LECTURE 06.

RISK-BASED TECHNIQUES. PART II

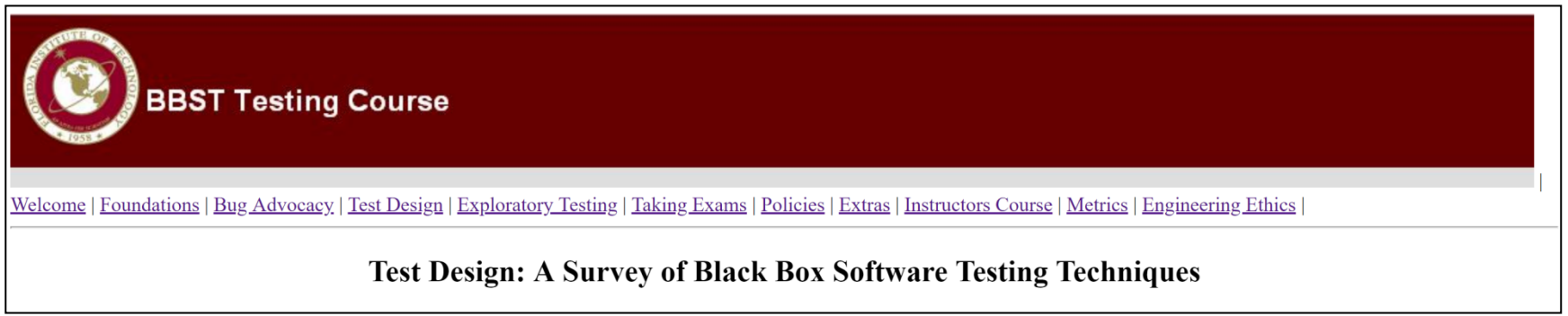
Test Design Techniques
[30 March 2022]

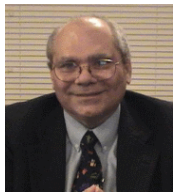
Elective Course, Spring Semester, 2021-2022

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.**

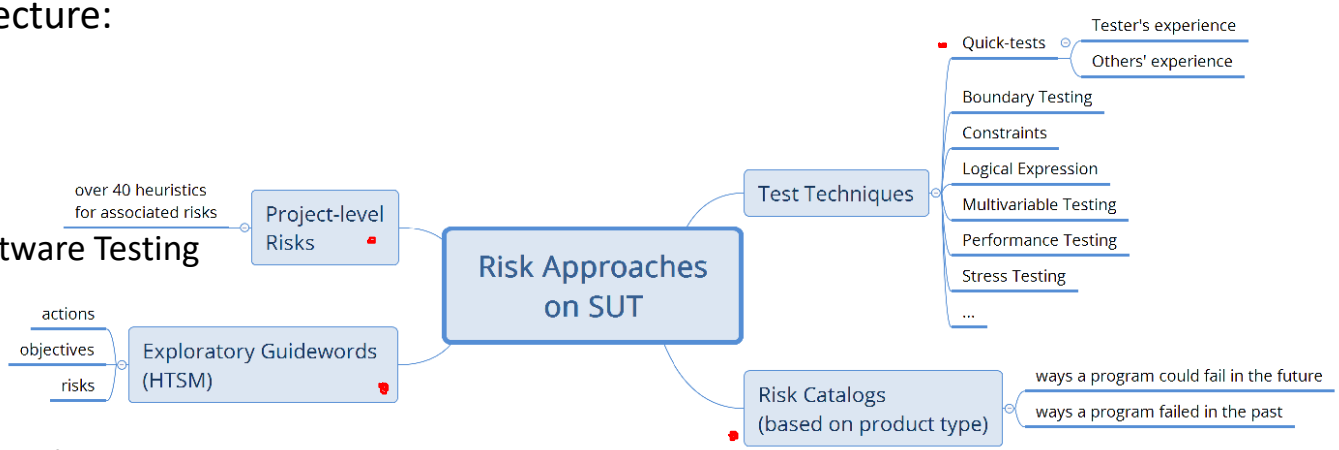
Contents

- Last lecture…
  - **Risk-based techniques (Part I)**
- **Risk-based techniques**
  - **Part II**
    - Specific Risk-based Techniques
      1. *Quick-tests;*
      2. *Boundary testing;
      3. Constraints;
      4. *Logical expressions;
      5. Stress testing;
      6. Load testing;
      7. Performance testing;
      8. History-based testing;
      9. *Multivariable testing;
      10. Usability testing;
      11. *Configuration / compatibility testing;
      12. Interoperability testing;
      13. Long sequence regression testing.

Last Lecture…

- Topics approached in last lecture:
  - **Risk-based techniques:**

    Part I
    - Risk
    - Risk Approaches to Software Testing
      - Guidewords. HTSM
      - Risk Catalogs
      - Project-level Risks
      - Specific Risk-based Techniques
        - *Quick-tests*



Risk-based Test Techniques

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

- **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 *why* it gets tested, *what risks* 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.

PART II

Quick-tests. Boundary Testing. Constraints. Logical expressions
Stress testing. Load testing. Performance testing
History-based testing.
Multivariable testing
Usability testing
Configuration / compatibility testing. Interoperability testing. Long sequence regression testing

Specific Risk-based Test Design Techniques

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

Quick-Tests. Details

- **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**.

- **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.

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

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

Boundary Testing. Risk-based Approach

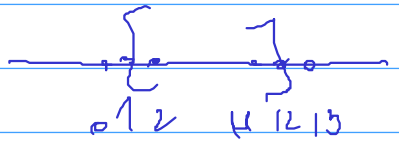
- **Boundary testing** arises out of a specific risk:
  - even if every other value in an equivalence class is treated correctly, **the boundary value might be treated incorrectly**, i.e., grouped with the wrong class;
- E.g.:
  - the programmer might *code* the classification rule incorrectly;
  - the specification might *state* the classification rule incorrectly;
  - the specifier might *misunderstand* the natural boundary in the real world;
  - the exact boundary might be *arbitrary*, but coded inconsistently in different parts of the program.

Risk: Misclassification of boundary values or mishandling equivalence classes.

BT

$m \in [1, 12]$

- Coverage based perspective



=> 6 TCs

- Risk-based p.
  - we cover the rules for the limits 1, 12 => 2 TCs

$[1, 12]$

misclassification

```
if (m>1 && m<12)
```

≥ 1 ≤ 12

$m \geq 0$

$r = 5$

BT
+ knowledge
+ time
1, 12

Quick testing
- knowledge
- time
0, 1, 12, 13

rely on the tester's ++ experience
address limits

Boundary Testing. Example

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

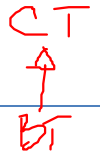
- **identified ECs:**
  - One valid EC: **EC_1: D_1 =[1, 12];**
  - Three invalid ECs: **EC_2: D_2= {month| month<1} = (-∞, 1); EC_3: D_3= {month| month>12} = (12, +∞); EC_4: D_4 = symbols/alphabet letters.**

• 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 value addressed: 1. **month = 1**; 2. **month = 12**; **misclassified by including them invalid ECs**.
- ECs addressed: **EC1: D1 = [1, 12]; mishandling valid ECs**.
- E.g.: `if (month>1) and (month<=12)…;` or

```
if (month<1) or (month<12) then throws new Exception (…);
```

Constraints Testing. Definition



- **A constraint** is a limit on what the program can handle.
- E.g.: if a program can only handle 32 (or fewer) digits in a variable, the *programmer* should provide protective routines to detect and reject any input outside of the 32-digit constraint;
- types of constraints [Whittaker2002]:

  (1) **Input constraints;** (2) **Output constraints;**

  (4) **Computation constraints;** (3) **Stored-data constraints.**
- **Constraints Testing** checks
  - how the program deals with limits;
- constraints testing generalizes the idea of **input boundaries** to all program data and activities [Whittaker2002];
- **boundary testing is an example of constraints testing**.
- **usually, constraints testing is mapped to quick-tests;**
  - for *computation constraints* and *store-data constraints* **quick-testing** is not a choice.

Risk: Misclassification of any type of constraints.

Logical Expressions. Risk-based Approach

- **A decision rule** expresses a logical relationship.
- **Logical Expression Testing** emphasizes
  - **coverage**-based approach (see **Lecture 04**) or **risk-based approach** [Marick2000];
- E.g.: a health-insurance product with the decision rule that states:
  - `if PERSON-AGE > 50 and`
    - `if PERSON-SMOKES is TRUE`
      - `then set OFFER-INSURANCE to FALSE.`
  - the testers can write this decision as a logical expressions, i.e., a formula evaluated to TRUE or FALSE;
- a risk-oriented approach to logical-expression testing *considers common mistakes* in designing/coding a series of decisions, i.e., **kinds of mistakes programmers are likely to make.**

Risk: Misclassification of decisions and short-cutting the logical expressions evaluation.

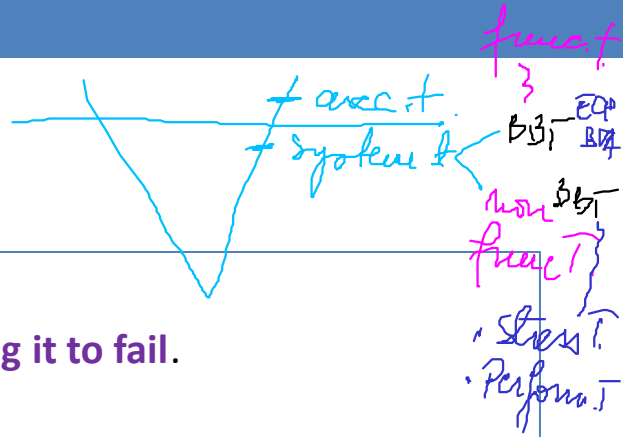
	Rule1	Rule2	----
• constrs./rules			
• person age (<70, ≥70)	<70	≥70	
• gender (M, F)	M	F	
• actions	30%	5%	

Insurance discount
- F, <70 => 10%
- F, ≥70 => 5%
- else, => 30%

Logical Exp

- Coverage-based p. $R_1, R_2, \ldots R_n$ => n TCs
- Risk-based p. • F, >5%? => check the theory on error => 2 TCs
  - F, ≥70 ?=> 5%
  - F, <70 ?=> 10%

Stress Testing. Definition



- **Stress Testing** consists of
  - tests designed and **intended to overwhelm the product, forcing it to fail**.
- E.g.:
  - intentionally subject the program to **too much input**, **too many messages**, **too many tasks**, **excessively complex calculations**, **too little memory**, **toxic data combinations**, or even **forced hardware failures**.
  - *explore* the behavior of the program *as it fails* and *just after it failed*.
- *questions to ask:*
  - How will the failures look like?
  - **What aspects of the program need hardening** to make consequences of failure less severe? (but not how to make it invulnerable)

Risk: Large amount of data, tasks to perform lack of memory, hardware failures.

Load Testing. Definition

- **Load Testing** consists of
  - tests designed **to overload the system with input or demands for work, that results in a reduced access to system resources.**
- E.g.:
  - A weak load test simply checks the number of users who can connect to a site or some equivalent count of some obvious or simple task;
  - A better load testing strategy takes into account that different users do different tasks that require different resources; on a system that can handle thousands of connections, a few users doing disk-intensive tasks might have a huge impact;
- for many programs, as load increased, there was an exponential increase in the probability that the program would fail on basic functional tasks [Savoia2000];
- *questions to ask:*
  - **Does the program handle its limitations gracefully or is it surprised by them?**
  - **Does it fail only under extreme cases?**

Risk: A user (or group of users) can **unexpectedly run out of resources** when using the software.

Performance Testing. Definition

- **Performance Testing** consists of
  - tests designed **to determine how quickly the program run** (does tasks, processes data, etc.) **under varying circumstances.**
- it can expose errors in
  - the **SUT** or
  - the **environment** SUT is running on.
- **tips:** *run a performance test today; run the same test tomorrow:*
  - if the execution times differ significantly and
  - the software was not intentionally optimized, then
  - something fundamental has changed for (apparently) no good reason.

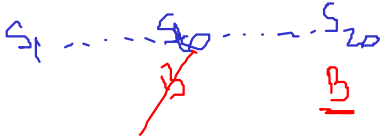
Risk: Program runs too slowly, handles some specific tasks too slowly, or changes time characteristics because of a maintenance error.

Performance Testing. Details

- software performance can be evaluated by
  - **running tests stress, load and performance tests together** or
  - running tests to understand the time-related characteristics of the program that will be experienced by most of the users most of the time;
- **Performance testing achieves quality investigation without looking for bugs.**
  - **The working systems is *studied* rather than bringing him to failure.**

History-based Testing. Definition

- **History-based Testing** allows
  - to run tests that check for errors that have happened before.
- studying the types of bugs that have occurred **in past versions of the tested product or in other similar products**;
  - what is difficult for one product in a class of products is often difficult for other products in the same class;
    - this is not regression testing, it is **history-informed exploration**.
- E.g.:
  - in a company that has a regression problem, i.e., bugs come back after being fixed, **regression tests for old bugs is a risk-based test technique**.



Risk: Old bugs can reappear, i.e., recurrent bugs.

Multivariable Testing. Risk-based Approach

- **Multivariable Testing** allows
  - to design and to run tests for various *independent* variables.
- there are three approaches on multivariable testing:
  - **mechanical:**
    - mostly discussed, e.g., **all-pair testing**; it uses an algorithm to determine what values of which variables to test together;
  - **risk-based:**
    - **techniques that select values based on a theory of error.**
  - **scenario-based:**
    - the *experienced tester* combines meaningful test values on the basis of interesting *stories* created for the combinations important to the *experienced user*.

Risk: Inappropriate interactions between variables (including configuration or system variables).

Multivariable Testing. Example

- E.g.:
  - testing the configurations, e.g., video, printer, language, memory, **based on troublesome past configurations**, e.g., technical support complaints;
  - picking values of variables to use together in a calculation **to maximize the opportunity for an overflow or a significant rounding error**.
  - a program might handle large values for one variable well, but fail when **several variables values are maximized within the same tes**t.

Testing Compatibility with SUT

- **The software environment** for a SUT is the computer or network that it runs on;

- the environmental requirements might be:
  - **narrow:** only *a specific* operating system, *a specific* printer, *at least* a specified amount of memory, works only with *a specified* version of some program, etc.;
  - **very flexible**: does not have many constraints.

Configuration/Compatibility Testing. Definition

- **Configuration testing** allows
  - to determine what environments the software will *correctly* work with or are compatible with;

- **compatibility testing requires limits, as there are too many possible configurations to test;**
- **risk-based compatibility approach** focuses on
  - **testing configurations that are more likely to cause the program to fail;**

- **tip:**
  - pick specific devices, specific test parameters **that have a history of causing trouble**.

Risk: Incompatibility with hardware, software, or the system environment.

Interoperability Testing. Definition

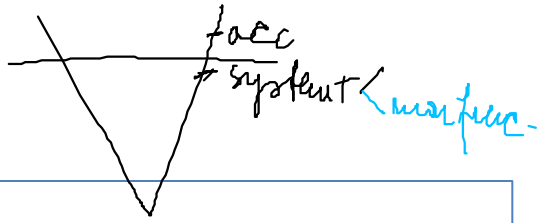
- **Interoperability Testing** checks
  - how the product interoperates (interacts or works with) others (programs, devices, external systems);
- *simple interoperability testing* is like function testing:
  - the tester tries the two together to see if they behave well together;
- *deep interoperability testing:*
  - the tester designs tests that focus specifically on ways in which **he suspects the software might not work correctly** with the other program, device or system.
- **tip:**
  - design tests starting from a list of common problems.

Risk: Incompatibility with other programs, devices, or external systems.

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**;
- these test technique names are sometimes used:
  - interchangeably;
  - in the opposite way they were defined here.

Usability Testing. Definition



- **Usability Testing** is
  - testing with the intent to demonstrate that some aspect of the software is **unusable for some members of the intended user community;**
  - **focused on the risk that the program is too hard to use.**
- E.g.:
  - too hard to learn;
  - too hard to use;
  - makes user errors too likely, i.e., it does not emphasize good practices;
  - wastes user's time.

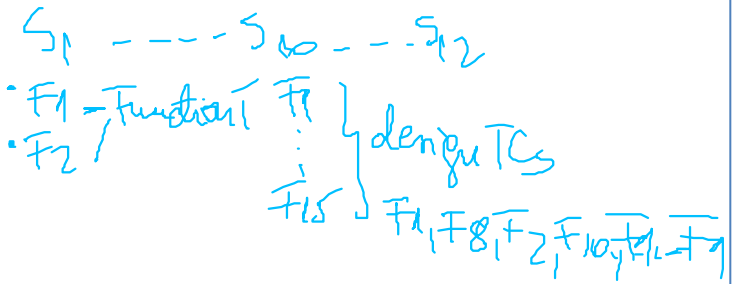
Risk: Software is unusable for some members of the intended user community, e.g., too hard to learn or use, too slow, annoying, triggers user errors, etc.).

Usability Testing vs User Testing

- **Usability testing**
  - is performed by usability testers **who might or might not be end users.**
- **User testing**
  - is performed by users, **who may or may not focus on the usability of the software.**

Long-Sequence Regression Testing. Definition

- **Long Sequence Regression Testing** means
  - testing the same build of the same software with the same tests (already passed) but **run many times in a random order, without resetting the software before running the next test**.
- long-sequence regression can expose bugs that are otherwise hard to find, such as **intermittent-seeming failures** [McGeeKaner2004] as:
  - memory leaks,
  - race conditions,
  - wild pointers and
  - other corruption of working memory or the stack.



Risk: Long sequence tests hunt bugs that won't show up in traditional testing (run tests one at a time and clean up after each test) and are hard to detect with source code analysis.

Long-Sequence Regression Testing. Example

- E.g.: *memory leaks exposed by running long-sequence tests:*
  - a function runs once with no problem; it runs twice with no problem;
  - each time the function is run, the function uses the space (memory) but does not free it;
  - **after a few hundreds uses later, the program runs out of memory;**
  - this type of bug cannot be found by running a simple test that executes the code once and eventually, resetting the tested software;

- **Long-sequence testing** is different than **quick-tests sequences** by the complexity and the time required **to set up long sequences**.

Risk-based Testing. Conclusions

- **there are many different tests the tester may design and run, but he does not have the time to run all of them;**
  - therefore, a subset of them is picked to run;
  - this subset should be **small enough** and have **good tests** that **find bugs that should be found** and **to expose other quality related information** that should be exposed;

1000 tests built on *function testing* ≠ 1000 tests built on *risk-based approach*

- **the testing strategy drives the choice of test techniques to use; the strategy depends on the information objectives (testing goals) and the context;**
  - **information objective = what the tester intends to do;**
  - **context = what is possible and how it can be done;**

- **Guidewords and risk catalogs are learning aids, they help the tester imagine ways the program can fail.**

Lecture Summary

- We have discussed:
  - Risk meaning in software testing;
  - Risk approaches in software testing:
    - Guidewords;
    - Risk Catalogs;
    - Project-level risks;
    - Specific risk-based techniques
      - 13 techniques that address functional and non-functional risks.

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.

Next Lecture

- **Lecture 07:**
  - **Invited IT Company: Evozon**
  - **Presentation topic: Load Testing**
  - **Date: Wednesday, 07 April 2021**
  - **Hours: 08:00-10:00**
  - **Online: Skype (the link will be provided over MS Teams)**

Lecture 08

- Activity-based techniques:
  - Focus. Objectives;
  - Specific Techniques:
    - Guerilla Testing;
    - Random Testing;
    - Use Case Testing;
    - Scenario-based testing;
    - etc.;
  - **Core** and **Desired** Test Attributes.

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)