LECTURE 05.
RISK-BASED TECHNIQUES. PART I

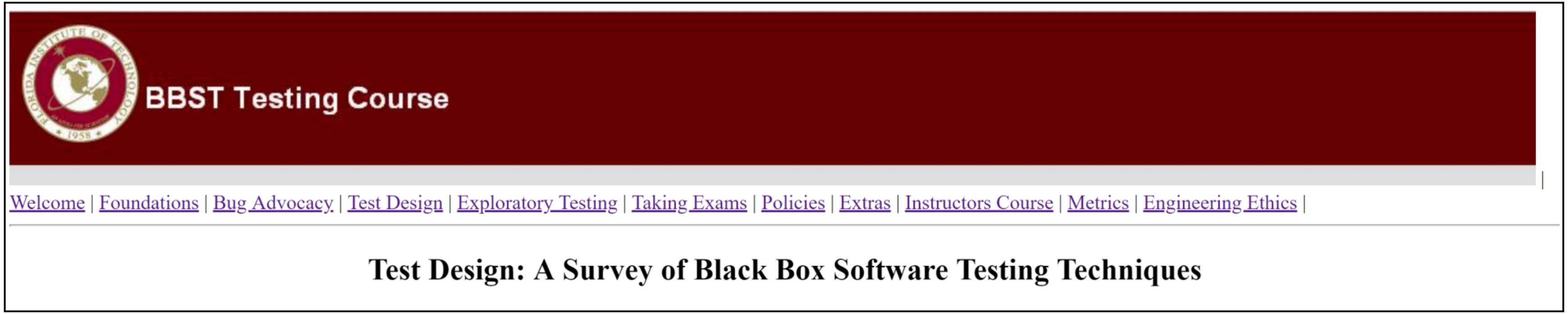
Test Design Techniques
[23 March 2022]

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

- Last lecture…
- Terminology
- **Part I**
  - Risk
  - Risk Approaches to Software Testing
    - Guidewords. HTSM
    - Risk Catalogs
    - Project-level Risks
    - Specific Risk-based Techniques
      - *Quick-tests*
- **Part II**
  - Specific Risk-based Techniques
    - *[Quick-tests]*
    - Constraints
    - Logical expressions
    - Stress testing
    - Load testing
    - Performance testing
    - History-based testing
    - Risk-based multivariable testing
    - Usability testing
    - Configuration / compatibility testing
    - Interoperability testing
    - Long sequence regression

Last Lecture...

- Topics approached from Lecture 02 to Lecture 04:
  - **Coverage-based techniques**
  - Techniques:
    - Tours;
    - Function testing;
    - Equivalence Class Partitioning;
    - Boundary Value Analysis;
    - Domain Testing;
    - Multivariable Testing;
    - Specification-based Testing;
    - Configuration Testing;
    - etc.

TDTs Taxonomy

- The main test design techniques are:
  - **Black-box approach:**
    - **Coverage-based techniques;**
    - **Risk-based techniques;**
    - **Activity-based techniques;**
    - **Tester-based techniques;**
    - **Evaluation-based techniques;**
    - **Desired result techniques;**
  - **White-box approach:**
    - **Glass-box techniques.**

Test Case. Attributes

- **A test case** is
  - a question you ask the program. [BBST2010]
  - we are more interested in the *informational goal*, i.e., to gain information; e.g., whether the program will pass or fail the test.
- Attributes of relevant (good) test cases:

•Power	**•Representative**	**•Coverage**	**•Supports troubleshooting**
•Valid	**•Non-redundant**	**•Easy to evaluate**	**•Appropriately complex**
•Value	**•Motivating**	**•Maintainable**	**•Accountable**
•Credible	**•Information value**	**•Performable**	**•Affordable**
		•Reusable	**•Opportunity Cost**

- **A test case has each of these attributes to some degree.**

Risk. Definition

- **Risk** means
  - **the possibility of suffering harm or loss.**

- there are *three dimensions* of risk in software testing:
  - **testers**: *design tests*;
    1. **How the program could fail?**
  - **managers**: *achieve risk analysis and prioritize testing*;
    2. **How likely it is the program could fail in a specific way?**
    3. **What consequences of a failure could be?/ How serious would a specific failure be?**

Risk. Testers' View

- **Testers**: *design tests;*
    1. **How the program could fail?**
- testers spend time thinking about the ways the program can fail;
- **goal: design tests that can expose the potential failures;**

Risk-based *test design* focuses on how the product can fail. The goal is to find bugs.

- testers perform risk-based test design.

Risk. Managers' View

- **Managers**: *achieve risk analysis and prioritize testing*;
    2. **How likely it is the program could fail in a specific way?**
    3. **What consequences of a failure could be?/ How serious would a specific failure be?**
- managers spend time doing risk analysis **to prioritize testing in order to reduce the likely consequences of the program's failures**;
- **goal: reduce the risk and risk consequences on the product;**

> **Risk-based *test management* focuses on the likelihood and cost of failures. The goal is to set appropriate priorities so that the highest-risk parts of the program receive the *most resources*.**

- managers perform risk-based test management, a management related activity (*not* testing).

Risk-based Test Techniques

- **A risk-based technique means**
  - **the tester should design and run tests that can make the program to fail.**
- Steps:
  1. imagine how the program can fail;
  2. design tests that expose these (potential) failures, i.e., problems of a specific type.

Risk-based Test Techniques. Focus

Risk-based techniques focus on <u>*why*</u> it gets tested, <u>*what risks*</u> it gets tested for.

- 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.

Risk-based techniques intend to find ways the program may fail. They do bug hunting.

Risk-based Testing. Approaches

- there are several approaches on risk:
  1. Exploratory Guidewords;
  2. Risk Catalogs;
  3. Project-level Risks;
  4. Specific Risk-based Techniques.

Risk-based Test Design Techniques

- **Specific risk-based techniques:**
  - Quick-tests;
  - Boundary testing;
  - Constraints;
  - Logical expressions;
  - Stress testing;
  - Load testing;
  - Performance testing;
  - History-based testing;
  - Risk-based multivariable testing;
  - Usability testing;
  - Configuration / compatibility testing;
  - Interoperability testing;
  - Long sequence regression.

Risk. Approaches

- there are several approaches on risk:
  1. **Exploratory Guidewords;**
  2. Risk Catalogs;
  3. Project-level Risks;
  4. Specific Risk-based Techniques.

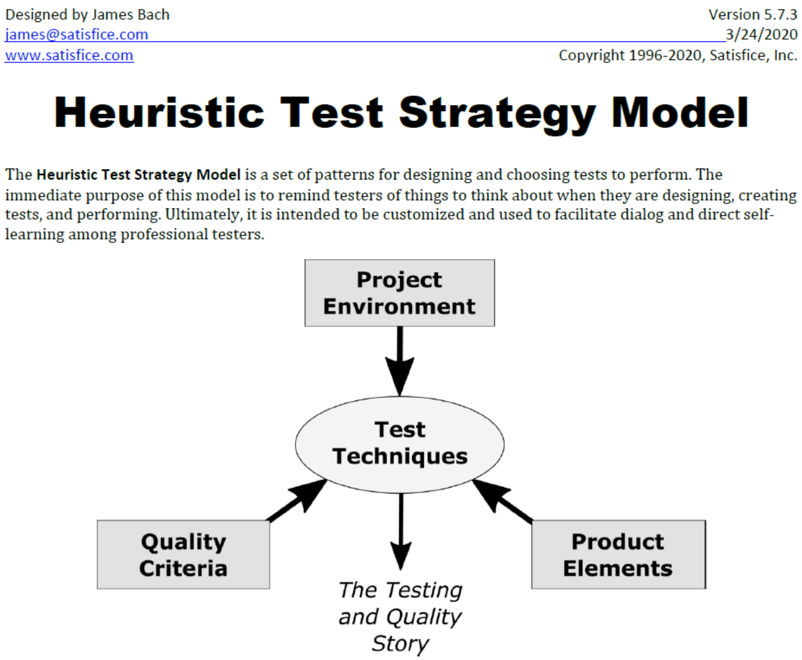
Exploratory Guidewords

- **critical safety systems testing** benefits of extensively using **guidewords**, i.e., keywords;
- guidewords are widely used in HAZOP (**ha**zard & **op**erability) analysis;
- **a list of guidewords** is
  - a list of **actions** or **objectives** or **risks** that the tester want to apply when he investigates each part of the system, e.g., components, variables, etc.;

- E.g.: considering a **list of elements** formed of *some variables*;
  - the **guidewords list** may consists of the items: *empty, small, large*;
  - for each variable we may ask:
    - What would happen if the value of this variable was **empty**?
    - What would happen if the value of this variable was very **small**?
    - What would happen if the value of this variable was very **large**?
  - **We would consider that every condition (empty, small, large) could cause failure and if so, what the failure would look like.**

Exploratory Guidewords. HTSM (1)

- an extended example of Guidewords is Heuristic Test Strategy Model (HTSM), developed by James Bach [Bach2006];
- HTSM is more detailed than HAZOPs list and is structured in layers;
- HTSM v.5.7.2 consists of 3 layers;
  - E.g.: **Product elements --> Structure --> Code;**
- the goal is similar to HAZOPs, i.e., **to evaluate each part of the system under test from several perspectives, identifying a diverse collection of risks.**



Exploratory Guidewords. HTSM (2)

- the **Project Environment** consists of elements that constrain what can be done in testing or that facilitate testing or test management:
  - Mission
  - Information
  - Developer relations
  - Test team
  - Equipment & Tools
  - Schedule
  - Test Items
  - Deliverables

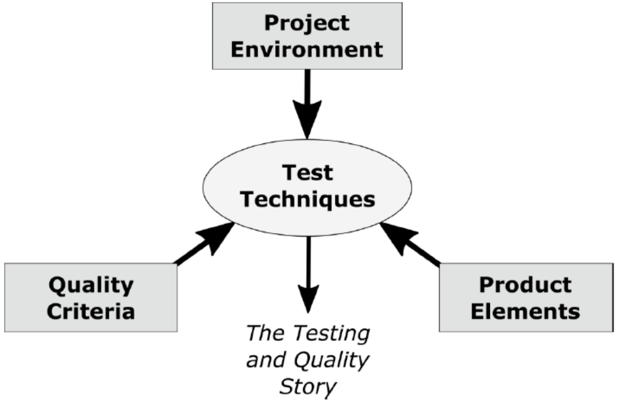
Designed by James Bach
james@satisfice.com
www.satisfice.com

Version 5.7.3
3/24/2020

Copyright 1996-2020, Satisfice, Inc.

Heuristic Test Strategy Model

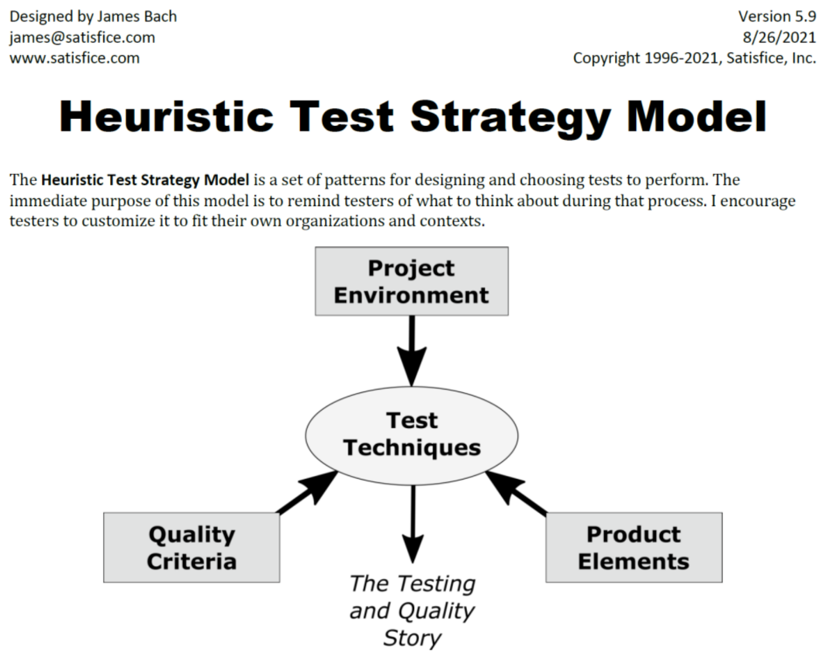
The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of things to think about when they are designing, creating tests, and performing. Ultimately, it is intended to be customized and used to facilitate dialog and direct self-learning among professional testers.



Exploratory Guidewords. HTSM (3)

- The **Products Elements** consists of the content of the application under test;
  - Structure
  - Function
  - Data
  - Interfaces
  - Platform
  - Operations
  - Time

Product elements indicates what gets tested.


Designed by James Bach
james@satisfice.com
www.satisfice.com

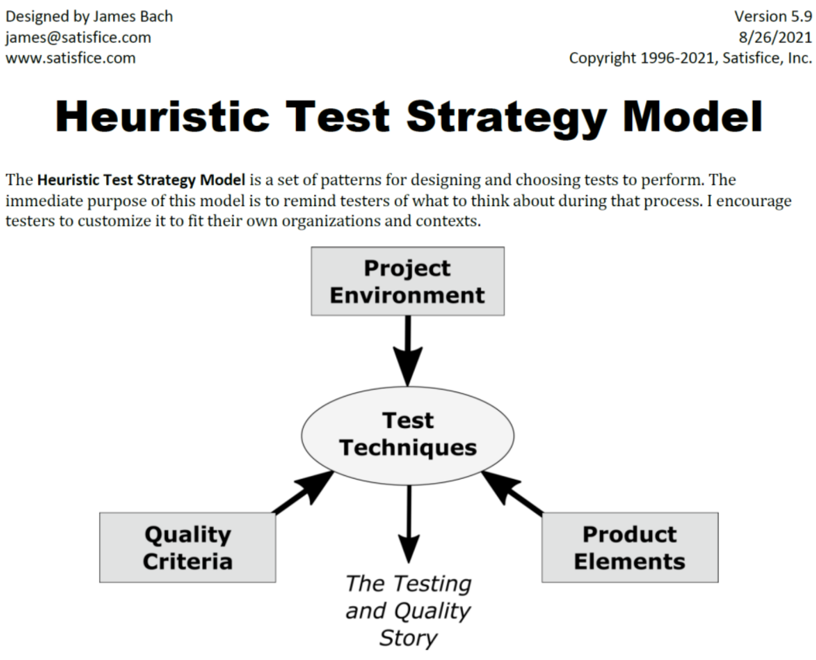
Version 5.9
8/26/2021
Copyright 1996-2021, Satisfice, Inc.

Heuristic Test Strategy Model

The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of what to think about during that process. I encourage testers to customize it to fit their own organizations and contexts.

Exploratory Guidewords. HTSM (4)

- The **Quality Criteria** consists of two categories:
  - **Operational criteria**
    - Capability
    - Reliability
    - Usability
    - Charisma
    - Security
    - Scalability
    - Compatibility
    - Performance
    - Installability
  - **Development criteria**
    - Supportability
    - Testability
    - Maintainability
    - Portability
    - Localizability


Designed by James Bach
james@satisfice.com
www.satisfice.com

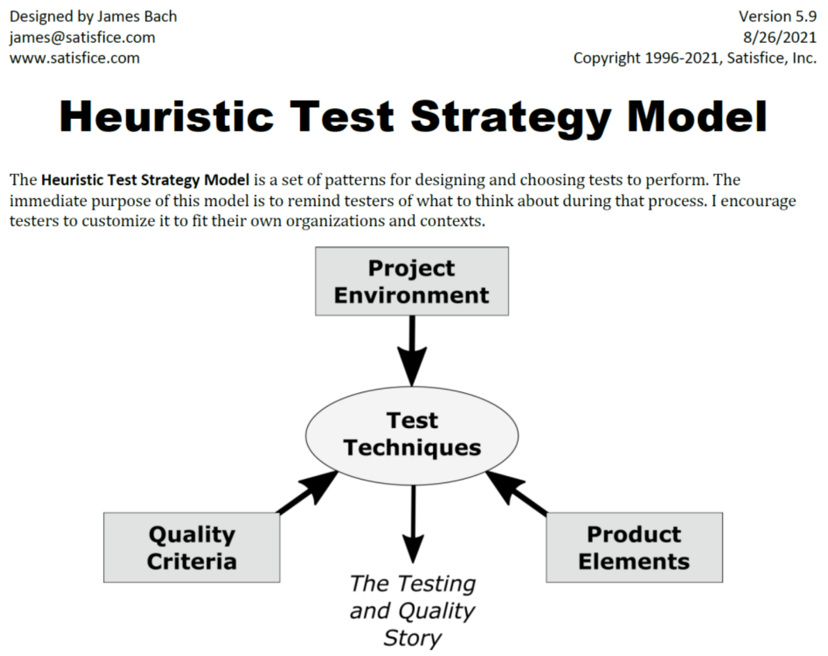
Version 5.9
8/26/2021
Copyright 1996-2021, Satisfice, Inc.

Heuristic Test Strategy Model

The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of what to think about during that process. I encourage testers to customize it to fit their own organizations and contexts.

Exploratory Guidewords. HTSM (5)

- HTSM can be used to guide testing;
- E.g.:
  - choose a guide word, e.g. *interfaces*;
    - identify “all” aspects of the program that match the guide word;
    - **one by one, what could go wrong with each?**
  - the tester can also combine guide words:
    - from different categories, e.g., Product elements: *interfaces* with Project environment: *customers*;
    - from the same category, e.g., Product elements: *interfaces* with Product elements: *data*.


Designed by James Bach
james@satisfice.com
www.satisfice.com

Version 5.9
8/26/2021
Copyright 1996-2021, Satisfice, Inc.

Heuristic Test Strategy Model

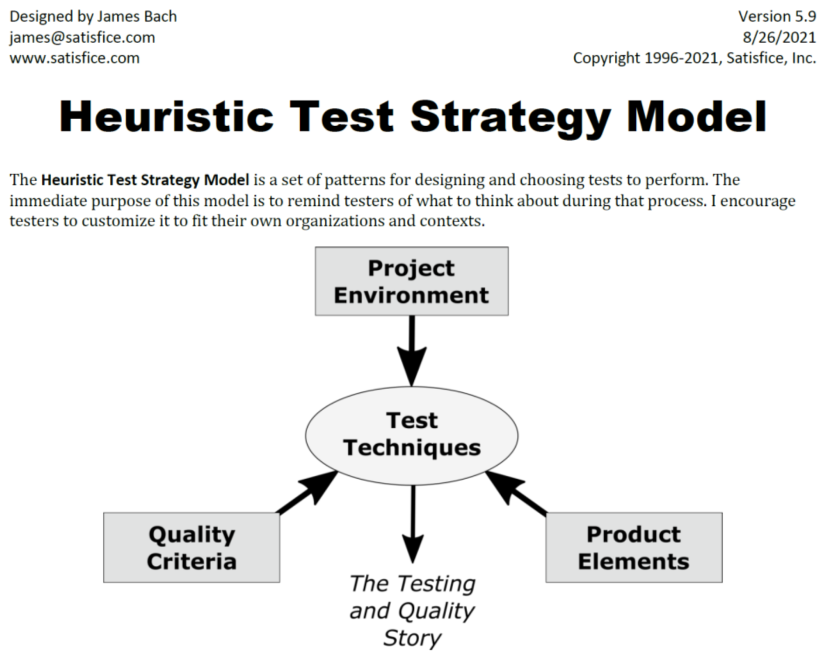
The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of what to think about during that process. I encourage testers to customize it to fit their own organizations and contexts.

Exploratory Guidewords. HTSM. Conclusions

- HTSM is a useful structure for exploratory testing;
- HTSM is not designed to apply every guideword to every element, only if it is suitable

  ==> Guidewords are heuristics;

  - *not always applicable, but useful;*
  - *reminders of ideas for analysis, not required/mandatory activities.*


Designed by James Bach
james@satisfice.com
www.satisfice.com

Version 5.9
8/26/2021
Copyright 1996-2021, Satisfice, Inc.

Heuristic Test Strategy Model

The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of what to think about during that process. I encourage testers to customize it to fit their own organizations and contexts.

Risk. Approaches

- there are several approaches on risk:
  1. Exploratory Guidewords;
  2. **Risk Catalogs**;
  3. Project-level Risks;
  4. Specific Risk-based Techniques.

Risk Catalog. Definition

- **A failure mode** is a way that the program could fail;
- one way to structure risk-based testing is with **a list of failure modes** also named:
  - **a risk catalog** or
  - **a bug taxonomy;**
- each failure mode has to be used as a test idea (something to create a test for) and create tests;
- **A risk catalog** collects
  - **ways the program has failed *in the past*,**
  - **ways the tester imagine the program could fail *in the future*.**
- **a risk catalog does not tell the tester how to test, but it describe the problem clearly enough so the tester can imagine appropriate tests;**
- E.g.: the list of 500 most common software errors [Kaner2000];

Risk Catalog. Example

- **E.g.: 1. Risk catalog for Installer Products** [Bach1999]:
  - *Wrong files installed:*
    - temporary files not cleaned up;
    - old files not cleaned up after upgrade;
    - unneeded file installed;
    - needed file not installed;
    - correct file installed in the wrong place;
  - *Files clobbered/affected:*
    - older file replaces newer file;
    - user data file clobbered during upgrade;
  - *Other apps clobbered/affected:*
    - file shared with another product is modified;
    - file belonging to another product is deleted.
- **E.g.: 2. Risk catalog for E-commerce shopping carts** [Vijayaraghavan2002];
- **E.g.: 3. Risk catalog for mobile applications** [Jha2007];
- **2.** and **3.** are based on:
  - used HTSM as a starting structure;
  - filled-in real life examples of failures from magazines, web discussions, some corporations' bug databases, interviews with people who had tested their class of products;
  - extrapolated to other potential failures;
  - extended to potential failures involving interactions among components.

Risk Catalog. Uses

- **Risk catalogs** may be used in four different ways:
  - **Generate test ideas:**
    - Find a potential defect in the list; ask whether the software under test *could have this defect*;
    - Ask how plausible it is that this bug could be in the program and how serious the failure would be if it was there;
    - If appropriate, design a test or series of *tests for bugs of this type, i.e., make the program to fail in this way.*
  - **Provide a structure for exploratory testing:**
    - as the tester learns more about how the product can fail, he can design **new tests** to explore potential failures and do **new research** (or follow new hunches) to find more new categories of ways the product can fail.
  - **Audit test plans:**
    - risk catalogs allow to assess risk coverage in test plans;
  - **Training new staff into risk-oriented thinking:**
    - expose staff to what can go wrong;
    - challenge them to design tests that could trigger those failures.

Risk. Approaches

- there are several approaches on risk:
  1. Exploratory Guidewords;
  2. Risk Catalogs;
  3. **Project-level Risks;**
  4. Specific Risk-based Techniques.

Project-level Risks

- **Project-level risks** consider
  - what might make the project as a whole to fail.

- project risk management involves
  - identifying **issues that might cause the project to fail** or **fall behind schedule** or **cost too much** or **alienate key stakeholders**;
  - **Analyzing potential costs associated with each risk**;
  - Developing **plans and actions to reduce the likelihood of the risk** or **the magnitude of the harm**;
  - **Continuous assessment or monitoring of the risks** (or the actions taken to manage them).

Project-level Risks. Heuristics (1)

- **Testers can use risks associated with the running of the project to suggest specific ideas that can guide their testing.**

- Ways to look for errors:
  1. **New things:** less likely to have revealed its bugs yet;
  2. **New technology:** same as new code, plus the risks of unanticipated problems;
  3. **Learning curve:** people make more mistakes while learning;
  4. **Changed things:** same as new things, but changes can also break old code;
  5. **Poor control:** without SCM, files can be overridden or lost;
  6. **Late change:** rushed decisions, rushed or demoralized staff lead to mistakes;
  7. **Rushed work:** some tasks or projects are chronically underfunded and all aspects of work quality suffer;
  8. **Fatigue:** tired people make mistakes;

Project-level Risks. Heuristics (2)

- Ways to look for errors:
  9. **Distributed team:** a far flung team often communicates less or less well;
  10. **Other staff issues:** alcoholic, mother died, two programmers who won't talk to each other (neither will their code)…
  11. **Surprise features:** features not carefully planned may have unanticipated effects on other features;
  12. **Third-party code:** external components may be much less well understood than local code, and much harder to get fixed;
  13. **Unbudgeted:** unbudgeted tasks may be done shoddily;
  14. **Ambiguous**: ambiguous descriptions (in specs or other docs) lead to incorrect or conflicting implementations;
  15. **Conflicting requirements:** ambiguity often hides conflict, result is loss of value for some person;
  16. **Mysterious silence**: when something interesting or important is not described or documented, it may have not been thought through, or the designer may be hiding its problems;
  17. **Unknown requirements**: requirements surface throughout development. Failure to meet a legitimate requirement is a failure of quality;

Project-level Risks. Heuristics (3)

- Ways to look for errors:
  18. **Evolving requirements**: people realize what they want as the product develops. Adhering to a start-of-the-project requirements list may meet the contract but yield a failed product;
  19. **Buggy:** anything known to have lots of problems has more;
  20. **Recent failure**: anything with a recent history of problems;
  21. **Upstream dependency**: may cause problems in the rest of the system;
  22. **Downstream dependency**: sensitive to problems in the rest of the system;
  23. **Distributed**: anything spread out in time or space, that must work as a unit;
  24. **Open-ended**: any function or data that appears unlimited;
  25. **Complex**: what's hard to understand is hard to get right;
  26. **Language-typical errors**: such as wild pointers in C;
  27. **Little system testing**: untested software will fail;
  28. **Little unit testing:** programmers normally find and fix most of their own bugs.

Project-level Risks. Heuristics (4)

- Ways to look for errors:
  29. **Previous reliance on narrow testing strategies:** can yield a many-version backlog of errors not exposed by those techniques;
  30. **Weak test tools:** if tools don't exist to help identify / isolate a class of error (e.g. wild pointers), the error is more likely to survive to testing and beyond;
  31. **Unfixable:** bugs that survived because, when they were first reported, no one knew how to fix them in the time available;
  32. **Untestable:** anything that requires slow, difficult or inefficient testing is probably under-tested;
  33. **Publicity**: anywhere failure will lead to bad publicity;
  34. **Liability**: anywhere that failure would justify a lawsuit;
  35. **Critical**: anything whose failure could cause substantial damage;
  36. **Precise**: anything that must meet its requirements exactly;

Project-level Risks. Heuristics (5)

- Ways to look for errors:
  37. **Easy to misuse**: anything that requires special care or training to use properly;
  38. **Popular**: anything that will be used a lot, or by a lot of people;
  39. **Strategic**: anything that has special importance to your business;
  40. **VIP**: anything used by particularly important people;
  41. **Visible**: anywhere failure will be obvious and upset users;
  42. **Invisible**: anywhere failure will be hidden and remain undetected until a serious failure results.

Risk. Approaches

- there are several approaches on risk:
    1. Exploratory Guidewords;
    2. Risk Catalogs;
    3. Project-level Risks;
    4. **Specific Risk-based Techniques.**

Risk-based Test Design Techniques

- **Specific Risk-based Techniques:**
  - **Quick-tests;**
  - *Part II (Lecture 06):*
    - **Boundary testing;**
    - **Constraints;**
    - **Logical expressions;**
    - **Stress testing;**
    - **Load testing;**
    - **Performance testing;**
    - **History-based testing;**
    - **Risk-based multivariable testing;**
    - **Usability testing;**
    - **Configuration / compatibility testing;**
    - **Interoperability testing;**
    - **Long sequence regression.**

Quick-Tests. Definition

- **A quick-test** is
  - an **inexpensive test**, optimized for a **common type of software error**, that **requires little time** or **product-specific preparation** or **knowledge to perform**.
- E.g.:
  1. **Boundary-value tests –** check whether a variables boundaries were misspecified;
     - the tester doesn't have to know much about the program to do this type of test.
  2. **Interference tests –** interrupt the program while it's busy;
     - the tester may try cancelling a print job or forcing an out-of-paper condition while printing a long document.
- **Testers can run quick-tests when start to test the product, *before they understand it very well*.**

Risk: Any common type of issue that requires little time to prepare and run the tests.

Quick-Tests. Details

Quick-tests are effective.

- **Every quick-test uses a theory on error.**
- E.g.: if an error that is so common that the tester is likely to see it
  - in many applications,
  - on several platforms

  ==> the tester should develop a test technique optimized for that type of error.
- **Testers perform quick-tests when they look for *typical symptoms in typical places of the software.***

Quick-Tests. Example (1)

- E.g.: **1. the shoe test is a quick-test**;
  - The tester uses auto-repeat on the keyboard as *a cheap stress test*.
    - this was one of the first tests for **input buffer overflows**;
      - if the program cannot handle all the data into one input filed, it will crash;
    - it was an effective test for a remarkably long time;
      - modern UIs are protected against this type of bug at the keyboard;
      - many web application protect them selves from excessively-long data items.

Quick-Tests. Example (2)

- E.g.: **2.** **boundary bugs**;
  - the tester needs *the variable* and *its possible values*;

  ==> quick-tests require very little information about the meaning of the variable (why the user assigns values to it, what it interacts with);
  - the boundaries gets tested because **miscoding of boundaries is a common error**.

  - intended domain: **0 < X < 100** <=> **1 ≤ X ≤ 99**;
    - common coding errors – *it is common to write inequality incorrectly*, i.e., **misclassifications errors**:
      - **0 ≤ X** (accepts 0) instead of **1 ≤ X**;
      - **X ≤ 100** (accepts 100) instead of **X ≤ 99**;
      - **1 < X** (rejects 1) instead of **0 < X**;
      - **X < 99** (rejects 99) instead of **X < 100**.
    - for this variable the boundary quick-tests values are: **0, 1, 99, and 100**.

Quick-Tests vs Code Inspection

- **simple boundary errors** could be easily exposed by **code inspection** and leave many other types of bugs get exposed by quick-tests;
  - **Question:** Why run quick-tests instead of doing more thorough code inspection?
  - **Answer:** Testers test the code that they get. (Usually, it is not their task to perform code inspection.)

- **tip:**
  - **if the tester routinely finds certain types of errors, he should design tests that are optimized to find these types of errors cheaply, quickly, and without requiring tremendous skill.**
    - there are test groups that spend a lot of time finding boundary bugs and **boundary testing becomes a main testing techniques** ==> this is a waste of **black-box tester**'s time, as **the programmer may find this bugs much more efficiently**.

Quick-Tests. List of Common Ideas

- there are seventeen common ideas for quick-tests [BBST2011]:
  1. User interface design errors;
  2. Boundaries;
  3. Overflow;
  4. Calculations and operations;
  5. Initial states;
  6. Modified values;
  7. Control flow;
  8. Sequences;
  9. Messages;
  10. Timing and race conditions;
  11. Interference tests;
  12. Error handling;
  13. Failure handling;
  14. File system;
  15. Load and stress;
  16. Configuration;
  17. Multivariable relationships.

This is not an authoritative structure for quick-tests ideas. Some of them are presented as separate test design techniques.

*Quick-Tests. Common Ideas (1)

- **User interface design errors** are detected by *touring* the user interface for things that are confusing, unappealing, time-wasting or **inconsistent with relevant design norms**;
- E.g.:
  - check for conformity to *Apple's Human Interface Guidelines;*
  - read menus, help and other on screen instructions;
  - try out the features;
  - watch the display as you move text or graphics;
  - force user errors, e.g., intentionally misinterpret instructions, do something "foolish" and see what happens.
- a new type of **tour** – ***hunt for user interface bugs***, i.e., gather information from user interface with focus on what disfavours the users (implementation errors and design annoyances).

*Quick-Tests. Common Ideas (2)

- **Boundaries** - the program expects variables to stick within a range of permissible values;
- E.g.: Provide inputs that:
  - are too big or too small;
  - are too short or too long;
  - create an **out-of-bounds calculation**;
  - combine to create **out-of-bounds output**;
    - can’t be stored;
    - can’t be displayed;
    - can’t be passed to external app.

*Quick-Tests. Common Ideas (3)

- **Overflow** or **Underflow** – values that are far too large or too small for the program to handle, i.e., process or store;
- E.g.:
  - input *empty fields* or *0*’s;
  - paste *huge string* into an input field;
  - *calculation overflows*, i.e., individual inputs are fain but an operation such as **add**, **multiply** or **string concatenation** yields a value too big for the data type, e.g. integer overflow) or a *result variable* that will store or display the result;
  - *read/write a file with too many elements,* e.g., overflow a list or an array.

- **A value can be out of bounds but not be so big that it causes an overflow.**

*Quick-Tests. Common Ideas (4)

- **Invalid calculations** and **operations;**
  - calculation involves evaluation of expressions, like 3*(-4);
  - some expressions evaluate to *impossible results*; others can't be evaluated because *the operators are invalid* for the type of data;
- E.g.:
  - enter data of the wrong type, e.g., non-numbers into a numeric field;
  - force the division by zero operation;
  - force a division by near-zero operation;
  - **arithmetic** operations on **strings**;
  - **string** operations on **numbers**;
  - arithmetic involving multiple numeric types: *if we get a result, is it the type we expect to be?*

*Quick-Tests. Common Ideas (5)

Generally, quick-tests are powerful.

- **Invalid calculations** and **operations;**
  - **most errors that create a risk of invalid operations or impossible calculations are either caught at compile time or are more easily visible to a reader of the code.**
- **Caution:** ***not any computation is a quick-test;***
  - Some people think they can check calculations primarily with quick-tests.
  - Many calculations involve several variables that all have to be set carefully to achieve a powerful test.
  - Because you have to know what you're doing, calculation tests are often **not** quick-tests.
- E.g.: **matrix inversion;**
  - the computations are subject to rounding errors, which may be huge if the programmer does not manage calculations carefully;
  - **powerful tests that assess how vulnerable program rounding are not quick-tests; they require a certain amount of time;**
- **Not every program calculations can benefit from an adequate testing using quick-tests.**

*Quick-Tests. Common Ideas (6)

- **Initial states – variable usage may indicate initial state bugs;**
  - what value does a variable hold the *first time when use it*?
  - a variable might be in one of the following states:
    - **uninitialized** – not explicitly set to any value; it holds random bits;
    - **initialized** – set to starting value, often 0; often given default value later;
    - **default** – set to a meaningful starting value;
    - **assigned** or **calculated** – intentionally set to a value appropriate to current need;
    - **carried over** – brings a previously assigned value to a new calculation that might expect the default value.
- **A variable can be in any one of these 5 states. There is a bug if the program operates on the assumption that the variable is in one of the other states.**

Quick-Tests. Common Ideas (7)

- **Initial states** - examples;
- E.g.: initial state bugs:
  - forget to initialize a variable and use it;
  - forget to reinitialize a variable and use it;
  - accidentally reinitialize a variable and use it as it would have the old value;

- **testers have to adapt the applied test technique to the testing mission;**
  - E.g.: early testing phase, having a testing objective to support programmers in designing unit tests;
    - if the tester applies *quick-tests* ==> a lot of bugs will be reported, but **the tester will fail the testing mission;**
    - if the tester applies *function testing* ==> **a set with consistent tests is obtained** and the goal is reached;

Quick-Tests. Common Ideas (8)

- **Initial states** - examples;
- E.g.: [Whittacker2002]
  - 1. a fresh copy of the program (with no saved data);
    - enter data into one dialog then do an operation (calculation or save) that uses data that was explicitly entered and data that have not been entered (you have not done any operation that would display those data). How are the unassigned data:
      - uninitialized?
      - initialized but to inappropriate values?
      - default values?
  - 2. a variable that has a reasonable value;
    - enter an impossible value and try to save it, or erase the value. What does the program insert:
      - the old value?
      - default value?
      - something else…

*Quick-Tests. Common Ideas (9)

- **Modified values – variables that have a value then it gets changed;**
  - this value change creates a risk if some other part of the program depends on this variable, i.e., design risk – **coupling** – two parts of the program depend a the same variable;
  - **high coupling ==> high risks, more side effects when changing a variable's value;**

- E.g.: a program that calculates sales tax:
  - buy something, calculate the tax, then change the person's state of residence;
  - change the location (address) of a device (make the change outside the program under test);
  - specify the parameters for a container of data, e.g. a frame that displays data, or an array that holds a number of elements; then increase the amount of data;
  - if there is auto-resize, increase and then decrease several times.

Quick-Tests. Common Ideas (10)

- **Control flow** of a program describes what it will do next;
  - **a control flow error** occurs when the program does the wrong thing, next;
- E.g.:
  - a *jump table* associates an address with each event in a list, e.g., an event might be a specific error or pressing a specific key, etc.;
  - press that key, jump to (or through the pointer in) the associated address; when the program changes state, it updates the addresses, so the same actions do new things;
  - if it updates the list incompletely or incorrectly, some new responses will be wrong;
  - **table-driven programming errors are often missed by tests focused on structural coverage.**
- if the programmers achieved a high level of structural coverage in their testing, the control flow bugs that are left are usually triggered by special **data values, interrupts or exceptions, race conditions or memory corruption.**
- **there are control flow tests that require more knowledge, a deeper investigation; these tests are not quick-tests.**

*Quick-Tests. Common Ideas (11)

- **Sequences (of tests)** are
  - various tests executed one after another without resetting the program, or the same test executed multiple times;
  - a program might pass a simple test but fail the **same test embedded in a longer sequence**.

- **sequence testing** aims to create
  - a stack overflow,
  - a memory leak,
  - a serious impact on performance (by running a test that includes lots of steps or a relatively simple test many times).

Quick-Tests. Common Ideas (12)

- **Sequences** - examples;
- E.g.:
  - **repeat the same test many times;** good candidates are:
    - anything that creates an **error message**;
    - anything that **halts a task** in the middle:
      - exception-handler may not free up memory or reset variables the program was in the middle of calculating;
    - anything that makes the program call itself, i.e., **recursion**;
      - *questions to ask:* Will this terminate? Will it exhaust system resources before terminating?
      - E.g.: adding/removing items from complex lists is handled recursively;
  - **run a suite of automated regression tests in a long randomized sequence**.

Quick-Tests. Common Ideas (13)

- **Messages** are
  - communications between programs;
  - **bad messages if they are corrupted messages;**
- E.g.: [Jorgensen2003]
  - **corrupt the connection string;**
    - some programs have a configuration file that includes a connection string to a remote resource, such as the database;
    - *questions to ask:* What if the string is a *little* wrong? Can the program gain access anyway? How will it function without access?
  - **corrupt the message and/or the response of the program A that communicates with B;**
    - A sends a message to B, expecting a response; normally, B will report the success or the failure;
    - corrupt the response so that it has elements of both (success and failure) and see which response (if either) program A believes;
    - corrupt the response so that it contains huge strings; Will this overflow a buffer or overwhelm Program A's error processing?
    - intercept the sent message and edit it; Will take down the receiving program? Will let go us the receiving program to execute commands at OS level?

*Quick-Tests. Common Ideas (14)

- **Timing and race conditions;**
  - **timing failures** can happen if the program needs access to a resource by a certain time, or must complete a task by a certain time;
  - **a race condition** is a **timing failure** if the program expects event A before B (and usually the event gets to A first), but the event gets to B first;
- E.g.:
  - When providing *input to a remote computer*, don't complete entry until just **before**, **just as**, or **just after** the application **times out** (stops listening for your input).
  - Delay input from a peripheral by making it busy, paused, or unavailable.

Quick-Tests. Common Ideas (15)

- **Interference tests** involve
  - two things that are happening in real time but *that are not perfectly coordinated*;
- they allow to do something that interfere with a task in progress that may result in:
  - **timeout** or
  - **race condition** or **loosing data** in transmission to/from an external system or device;
- E.g.: test ideas:
  - **create interrupts**;
  - **change** something the task depends on;
  - **cancel** a task in progress;
  - **pause** a task in progress;
  - **compete** for a resource needed by the task;
  - **swap** task-related code or data out of memory.

Quick-Tests. Interference Tests (1)

- **Interference tests – by creating interrupts;**
- **interrupt** is
  - a message received during an ongoing event that may affect the event handling (processing);
- E.g.:
  1. from **a device related** to the task:
     - E.g.: pull out a paper tray, perhaps one that isn't in use while the printer is printing; then switch between paper trays;
     - *questions to ask:*
       - Is the program interpreting the paper tray correctly?
       - How the program interprets a tray that is back in?
       - How does the program failure look like? (if the program fails; hard resetting may be required sometimes)
  2. from **a device unrelated** to the task:
     - E.g.: move the mouse and click while the printer is printing;
  3. from a **software event**:
     - E.g.: set another program's (or this program's) time-reminder to go off during the task under test.

Quick-Tests. Interference Tests (2)

- **Interference tests – by changing something (data, device state);**
- **change** has the meaning
  - of a **modification** (alteration) **of something the task depends on**;
- E.g.:
  - swap out a disk;
  - disconnect/reconnect with a new IP address;
  - disconnect/reconnect with a new router that uses different security settings;
  - **change the contents** of a file that this program is reading;
  - **change the printer** that the program will print to (without signaling a new driver);
  - **change the video resolution**.

Quick-Tests. Interference Tests (3)

- **Interference tests – by canceling a task (itself or another);**
- **cancel** has the meaning
  - of **aborting** (killing) **the task while it performs**;
- E.g.:
  - **cancel the studied task**
    - at different points during its completion;
    - *questions to ask:*
      - Does it cancel?
      - How much time does it require to cancel?
      - How does the program interpret the cancel? Does it corresponds to the *canceled* status? (*status mismatch* ==> program lock, lose the job that was sent, corrupt data);
  - **cancel some other task while the studied task is running**
    - a task that is in communication with the addressed task (this task is being studied);
    - a task that will eventually have to complete as a prerequisite to completion of the addressed task;
    - a task that is totally unrelated to the addressed task.

Quick-Tests. Interference Tests (4)

- **Interference tests – by pausing to create timeout;**
- **pause** has the meaning
  - of **temporary interruption in the task;**
- **timeout** has the meaning
  - of **giving up** (abandon) **the effort because too much time has passed**, while waiting (listening) for something;
- E.g.: **pause the task;**
    - for a short time;
    - for a long time (long enough for a timeout, if one will arise);
    - *questions to ask:*
      - What happens when the task is becomes unpaused, considering the possible timeout?
      - Does it cause mismatch? Does it trigger mischiefs?
  - put the printer on local;
  - **sleep the computer;**
  - put a database under use by a competing program, **lock a record so that it can’t be accessed, yet.**

Quick-Tests. Interference Tests (5)

- **Interference tests – by swapping memory;**
- **swap** has the meaning
  - of switching a process out of memory while it’s running;
- **a process** is a program (typically one that is running now) concurrently with other programs (processes).

- approaches:
  - **swap the *studied process* out of memory;**
  - **swap a *related process* out of memory while the process under test is running.**

- E.g.:
  - ***next slide…***

Quick-Tests. Interference Tests (6)

- **Interference tests – by swapping memory;**
- E.g.:
  - change focus to another application and keep loading or adding applications until the application under test is paged to disk; being inactive, change some system parameters or move some data used by the program; refocus on the program;
    - *questions to ask:*
      - Will it recognize the changes and cope with them appropriately?
  - leave it **swapped out for 10 minutes** (or whatever the timeout period is);
    - *questions to ask:* Does it come back? What is its state?
      - What is the state of processes that are supposed to interact with it?
  - leave it **swapped out much longer than the timeout period** or get it to the point where it is supposed to time out, then send a message that is supposed to be received by the swapped-out process, then time out on the time allocated for the message;
    - *questions to ask:*
      - What are the resulting state of this process and the one(s) that tried to communicate with it?

Quick-Tests. Interference Tests (7)

- **Interference tests – by competing for something required;**
- **compete** has the meaning
  - of **asking to use a specific resource**;
- E.g.:
  - **compete for a device, e.g., a printer;**
    - put device in use, then try to use it from software under test;
    - start using device, then use it from other software;
    - if there is a priority system for device access, use software that has ***higher***, ***same*** and ***lower priority access*** to the device *before* and *during* attempted use by software under test;
  - **compete for processor attention;**
    - some other process generates an interrupt, e.g., ring into the modem, or a time-alarm in the contacts manager;
    - try to do something during heavy disk access by another process;
  - **send to the studied process another job while one is underway.**

Quick-Tests. Interference Tests (8)

- **Interference tests – by competing for something required;**
- E.g.: ***(continued)***
  - **the bug deadly embrace – processes that wait forever;**
    - **Process 1** opens a *file 1* and updates it; to complete the update it needs the data from *file 2*;
    - **Process 2** opens a *file 2* and updates it; to complete the update it needs the data from *file 1*;
    - if the processes have locked the files, then **Process 1** cannot read *file 2* until **Process 2** is done with it; but **Process 2** cannot do that because it cannot read *file 1* until **Process 1** is done with it;

- **When the tester can create a competition between two processes for the same resource there is an opportunity to cause a failure;**

- other aspects to consider:
  - the timing for each request for a resource;
  - the priority the OS assigns to each process.

Quick-Tests. Common Ideas (16)

- **Error handling error** consists of
  - **errors in dealing with errors;**
- types:
  - failure to anticipate the possibility of errors and protect against them;
  - **failure to notice error conditions**;
  - failure to deal with detected errors in a reasonable way;
- **these are among the most common bugs;**
- E.g.:
  - ***next slide…***

*Quick-Tests. Common Ideas (17)

- **Error handling error** - examples;
- E.g.:
  - **make the program generate every error message;**
    - if two errors yield the same message, create both;
  - after eliciting (getting) an error message, **repeat the error several times;**
    - check for a memory leak, stack overflow, data corrupted;
    - *questions to ask:*
      - Did the program halt?
      - How gracefully does it recover? (if it recovers, **product reliability**)
      - How long does it take to recover?
  - after eliciting an error message, **keep testing;**
    - **look for side effects of the error and error handling;**
    - *questions to ask:*
      - Does the program gradually slow down?
      - Is some data saved to the disk? (if it does so, is it genuine?)

*Quick-Tests. Common Ideas (18)

- **Failure handling** consists of
  - program investigation when a bug was found, considering the actual program state;
  - **looking for related bugs;**
- E.g.:
  - **keep testing after the failure;**
    - *questions to ask:*
      - What vulnerabilities does recovery from the failure expose? E.g.: data might not be properly saved after an exception-handling exit from a task;
  - **test for related bugs *while troubleshooting a failure*;**
    - *questions to ask:*
      - What more serious or different symptoms are revealed by varying the test conditions?
  - **test for related bugs *after a bug was fixed*.**
- **looking for related bugs helps to address its maximal impact and to write a bug report in its most serious version.**

Quick-Tests. Common Ideas (19)

- **File system handling** consists of
  - program investigation related to the used files;
- read or write to files under conditions that should cause a failure;
  - *question to ask:*
    - How does the program recover from the failure? (if it does)
    - How does the program handle the error message sent by the OS?
- E.g.: read or write:
    - to a nonexistent file;
    - to a locked (read-only) file;
    - to a file that is open in another program (maybe another instance of the same program) but not locked;
    - to a file when you have insufficient privileges;
    - to a file that exceeds the maximum file size;
    - to a file that will overfill the disk (when writing) or memory (when reading);
    - to a disk that is damaged (“it has bad sectors”);
    - to a remote drive that is not connected;
    - to a drive that is disconnected during the read or write.

Quick-Tests. Common Ideas (20)

- **Load** consists of
  - the amount of data processed or tasks to perform;
- **significant background activity uses resources and adds delays ==> failures that would not show up on a quiet system;**
- E.g.:
  - test (generally) on a significantly **busy system**;
  - run **several instances of the same application** in parallel; open the same files;
  - try to get the application to do **several tasks in parallel**;
  - send the application significant amounts of input from other processes.
- **this is related to *load tests* and *stress tests*, but not the ones that require modelling (resources, operations, user profiles, scenarios, etc.);**
- **quick-tests allows to try simple things and get quickly a sense of whether there are obvious issues in the product.**

Quick-Tests. Common Ideas (21)

- **Configurations** refers to
  - **the application's compatibility with different system configurations;**
- Here, **the tester intents to *change details* of the system the program works on, not how the program works;**
- E.g.:
  - **progressively lower memory and other resources** until the product gracefully degrades or ungracefully collapses;
    - *questions to ask:*
      - Did the program halt?
      - How gracefully does it recover? (if it recovers, **product reliability**)
      - How long does it take to recover?
      - Is some data saved to the disk? (if it does so, is it genuine?)
  - **change the letter of the system hard drive**;
    - *questions to ask:*
      - Does the program require to work only with the hard drive named "C"?
  - turn on "high contrast" and other accessibility options;
  - **change localization settings**.

Quick-Tests. Common Ideas (22)

- **Multivariable relationships** refers to
  - **the collaboration between independent and dependent variables;**
- **any relationship between two variables is an opportunity for a relationship failure;**
- E.g.:
  - test with values that are individually valid but invalid together, i.e., **impossible combinations**;
    - E.g.: February 30;
  - try similar things (operations) with dissimilar objects together;
    - E.g.:
      - copy, resize or move; on graphics and text/paragraph together;
      - multiply a matrix by a number;
      - concatenate a string to an array of strings;
- **quick-tests looks for easy-to-imagine bugs and easy-to-recognize relationships between variables, but not performed in a systematic way (as combination testing does).**

Quick-Tests. Attributes

Good candy	Good quick-tests
yummy	effective
popular	representative
impressive	powerful
not very nutritious	they do not take the tester to deeper issues of the program

Quick-tests are a great way to start testing a product but not very helpful to perform deeper risk-based testing.

Quick-Tests and Exploratory Testing

- some people incorrectly characterize *exploratory testing* as if it were primarily a collection of quick-tests;
- **exploratory testing allows to learn new things about the product throughout the testing process;**
- types of techniques used in early exploratory testing:
  - **tours;**
  - **quick-tests;**
- **these techniques does not explore in depth the product as they cannot provide *knowledge* (understanding) about the product and its risks.**

Quick-Tests. Proper way to use them

- if the tester receives the product *in late development*
  - **quick-tests** are useful to start testing, as it allows **to find lots of general bugs quickly**;
    - while programmers fix these quick reported bugs, the tester has the time to understand the product deeper;
    - these bugs:
      - are based on ***generic errors***, i.e., almost any program (no matter what it does) can have these bugs;
        - are found **even if the tester does not have much knowledge** about the tested program;
        - **the tester does not perform specific bug hunting for the program** (the tester does not know where to look for them).

Quick-Tests. Conclusions

- for the **test designer**, the essence of **risk-based testing** is:
  1. imagine how the product can fail;
  2. design tests to expose these (potential) failures.

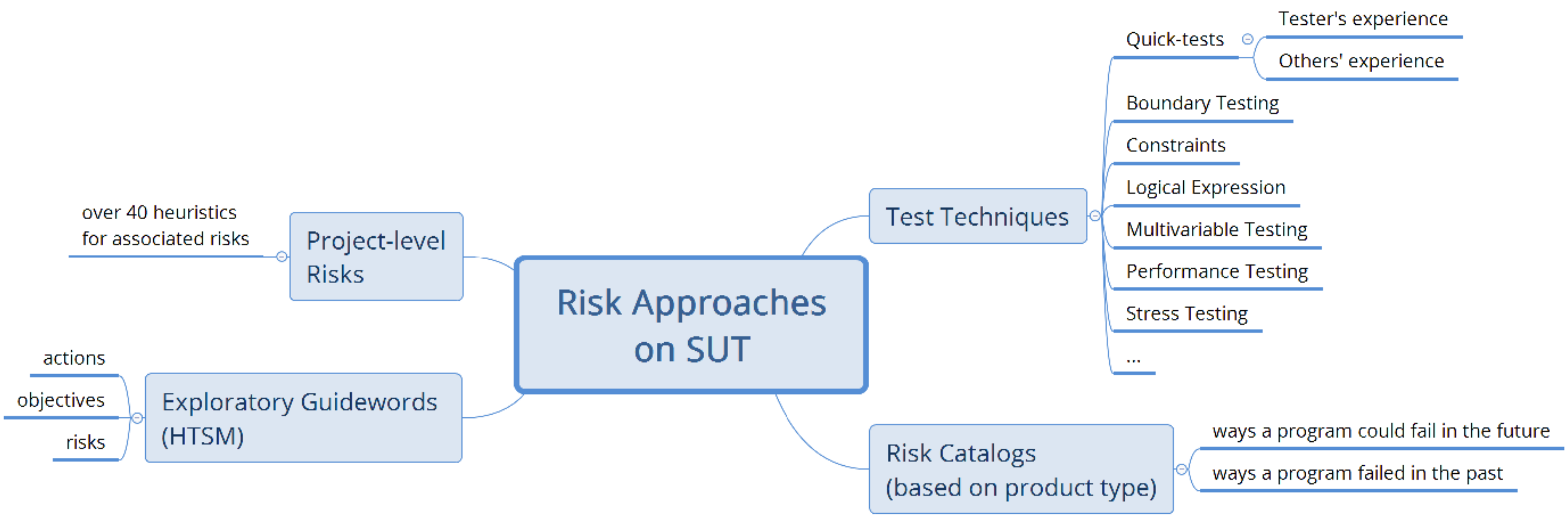
- **Quick-tests** address the tasks:
  1. the tester's experience (or the experience of others) to build a list of failures that are commonplace across many types of programs;
  2. design straightforward tests that are focused on these specific bugs.

Quick-tests rely on history, i.e., tester's experience, others' experience.

Lecture Summary

- We have discussed:
  - Risk meaning in software testing;
  - Risk approaches in software testing:
    - Exploratory Guidewords;
    - Risk Catalogs;
    - Project-level Risks;
    - Specific Risk-based Techniques:
      - Quick-tests;
      - other techniques (*next lecture*).

Wrap up on Risk Approaches



Next Lecture

- **Specific Risk-based Techniques:**
  - **Quick-tests;**
  - *Part II (lecture 06):*
    - Boundary testing;
    - Constraints;
    - Logical expressions;
    - Stress testing;
    - Load testing;
    - Performance testing;
    - History-based testing;
    - Risk-based multivariable testing;
    - Usability testing;
    - Configuration / compatibility testing;
    - Interoperability testing;
    - Long sequence regression.

Lab Activities on weeks 05-06

- Tasks to achieve in week 05-06 during Lab 03:
  - **Identify risks and test ideas based on HTSMGuidewords** for a chosen feature within a software product;
  - **Build a risk catalog** and **play the game** ***"Bug Hunt"*** to perform **quick-tests** on a authentication feature.

References

- **[JonathanKohl2006]** Jonathan Kohl, *Modeling Test Heuristics*, http://www.kohl.ca/2006/modeling-test-heuristics/, 2006.
- **[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.
- **[Whittacker2002]** Whittaker, J.A. (2002). How to Break Software. Addison Wesley.
- **[Marick2000]** Marick, B. (2000) , Testing for Programmers, http://www.exampler.com/testing-com/writings/half-day-programmer.pdf.
- **[Savoia2000]** Savoia, A. (2000), The science and art of web site load testing, International Conference on Software Testing Analysis & Review (STAR East), Orlando. https://www.stickyminds.com/presentation/art-and-science-load-testing-internet-applications
- **[McGeeKaner2004]** McGee, P. & Kaner, C. (2004), Experiments with high volume test automation, Workshop on Empirical Research in Software Testing, International Symposium on Software Testing and Analysis, http://www.kaner.com/pdfs/MentsvillePM-CK.pdf
- **[Jorgensen2003]** Jorgensen, A.A. (2003), Testing with hostile data streams, ACM SIGSOFT Software Engineering Notes, 28(2), http://cs.fit.edu/media/TechnicalReports/cs-2003-03.pdf
- **[Bach2006]** Bach, J. (2006), Heuristic Test Strategy Model, Version 5.7, https://www.satisfice.com/tools/htsm.pdf.
- **[Kaner2000]** Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000b), Bug Taxonomy (Appendix) in Testing Computer Software, Wiley, http://www.logigear.com/logi_media_dir/Documents/whitepapers/Common_Software_Errors.pdf
- **[Bach1999]** Bach, J. (1999), Heuristic risk-based testing, Software Testing & Quality Engineering, http://www.satisfice.com/articles/hrbt.pdf
- **[Vijayaraghavan2002]** Vijayaraghavan, G. (2002), A Taxonomy of E-Commerce Risks and Failures, Master’s Thesis, Department of Computer Sciences at Florida Institute of Technology, http://www.testingeducation.org/a/tecrf.pdf (chapter 7 and 8)
- **[Jha2007]** Jha, A. (2007), A Risk Catalog for Mobile Applications, Master's Thesis in Software Engineering, Department of Computer Sciences at Florida Institute of Technology, http://testingeducation.org/articles/AjayJha_Thesis.pdf (chapter 3)