**
- particularities of smoke testing:
  - most tests in the test suite are function tests;
  - relatively small set of tests run whenever there is a new build.

- in some companies, testers design the smoke tests and give them to the programmers, who run the smoke tests as part of their build process.

Function Testing. Beyond Early Testing

- beyond early testing the tester runs more complex tests that involve several features;
- the function list can serve as a coverage guide for this testing; the tester may intend to reach:
  - *every feature* or
  - *every sub-function of every feature* or
  - *every option of every feature*.

  ==> to build scenarios.

Function Testing. Main Testing Technique (1)

- when function testing is applied as the main test technique:
  - functions are tested one by one thoroughly;
  - the function list is extended to include more detail;
  - the specific tests suggested by the list are run;
  - the testing continues using the list as a foundation.

- **Function testing is not recommend as the main testing technique.**

Function Testing. Main Testing Technique (2)

- **Function testing is primarily a test of capability of *individual units of the software*.**
- Function testing does not emphasize:
  - **interactions of features**;
  - special values of data, and interactions of values of several variables;
  - **missing features**;
  - **user tasks** – whether the customer can actually achieve benefits promised by the program;
  - interaction with background tasks that are effects of interrupts;
  - responsiveness and how well the program functions under load;
  - **usability**, **scalability**, interoperability, testability, etc.

Feature Integration Testing. Definition

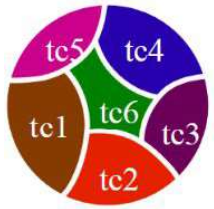
- **Feature Integration Testing tests several features or functions together.**
- It is performed with functions that
  - **will often be used together**
    - E.g.: in a spreadsheet, sum part of a column, then sort data in the column. Sorting should change the sum if and only if are sorted different values into the part being summed.
  - **work together to create a result**
    - E.g.: select a book, add it to a shopping cart, pay for the book.

Coverage: Feature integration testing provides coverage of the interactions of the product's features.

Lecture 03

Equivalence Class Analysis. Definition

- **An equivalence class** is a set of values for a **variable** that you consider equivalent. Test cases are equivalent if
  - they test the same thing,
  - if one of them catches a bug, the others probably will too, and
  - if one of them doesn't catch a bug, the others probably won't either.



- The set of values that can be assigned to a variable is the variable's domain;
- Equivalence class analysis divides a variable's domain into non-overlapping subsets (partitions) that contain equivalent values;
- **Equivalence class analysis allows test one or two values from each partition.**
- **Equivalence class-based testing makes testing more efficient by reducing redundancy of the tests. It increases confidence in the product when it is used in usual (normal) scenarios.**

Coverage: Equivalence class analysis test all the equivalence classes of every variable.

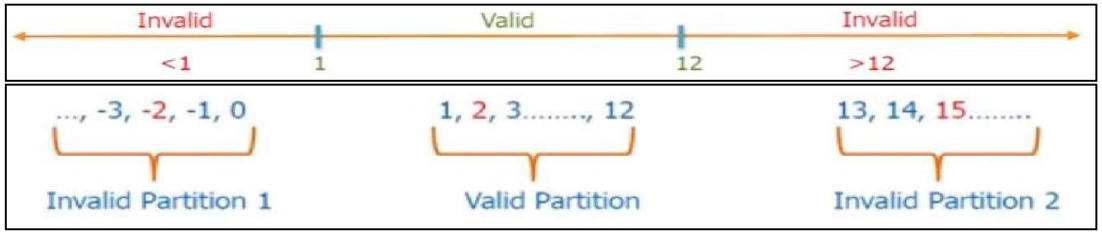
Equivalence Class Analysis. Example (1)

- Let be a form that allows contest registration. Among the information required, there is details on the birth date given as **day**, **month** and **year**, as **separate input text fields**. If the input data is not valid an error message will be shown.
- Test case design by Equivalence Class identification for the input text field month that has the values in the domain [1, 12].

Primary dimension:
A number >=1 and <=12;
- **One valid EC:**
  EC_1: D_1 = [1, 12];
- **Three invalid ECs:**
  EC_2: D_2= {month<1} = (-∞, 1);
  EC_3: D_3= {month>12} = (12, +∞);
  EC_4: D_4= symbols/alphabet letters.

Secondary dimensions:
- ***The order of the month within the year:*** *1st, 2nd, 3rd, etc.*
- ***Number of digits:*** *0 digits, 1 to 2 digits (1 to 12), 3 digits;*
- ***Leading spaces:*** *0 (typical case), >=1 (not unusual);*
- ***Spaces between digits:*** *0 (typical), >0 (some programs (OOWriter) ignore "invalid" characters inside a number given as string)*
- ***ASCII codes:*** *for digits (48 to 57), non-digits (58 to 127), etc.*

Equivalence Class Analysis. Example (2)



- **identified ECs:**
  - One valid EC: **EC_1: D_1 =[1, 12];**
  - Three invalid ECs: **EC_2: D_2= {month| month<1} = (-∞, 1); EC_3: D_3= {month| month>12} = (12, +∞); EC_4: D_4 = symbols/alphabet letters.**
- **Designed test cases based on identified ECs:**
  - One valid EC ==> one valid test case, e.g., **TC_{01}: month = 2;**
  - Three invalid ECs ==> three invalid test cases, e.g., **TC_{02}: month =-2, TC_{03}: month = 15, TC_{04}: month = "%L10";**

  - **From each EC identified a single value will be chosen. Equivalence class testing considers that the program handles all the values from the same EC in a similar manner.**

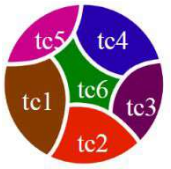
Equivalence Class Analysis. Rules (1)

1. **if a condition states the value belongs to the range [a,b]:**

- ==> 1 valid EC, 2 invalid ECs;
  - E.g.: see the previous example on month variable, where the valid range is [1, 12];

2. **if a condition states the value belongs to finite set of values:**

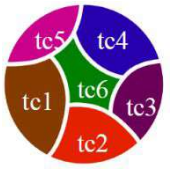
- ==> 1 valid EC **for each value**, 1 invalid EC;
  - E.g.: `type ∈ CourseType = {optional, mandatory, non-compulsory};`
  - 1 valid EC for each item in set `CourseType`:
    - **EC_1:** {`optional`},
    - **EC_2:** {`mandatory`},
    - **EC_3:** {`non-compulsory`} ===> 3 valid ECs;
  - 1 invalid EC:
    - **EC_4:** M= $\{e \mid e \notin$ `CourseType`$\}$;



Equivalence Class Analysis. Rules (2)

3. **if a condition states the number of values:**

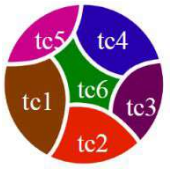
- ==> 1 valid EC, 2 invalid ECs;
  - E.g.: “from 1 to 5 students”;
  - 1 valid EC:
    - **EC_1:** D=[1,5];
  - 2 invalid ECs:
    - **EC_2: no student**, i.e., empty set;
    - **EC_3: too many students**, i.e., more than 5 students.



Equivalence Class Analysis. Rules (3)

4. **if a condition states the case of “must be”:**

- ==> 1 valid EC, 1 invalid EC;
  - E.g.,: “the first symbol in the password must be a digit”;
  - 1 valid EC:
    - **EC_1:** first symbol is a digit;
  - 1 invalid EC:
    - **EC_2:** first symbol is not a digit.



Equivalence Class Analysis. Coverage

- ECs coverage for equivalence class analysis:

$$\text{EC Coverage} = \frac{\text{number of tested ECs}}{\text{number of identified ECs}} \times 100$$

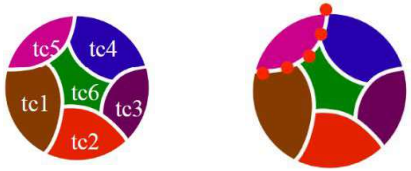
- E.g.:
  - following the specifications the tester has identified 18 ECs for input and output data;
  - he achieved to design, implement and run tests for 15 ECs by now;
  - **EC Coverage = (15/18)*100 = 83,33%.**

- **EC Coverage may be used as exit criteria.**

Boundary Testing. Definition

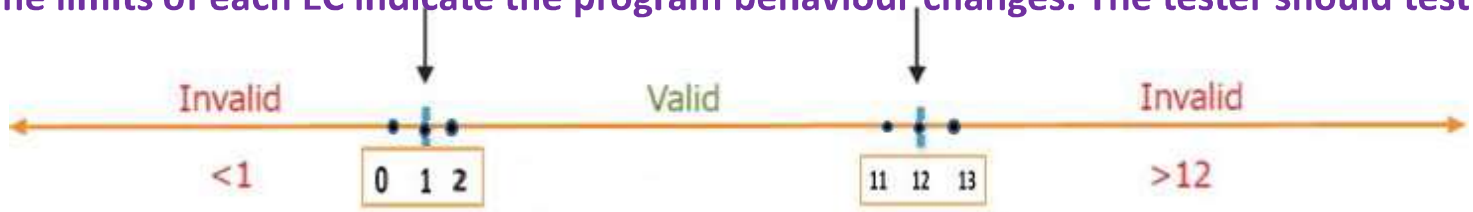
- **Boundary-value testing allows to partition the values of a variable into its equivalence classes and then test the upper and lower bounds of each equivalence class.**
- A boundary value is a particularly good member of an equivalence class to test because:
  - it carries the same risks as all the other members of the class;
  - boundary values carry an additional risk because off-by-one errors are common.
- **Boundary-value testing adds a risk model to equivalence-class based testing.**
- **Boundary-value testing is applied when bug hunting.**

> Coverage: **Boundary testing ensures that every boundary of every variable is tested, i.e., covered.**



Boundary Testing. Example (1)

- **The limits of each EC indicate the program behaviour changes. The tester should test them!**



- **identified ECs:**
  - One valid EC: **EC_1: D_1 =[1, 12];**
  - Three invalid ECs: **EC_2: D_2= {month| month<1} = (-∞, 1); EC_3: D_3= {month| month>12} = (12, +∞); EC_4: D_4 = symbols/alphabet letters.**
- Boundary value conditions (BVC) built on the limits of the valid ECs:

• Lower limit of EC_1: • 1. month = 0; (invalid value) • 2. month = 1; • 3. month = 2;	• Upper limit of EC_1: • 4. month = 11; • 5. month = 12; • 6. month = 13; (invalid value)

Boundary Testing. Example (2)

- Valid ECs identified:
  - One valid EC: **EC_1: D_1 =[1, 12];**

- **Designed test cases following the BVCs identified:**
  - Lower limit of EC_1:
    - 1. month = 0 ==> **TC_{01}: month= 0; (invalid value)**
    - 2. month = 1 ==> **TC_{02}: month= 1; (valid value)**
    - 3. month = 2 ==> **TC_{03}: month= 2; (valid value)**
  - Upper limit of EC_1:
    - 4. month = 11 ==> **TC_{04}: month= 11; (valid value)**
    - 5. month = 12 ==> **TC_{05}: month= 12; (valid value)**
    - 6. month = 13 ==> **TC_{06}: month= 13; (invalid value)**

Boundary Testing. Exceptions on BVC Identification

- **ECs does not have boundaries** (limits):
  - E.g.: sets {optional, mandatory, non-compulsory} and {Mr, Miss, Mrs, Dr, PhD};
- **ordered ECs with a single boundary** (either lower or upper bound);
  - E.g.: deposit amount into a bank account;
- **multiple dependent variables:**
  - E.g.: variables: card number, issue date, expiry date, owner name;
    - all variables have valid values and the constraints between them are satisfied <==> valid card;
    - if a single variable has invalid values ==> invalid card;
- **dependent ECs** – value of one variable depends upon another
  - E.g.:, in OpenOffice Writer there are available various page format: page format= {A2, A3, A4}; A4 page format constraints header's height no higher than 20.56 cm.

Boundary Testing. ECs and their Limits

EC Type	Does it have boundaries (limits)?
Range of values	Yes
Number of values	Yes
Unordered set	**No**
Ordered set	**Yes**
"must be" value	No
Sequence	Yes
Dependent ECs	**Yes**
Multiple dependent variables	**No**

Boundary Testing. Rules

1. **if an input/output condition states the value belongs to the range [a,b]:**

- ==> tests for:
  - (1) valid BVCs – the range bounds and values next to those (e.g., a, a+1; b-1, b);
  - (2) invalid BVC – values beyond and above the bounds (e.g., a-1, b+1);

2. **if an input/output condition states the value belongs to finite <u>ordered</u> set of values:**

- ==> tests for:
  - (1) valid BVCs – the first and the last item in the set;
  - (2) invalid BVCs – the values next to the lowest and highest values in the set;

3. **if an input/output condition states the number of values (e.g., "from 1 to 5 students"):**

- ==> tests for:
  - (1) valid BVCs – the lowest and the highest values, i.e., 1 and 5;
  - (2) invalid BVCs – the values next to the lowest and the highest values, i.e., 0 and 6.

Boundary Testing. Coverage

- BVCs coverage is computed as:

$$\text{BVC Coverage} = \frac{\text{number of tested BVCs}}{\text{number of identified BVCs}} \times 100$$

- E.g.:
  - Following the specifications the tester has identified 64 valid BVCs for input and output data;
  - He achieved to design, implement and run tests for 48 BVCs by now;
  - **BVC Coverage = (48/64)*100 = 75%.**

- **BVC Coverage may be used as exit criteria.**

Best Representative Testing. Definition

- **Best representative testing allows to test variable values that are *most likely to cause a failure*.**
- the best representative of a partition (*of the domain of a variable*) is
  - the one most likely to cause the program to fail a test.
- E.g.:
  - if the values in the domain **can be ordered** from **small to large**, then best representatives are typically **boundary values**;
  - if the values in the domain **cannot be ordered**, then a best representative is identified by considering more than one risk; **two values of a variable might be equivalent with respect to one risk, but not with respect to the other**.
  - if all values in a partition are **truly equivalent** then **any of them** may considered as a best representative.
- the "best representatives" generalizes *domain testing to non-ordered sets and to secondary dimensions*.

Coverage: Best representative testing ensures that every best representative of every variable, relative to every risk is tested (covered).

Domain Testing. Definition

- **Domain testing formalizes and generalizes equivalence class and boundary analysis.**
- Steps:
  1. partition the variable's domain into **equivalence classes** and test **best representatives;**
  2. test **output domains** as well as **input domains;**
  3. test **secondary** as well as **primary dimensions**;
  4. test consequences as well as **input filters**;
  5. test **multidimensional variables** and **multiple variables** together.

Coverage: **Domain testing ensures that each equivalence class and every best representative of every variable are tested, i.e., covered.**

Domain Testing. Example (1)

- **Primary dimension, identified ECs:**
  - One valid EC: **EC_1: D_1 =[1, 12];**
  - Three invalid ECs: **EC_2: D_2= {month| month<1} = (-∞, 1); EC_3: D_3= {month| month>12} = (12, +∞); EC_4: D_4 = symbols/alphabet letters.**
- **Designed test cases following the domain testing technique, considering the *primary dimension*:**

Variable	Valid EC	Invalid EC	Boundaries& special cases	Notes
month		>12	13	**Error message**
	1-12		12	
			1	
		<1	0	**Error message**

Domain Testing. Example (2)

- **Secondary dimension:** *Number of digits: 0 digits, 1 to 2 digits (1 to 12), 3 digits, Leading spaces: 0 (none, typical case), 1 (not unusual), etc.*
- **Designed test cases following the domain testing technique, considering the *primary* and *the secondary dimensions*:**

Variable	Dimension	Valid EC	Invalid EC	Boundaries& special cases	Notes
month	primary		>12	13	Error message
		1-12		12	
				1	
			<1	0	Error message
	secondary		0 digits	none	Default value is deleted
		1 to 2 digits		1	
				12	
			3 digits	122	Error message

Test Idea Catalogs. Definition

- **A test idea catalog (or matrix) is a standard set of tests for a specific type of object (or risk) that is developed and reused for similar things in current product and later products.**
- E.g.: test idea catalogs for [KanerBachPettichord2001]:
  - numeric input variables, rational numbers (of various precisions), character fields (of various widths), filenames, file locations, dates, etc.;
- if the testers runs into one type of input field, product after product or time after time in the same product that he is testing, it is important to save time by creating from the beginning such reusable matrix.

Coverage: **A test idea catalog lists the test ideas to cover.**

Test Idea Catalogs. Example (1)

- **v in [LB, UB]**

Numeric Input Field	Nothing	Empty (clear default)	0	LB-1	LB	UB	UB+1	Far below LB	Far above UB	UB number of chars	UB +1 chars	Far beyond UB chars	Negative	Non-digit (< ASCII 47)	Non-digit (> ASCII 58)	wrong data type	expressions	Leading spaces	Non-printing char	O/S file name	Upper ASCII	Upper case	lower case	Modifiers (Ctrl, Alt, etc.)	Function keys

[KanerBachPettichord2001]

Test Idea Catalogs. Example (2)

- Test matrix consists of:
  - **Columns**: the tests that will be used every times is required;
  - **Rows**: the fields that will be tested;
- E.g.: the typical **Print dialog,** where one of the fields is ***Number of Copies***;
  - The range of valid values for Number of Copies is typically [1..99] or [1..255] (depending on the printer);
  - In the matrix the tester writes **Print: Number of Copies** on one row, then he runs some or all of the tests on that field, and then he fills in the results accordingly;
  - It is recommended to use **green** and pink highlighters to fill in cells that yielded passing and failing test results;
- the test matrix provides a structure for easy delegation:
- when a new feature is added or a feature is changed late in the project, the tester can assign several of these standard matrices to a tester who is relatively new to the project (but experienced in testing); his task is to check for basic functioning (of the new feature or continued good functioning) of older features that the tester expects to be impacted by the one that was changed. [KanerBachPettichord2001]

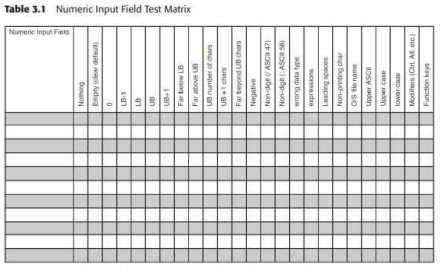


Table 3.1 Numeric Input Field Test Matrix

Numeric Input Field	Nothing	Empty (clear default)	0	LB-1	LB	UB	UB+1	Far Below LB	Far above UB	UB number of chars	UB +1 chars	Far Beyond UB chars	Negative	Non-Digit (ASCII 47)	Non-Digit (ASCII 58)	wrong data type	expressions	Leading spaces	Non-printing char	O/S file name	Upper ASCII	Upper case	boot code	Modifiers (Ctrl, Alt, etc.)	Function keys

Lecture Summary

- We have discussed:
  - Coverage-based techniques Part I
    1. Tours (24 types of tours);
    2. Function testing;
    3. Feature/function integration testing;
    4. Equivalence class analysis;
    5. Boundary testing;
    6. Best representative testing;
    7. Domain testing;
    8. Test idea catalogues.

Next Lecture

- Coverage-based Techniques Part II
    9. Logical expressions
    10. Multivariable testing
    11. State transitions
    12. User interface testing
    13. Specification-based testing
    14. Requirements-based testing
    15. Compliance-driven testing
    16. Configuration testing
    17. Localization testing

Lab Activities on week 01-02

- Tasks to achieve in week 01-02 during Lab 01:
  - **Play the game *"Testing is…"*** and **vote your favorite testing definition**;
  - **Work with a *mind mapping tool*** and **elaborate a mind map for a testing concept** presented in **Lecture 01** or **Lecture 02**.

References

- **[Kaner2003]** Cem Kaner, An introduction to scenario testing, http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf, 2003.
- **[JonathanKohl2006]** Jonathan Kohl, *Modeling Test Heuristics*, http://www.kohl.ca/2006/modeling-test-heuristics/, 2006.
- **[DevelopSense2009]** Developer Sense, *Blog: Of Testing Tours and Dashboards,* http://www.developsense.com/blog/2009/04/of-testing-tours-and-dashboards/, 2009.
- **[Whittaker1997]** Whittaker, J.A. (1997). Stochastic software testing. Annals of Software Engineering, 4, pp. 115-131.
- **[BBST2011]** BBST – Test Design, Cem Kaner, http://www.testingeducation.org/BBST/testdesign/BBSTTestDesign2011pfinal.pdf.
- **[BBST2010]** BBST – Fundamentals of Testing, Cem Kaner, http://www.testingeducation.org/BBST/foundations/BBSTFoundationsNov2010.pdf.
- **[KanerBachPettichord2001]** Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: Chapter 3: Test Techniques, http://media.techtarget.com/searchSoftwareQuality/downloads/Lessons_Learned_in_SW_testingCh3.pdf