LECTURE 08.
ACTIVITY-BASED TECHNIQUES

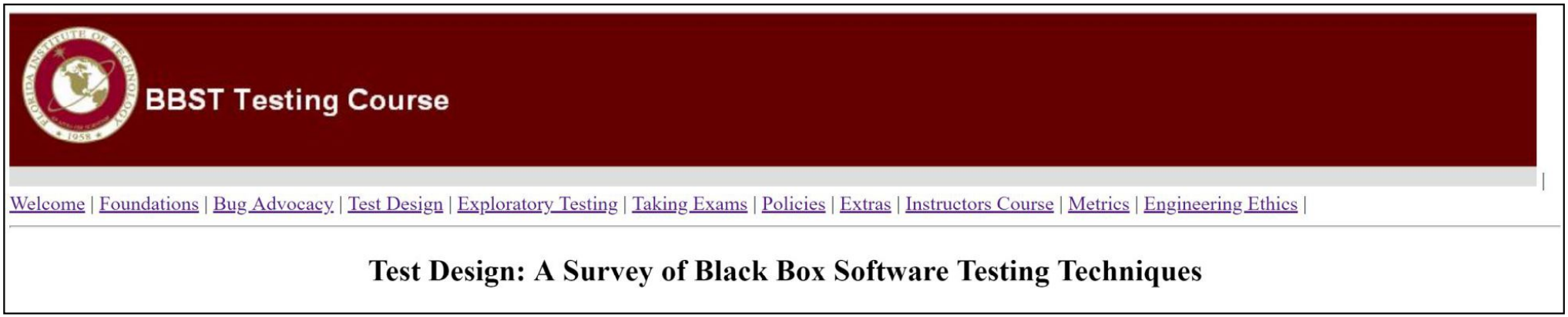
Test Design Techniques
[13 April 2022]

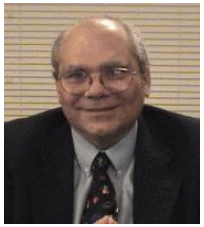
Elective Course, Spring Semester, 2021-2022

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.





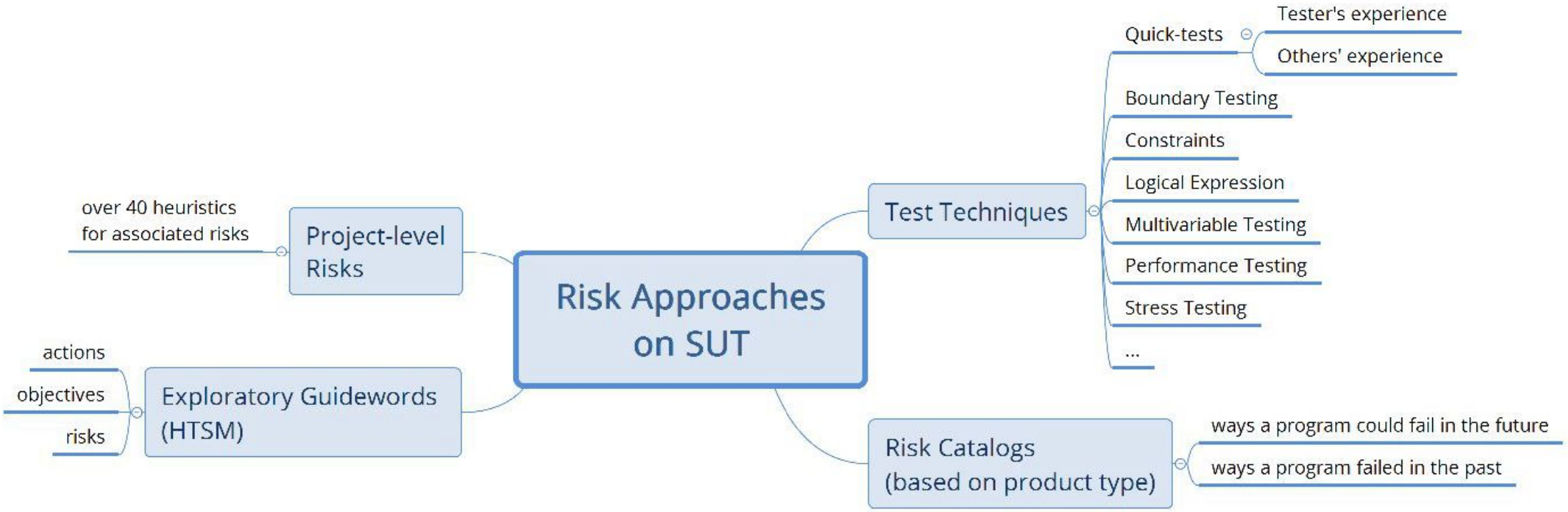
The BBST Courses are created and developed by **Cem Kaner, J.D., Ph.D.., Professor of Software Engineering at Florida Institute of Technology.**

Contents

- Last lecture…
- Terminology
- Activity-based Techniques
    - Use Case Testing;
    - Scenario-based testing;
    - Other activity-based techniques
        - Guerilla testing;
        - All-pairs testing;
        - Random testing;
        - Installation testing;
        - Regression testing;
        - Long sequence testing;
        - Dumb monkey testing;
        - Load testing;
        - Performance testing.

Last Lecture...

- Topics approached in Lecture 05 and Lecture 06:
  - **Risk dimension in testing:**



TDTs Taxonomy

- The main test design techniques are:
  - **Black-box approach:**
    - **Coverage-based techniques;**
    - **Tester-based techniques;**
    - **Risk-based techniques;**
    - **Activity-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**	**•Maintainable**	**•Supports troubleshooting**
•Valid	**•Non-redundant**	**•Information value**	**•Appropriately complex**
•Value	**•Motivating**	**•Coverage**	**•Accountable**
•Credible	**•Performable**	**•Easy to evaluate**	**•Affordable**
	•Reusable		**•Opportunity Cost**

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

Activity-based Test Techniques

- **An activity-based technique considers**
  - **the actions (activities) that need to be performed by the tester in order to design and run tests.**

- most of the activity-based techniques are classified in another way as well – almost each test design techniques requires to address some activities at some degree.

Activity-based Test Techniques. Focus

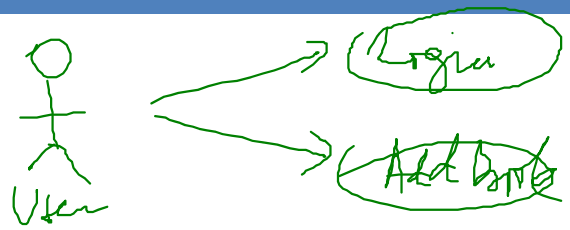
Activity-based techniques focus on *how-to* test/perform testing. This is why these techniques are most closely match the classical notion of a *technique*.

- a technique may be classified depending on what the tester has in mind when he uses it.
- E.g.: **long-sequence (automation) testing**
  - is **activity-oriented** because every time the tester thinks about the activities to perform:
    - programming, maintenance and developing diagnostics;
    - types of work required to create and run the tests;
  - is **risk-oriented** because the tests are especially suited to hunt for specific types of bugs that will not show up in a normal testing.

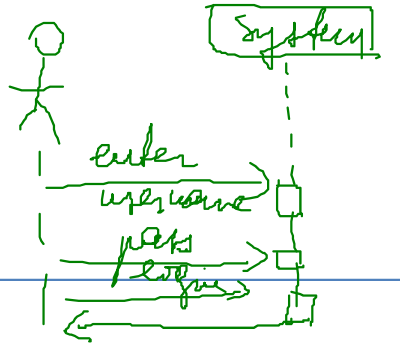
Activity-based Test Design Techniques

- **Activity-based techniques [11 techniques]:**
  - Use case testing;
  - Scenario testing
  - Guerilla testing;
  - All-pairs testing;
  - Random testing;
  - Installation testing;
  - Regression testing;
  - Long sequence testing;
  - Dumb monkey testing;
  - Load testing;
  - Performance testing.

Use Cases. Definition



- **A Use Case** specifies
  - a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor [Jacobson1995];
  - a system's behavior in response to a request from *an actor* which might be a human or another system;
  - the intended behavior, i.e., **how the system should work** to achieve *a goal*, **but not the motivation of the actor or the consequences for the actor if the request fails**;
  - the actor's steps and system behavior on *a sequence diagram*;
    - "happy path/flow" is the sequence diagram that shows the simplest set of steps, i.e., *sequence of actions*, that lead to **success**;
    - other paths show complications, some leading to **failures**;



Use Cases. Details

- Concepts used with use cases [Jacobson1995]:
  - **An actor** is
    - a person, process or external system that interacts with your product;
  - **A goal** is
    - the reach of a desired state of the system, i.e., the observable result of value;
  - **An action** is
    - a change of state and is realized by sending a message to an object or modifying a value in an attribute;
    - something the actor does as part of the effort to achieve the goal;
  - **Sequences of actions** are
    - a specific flow of events through the system;
    - many different flows are possible and many of them may be very similar;
    - to make a use-case model understandable, similar flows of events are grouped into a single use case;
  - **A sequence diagram** is
    - a diagram that shows actions and states of a use case, emphasizing the ordering of the actions in time.

Use Case Testing. Definition

- **Use-Case Testing** consists of
  - **the modeling and testing down the sequence *diagram's paths;***

- the required activities are achieved in several steps;
  - ***next slide…***

Activity: The tester creates sequence diagrams (behavior models) and runs tests that trace down the paths of the diagrams.

Use Case Testing. Proper Use Steps

- A working pattern to describe a full set of use cases [Cockburn2001]:
  1. brainstorm and list *the primary actors*;
  2. brainstorm and exhaustively *list user goals* for the system;
  3. capture the summary goals (higher-level goals, which include several sub-goals);
     - these capture the meaningful benefits offered by the system;
  4. select one *use case* to expand;
     - capture stakeholders and interests, preconditions and guarantees;
     - write the main success scenario, i.e., a sequence diagram;
     - brainstorm and exhaustively *list extension conditions:*
       - alternate **sequence diagrams to achieve the same result**, or
       - **sequences diagrams that lead to failure.**
  5. repeat **step 4.** for each distinct use case identified.

Use Case Testing. Benefits

- Use-case testing encourages the tester:
  - to identify the actors in the system: **human** and/or **other processes or systems**;
  - to inventory the possible actor goals;
  - to identify the benefits of the system by identifying the summary goals;
  - to develop some method, e.g., sequence diagrams, outlines, textual descriptions, for describing a sequence of actions and system responses that ultimately lead to a result;
  - to develop variations of a basic sequence, to create meaningful new tests.

- **the tester goes beyond features or individual specification-claims ==> identifies meaningful sequences;**
- **the use case contains its own oracle:**
  - if the sequence should lead to achievement of some goal, **but it does not actually achieve it ==> the program is broken;**
  - if the sequence that should lead to error handling, **but it does not or the error is not handled well ==> there is a failure.**

Use Case Testing. Downsides

- Use-case testing brings some drawbacks:
  - the approach abstracts out the human element, i.e., **does not consider the human as a relevant factor**;
    - because the actor may not be human, actors are described in ways that **are equally suitable for things that have no consciousness**;
    - **human goals go beyond a desired program state**; they are more complex;
      - for humans:
        - **goals are intimately connected with motivation:**
          - *Why does this person want to achieve this goal? How important is it to them? Why?*
        - **failure to achieve a goal causes consequences, including emotions:**
          - *How upset will the user be if this fails? Why?*
- **understanding the human element might be irrelevant for sequence diagrams**, but it proves to be valuable:
  - **to prioritize the tests;**
  - **to combine goals in human-meaningful ways;**
  - **to interpret and to explain the results.**

Use Case Testing vs Tours and Function Testing

- **a use case-based approach to testing provides a good starting point (like tours) for testers if they don't know much about the application;**
- advantages of using use case testing:
  - provides a **structure for tracing** through the application by **building diagram sequences**;
  - **simple as function testing but it uses several functions together.**

Use Case Testing and Scenarios. RUP Approach

- **The Rational Unified Process (RUP) defines scenarios in terms of use cases [Collard1999]:**
  - **a scenario is an instantiation of a use case, i.e., it specify the values of the use case data to create one instance of the use case;**
  - a RUP-scenario traces one of the paths through the use case;
  - if the tester actually executes the path, he runs a scenario test;
  - thorough use case-based testing involves
    - tracing through **all** (most) of **the paths**
    - through **all** (most) of **the use cases**, paying special attention to failure cases.
- **RUP Approach: scenario= instance of a test case;**
- **This approach on scenario is not applied in this course.**
  - ***use-case tests* are not considered to be similar to *scenarios*.**

RUP Approach. Scenario Analysis

- **there can be scenarios with no people in them, but when there are people, scenario writers are interested in them;**
  - even if all the obvious actors are human, there is a person who has started the scenario in motion;
    - **the scenario analyst will be on the lookout for this *human-in-the-background* and will be interested in the motivation and reactions of that person;**
- **more generally, scenarios involve a much richer view of the system and the people who use it, including details that use case authors would normally exclude.**

Even though use case-based testing is useful in its own right, *as a basic approach to scenario testing in RUP*, it misses the deep value of what we know as *scenario analysis*.

Scenario. Definition

- **A scenario** is
  - a *hypothetical story* about how someone uses a program;

- **the elements of the story** [Carroll1999] are:
  - **setting**;
  - **agents** / **actors**;
  - **goals** / **objectives**;
  - **motivations** and **emotions**;
  - **plot** - sequences of actions and events;

- **actions** and **events** can change the goals.

Scenario-based Testing. Definition

- **A scenario-based testing** allows
  - to design and run tests which is based on a ***scenarios***;

- it checks the stories presented by scenarios;
- **it relies on *good* scenarios.**

Activity: Creating a story (or a related-family of stories) and a test that expresses it.

Scenario-based Testing. Attributes

A scenario is a coherent story, credible, motivating, complex, and easy to evaluate.

- **Ideal scenario test** has several *characteristics*:
  1. the test is based on **a coherent story** about how the program is used, including *goals* and *emotions* of the involved people.
  2. **the story is credible**; stakeholders will believe that something like **it probably will happen**;
  3. **the story is motivating**; a stakeholder with influence will advocate for fixing a program that failed this test;
  4. **the story involves complexity**: a complex use of the program or a complex environment or a complex set of data;
  5. **test results are easy to evaluate**; this is important for scenarios because they are *complex*.

Scenario-based Testing. Benefits (1)

- **scenario-based thinking** was popular in the 1950s in *military planning*;
- later it was adopted and proved useful to *commercial field* by **imagining scenario crisis.**
- **Benefits of using scenarios** [Kahn1967]:
  - **call attention to the larger range of possibilities** that must be considered by the analysis in the future;
  - **dramatize** and **illustrate the possibilities**;
  - **force analysts to deal with details and dynamics** that they might avoid if they focus on abstract considerations;
  - **illuminate interactions** of psychological, social, economic, cultural, political, and military factors, including the influence of individual personalities, in a form that permits the comprehension **of many interacting elements at once**;
  - **consider alternative possible outcomes of certain real past and present events**.

Scenarios help imagine complexity (people, society) and help work with it (how entities interact between them). This complexity misses from sequence diagrams, i.e., working with use cases.

Scenario-based Testing. Benefits (2)

- many test techniques tell the tester how the program will behave in the first few days that someone uses it;
- Scenario-based testing ensures that good scenarios tests
  - **go beyond the simple uses of the program** to ask whether the program is delivering the benefits it should deliver;
  - often give the tester **insight into frustrations that an experienced user** will face – someone who has used the program for a few months and is now trying to do significant work with the program.

Scenario-based Testing. Combination Testing Usage

- there are **three approaches to combination testing**:
  - ***Mechanical (or procedural):***
    - the tester uses a **routine procedure** to determine a good set of tests;
    - E.g.: **random combinations** and **all-pairs**;
  - ***Risk-based:***
    - the tester combines test values (the values of each variable) based on perceived **risks** associated with noteworthy combinations;
    - E.g.: **quick-tests** and **stress testing**;
  - ***Scenario-based:***
    - **the experienced tester** combines meaningful test values on the basis of **interesting stories** created for the **combinations important to the experienced user**;
    - E.g.: **scenario-based testing**.

Scenario-based thinking provides a strategy for selecting meaningful combinations, such as combinations important to the experienced user.

Scenario-based Testing. Types

- there are **17 types of types of scenarios**, i.e., **lines of inquiry**, useful to develop *suites of scenarios*:

1. **Create follow-up tests;**
2. **List possible users;**
3. **Work alongside users;**
4. **Interview users;**
5. **Look at specific transactions;**
6. **Work with sequences;**
7. **Consider disfavored users;**
8. **Work with forms used by the user;**
9. **Write life stories for objects;**
10. **List system events;**
11. **List special events;**
12. **List benefits and create and to end tasks;**
13. **Work with competing systems, read books/papers on them;**
14. **Study complaints about the systems and competitor systems;**
15. **Create mock business;**
16. **Try to convert real-life data from competitor or predecessor systems;**
17. **Look at the output of competitor systems.**

- these ideas are about the user, the program, the task, or the market;
- **they are the basis for *suites of scenarios* rather than *one scenario focused on one bug*.**

Scenario Types (1)

1. **Create follow-up tests for bugs that look controversial or deferrable;**

2. **List possible users – analyze their interests and objectives;**
   - **interests:** broader motivators of the person;
   - **objectives:** specific tasks the user wants to achieve with the program;
   - E.g.: for a program that helps people find a job, the types of users would be:
     - *the traditionalist user;*
     - *the young networker;*
     - *the socialite salesperson;*
     - *the support group.*

Scenario Types (2)

3. **Work alongside users to see how they work and what they do;**
   - *questions to ask:*
     - What are they doing? Why?
     - What confuses them?
     - What irritates them?
   - all of above will become tests;

4. **Interview users about famous challenges and failures of the old system;**
   - study users or workflows.

Scenario Types (3)

5. **Look at the specific transactions that people try to complete;**
   - E.g.:
     - opening a bank account;
     - sending a message.
   - the tester can design scenarios (one, or probably more) for each transaction, plus scenarios for larger tasks that are composed of several transactions;
   - in a transaction processing system *a transaction is an indivisible operation*, i.e., **the system completes it or cancels it.**

Scenario Types (4)

6. **Work with sequences;**
   - people (or the system) typically do tasks (like Task X) in an order;
     - What are the most common orders (sequences) of subtasks in achieving X?
   - it might be useful to map Task X with a *behavior diagram*;
   - **this is the closest analog to the use-case based scenario;**
     - **a use-case test is transformed into a scenario by adding the human details to it;**
       - **goal and alternate sequences are added;**
       - **motivation and consequences should be considered as well.**

Scenario Types (5)

7. **Consider disfavored users;**
   - *question to ask:* How do they want to abuse your system?
   - *disfavored users* are humans too;
     - the tester should analyze their interests, objectives, capabilities, and potential opportunities;
     - they are the source of lots of scenarios;
   - E.g.: *hacking into the web application*;
     - **the systems is successful if it blocks bad actions of disfavored users, rather than enabling them.**

Scenario Types (6)

8. **What forms do the users work with?**
   - work with these forms by performing operations, e.g., read, write, modify, etc.;
   - E.g.: for a program that helps people find a job, some forms to fill in would be:
     - several standard résumé templates;
     - automatically filling in fields in employer-site or recruiter-site forms;
   - **any form may be the source of various scenarios.**

Scenario Types (7)

9. **Write life histories for objects in the system.**
   - *questions to ask:*
     - How was the object created, what happens to it, how is it used or modified, what does it interact with, when is it destroyed or discarded?
   - similar to creating a list of possible users and base the scenarios on who they are and what they do with the system, the tester can create a list of objects and base his scenarios on **what they are, why they are used**, and **what the system can do with them**;
   - E.g.: for a program that helps people find a job, the list of objects for which life history can be built would be:
     - Resumes;
     - Contacts;
     - Downloaded ads;
     - Links to network sites;
     - Emails.

Scenario Types (8)

10. **List system events;**
    - *questions to ask:* How does the system handle them?
    - **an system event** is
        - **any occurrence** (all things that can happen) that the system is designed to respond to;
        - anything that causes an **interrupt** and the system has to respond to;

    - E.g.: for a program that helps people find a job, some business events would be:
        - going to an interview;
        - sending a résumé;
        - getting called by a prospective employer.

Scenario Types (9)

11. **List special events;**
    - **A special event** is
        - **a system event that is handled in a special way than usual, considering the contract;**
        - **a thing that do not happen very often, but might cause the system to work differently when it happens;**
    - **the system might change how it works or do special processing in the context of a special event;**
    - **they are predictable but unusual occurrences;**
    - E.g.:
        - last (first) day of the quarter or of the fiscal or calendar year;
        - while installing or upgrading the software;
        - holidays.

Scenario Types (10)

12. **List benefits and create end-to-end tasks to check them.**
    - *questions to ask:*
        - What benefits is the system supposed to provide?
    - the tester should not rely only on an official list of benefits;
        - he should **ask stakeholders what they think the benefits of the system are supposed to be**;
    - look for **misunderstandings and conflicts** among the stakeholders;
        - **same system offers different benefits to different people;**
    - E.g.: the current project is an upgrade:
        - *What benefits will the **upgrade** bring?*

Scenario Types (11)

13. **Work with competing systems, or read books / articles about what systems like this are supposed to do;**
    - E.g.: for a program that helps people find a job, some questions to ask would be:
        - What programs compete with the tested system? How do they work?
        - If the tester knows the product well, what would he expect of the tested system?
        - What programs offer some of the capabilities of the tested system?
            - if the tester knew contact management programs well, what expectations would he have of the tested system's contact management features?
    - **anything that attracts the tester's interest in the competitor program becomes scenario for the tested system.**

Scenario Types (12)

14. **Study complaints about this system's predecessor or competitors;**
    - software vendors usually create a database of customer complaints;
    - companies that write software for their own use often have an in-house help desk that keeps records of user problems;
    - E.g.:
        - look for complaints about the tested product (earlier versions) or similar ones online, e.g., customers forums, help pages;
        - read the complaints; the tester should take the user errors seriously:
            - **they reflect ways that the users expected the system to work, or things they expected the system to do.**

Scenario Types (13)

15. **Create mock objects (businesses) and treat them as real and process their data;**
    - E.g.: for a mock business the tester should:
        - choose the characteristics of the business well:
            - simulate a business that fits the **profile of the intended users**;
        - create events that are realistic for that business:
            - see **how the tested system copes with these events**;
        - when he runs into problems or limitations:
            - consider (and describe) **how they impact the simulated business**.

Scenario Types (14)

16. **Try converting real-life data from a competing or predecessor application;**
    - E.g.: for a program that helps people find a job, design tests by feeding it with user files from previous versions
        - *questions to ask:*
            - Does it remember all the contacts?
            - Does it look up the right receipts and do the right calculations for job-hunting tax deductions?
            - How can the tester be sure of it? (oracles should be used)
                - How much work does it require to evaluate the results?
                - without an evaluation strategy the tester will recognize the most obvious failures only;
                - **important bugs will be missed**, e.g., output may be wrong but formatted correctly.

Scenario Types (15)

17. **Look at the output that competing applications can create;**
    - *questions to ask:*
        - What reports does it print?
        - How would the tester create these reports / displays / export files in the tested application?
            - How much work does it take to these reports?
            - What kinds of mistakes does it make?

Test Suite Scenario. Definition

- **A Test Suite** is
  - a collection of several related tests;
- the inquiry list (17 scenario ideas) previously presented **should not** be applied all together in order to develop suites;
  - it is wiser to design a collection of scenarios by following **one line of inquiry at a time instead of combining them;**
  - each presented approach can be used to generate many interesting tests;
  - the tester should develop his skills by working with them individually; he should combine them later, when he has more experience;
- E.g.: the list of the types of objects in the system (so the tester can develop a set of possible life histories for each of them);
  - *given an item in the list,* the tester should *ask scenario-building questions, i.e., coherent story, credible, complex, motivating, easy to evaluate;*
  - the tester should do this for several scenarios;
    - **the tester can build several scenarios for each item (type of object) in the list.**

Test Suite Scenario. Attributes (1)

1. **Create a Coherent Story**

- given an item in the list, the tester should ask the scenario questions:
  - **How to create a story that people will listen to?**
    - **setting**;
    - **agents** / **actors**;
    - **goals** / **objectives**;
    - **motivations** and **emotions**;
    - **plot** – sequences of actions and events;
    - **actions** and **events** can change the goals.

- **The expected result of the story is the result the tester expects if the program is working correctly.**
  - **it has to run from start to finish in a way that makes sense;**
    - *an arbitrary sequence of actions is not a scenario*, as there is no *motivation*, no *goal*.

Test Suite Scenario. Attributes (2)

2. **Create a Credible Story**

- ask the scenario questions:
  - **What would make a story about this be credible?**
    - When would this come up, or be used?
    - Who would use it?
    - What would they be trying to achieve?
    - Competitor examples?
    - Spec/support/history examples?

- **Developing a credible story means that people who read the story should believe the program will run into a situation like this. Even if does not happen very often, but it will happen.**

Test Suite Scenario. Attributes (3)

3. **Create a Motivating Story**

- given an item in the list, ask scenario-building questions:
  - **What is important (motivating) about this?**
    - Why do people care about it?, Who would care about it?
    - What does it relate to that modifies its importance?
    - What gets impacted if it fails?, What does failure look like?
    - What are the consequences of failure?

- **The story is motivating if someone important (specific stakeholder) thinks the program should pass the test.**
  - **scenarios are powerful tools for building a case that a bug should be fixed;**
    - the tester should make the problem report *meaningful to a powerful stakeholder who should care about this particular failure;*
  - inability to develop a **strong** scenario around a failure may be a signal that the failure is not well understood or not important.

Test Suite Scenario. Attributes (4)

4. **Create a Complex Story**

- given an item in the list, ask the scenario questions:
  - **How to increase complexity?**
    - What does this naturally combine with?
    - What benefits involve this and what collection of things would be required to achieve each?
    - Can the tester make it bigger? Do it more? Work with richer data? (What boundaries are involved?)
    - Will any effects of the scenario persist, affecting later behavior of the program?
- **meaningful complexity of the story/scenario** consists of using:
  - **many related features and actions;**
  - **many data values.**

Test Suite Scenario. Attributes (5)

4. **Create a Complex Story – Handling Complexity in Scenarios**

- Each feature is tested in isolation (or in small mechanical clusters of features) before testing it inside scenarios;
  - it allows to reach **straightforward failures sooner and more cheaply;**
  - **tests reused expose better weak designs** with cheap function tests than more expensive scenarios;
  - **combination failures are harder to troubleshoot;** simple failures that appear first inside a combination can be unnecessarily expensive to troubleshoot;
  - **scenarios are prone to blocking bugs: a broken feature blocks running the rest of the test;** once that feature is fixed, the next broken feature blocks the test.
- **Adding complexity to a scenario arbitrarily will not work;** the story must still be coherent and credible;
  - *arbitrary combinations* is reasonable for combination tests, not for scenarios.

Test Suite Scenario. Attributes (6)

5. **Create an Easy-to-evaluate Test**

- given an item in the list, ask the scenario questions:
  - **How to design an easy-to-evaluate test?**
    - self-verifying data sets?
    - automatable partial oracles?
    - known, predicted result?
- **Evaluability is important because so many failures have been exposed by a good scenario but missed by the tester.**
  - E.g.: summarized IBM data showed that over 30% of the bugs discovered has been actually exposed by tests run by testers, but the bugs were not noticed it took to much time and attention to check the results closely enough to realize there was a bug;
- **When designing complex tests, i.e., scenario-based testing, it is important to design them such that the failures are obvious.**

Scenario-based Testing. Good Practices

- **sketch the story**, briefly; the tester does not have to write down the details of the setting and motivation if he understands them;
  - the good story need to be developed in tester's head, first;
  - **the full story is written down when if the bug report is written;**
- some skilled scenario testers add detail early;
- only write down the steps that are essential, i.e., the tester might forget them or the tester is likely to make a mistake;
- **the expected result is always the correct program behavior.**

late

S_1 - - - S_{10}

F_1 - - F_{15} F_2, F_4

scenario

F_1 - - F_{10}

SC1: F_1, F_2, F_4, F_8, F_1 - - -

SC2: F_3, F_8, F_3, F_8 - - -

coverage ↓ not in focus

Complexity ↓ high

Scenario-based Testing and Testing Coverage

- in general, **scenario-based testing cannot guarantee high code coverage.**
- **each line of inquiry is like a *tour***
  - the tester could explore that line thoroughly to achieve a level of coverage;
  - E.g.:
    - system events;
    - objects created by the system;
    - required benefits;
    - features;
- **however, coverage-oriented testing often uses simpler tests;**
- **some old research results suggests that traditional black-box testing achieves less than 33% coverage of the lines of the program ==> testing misses 2/3 parts of the code, mostly error handling.**

Scenario-based testing is not coverage focused.

Scenario-based Testing and Regression Testing

- documenting and *reusing* scenarios seems efficient because it takes work to create a good scenario;
  - **scenarios often expose design errors but the test teaches the tester about the design mostly;**
    - **the power of scenario-based testing is to expose design flaws or disagreements;**
  - **scenarios expose coding errors because they combine many features and much data;**
    - to cover more combinations, the tester needs new tests, not repetition of old ones ==> **no information value is added by running again the same test;**
    - **it cost time and money to maintain automated tests that runs the scenario and collects the program's responses;**
  - **it might be more effective to do regression testing using single-feature tests or unit tests, not scenarios.**

Regression testing based on scenario tests might be less powerful and less efficient than regression testing based on other techniques, e.g., function testing.

Scenario-based Testing vs Risk-based Testing (1)

- **Scenario testing**
- Tests are **complex** and **coherent stories** that capture how the program will be used in real-life situations.
- These are combination tests, whose combinations are credible reflections of real use.
- These tests are highly **credible** (stakeholders will believe users will do these things) and so failures are likely to be fixed.

- **Risk-based testing**
- Tests are derived from ideas about how the program could fail.
- These tests might focus on individual variables or combinations.
- These tests are designed for **power** and **efficiency** – find the bug quickly – rather than for credibility. Extreme-value tests that go beyond reasonable use are common.

When a bug is found with risk-based testing, it is recommended to do follow-up testing to describe a scenario that demonstrates the bug in a more credible and motivating way to investigate, i.e., bug advocacy.

Scenario-based Testing vs Risk-based Testing (2)

Attribute	Scenario-based Testing	Risk-based Testing
Credible	++ *Core*	+ *Desired*
Efficiency	*Desired*	*Core*
Motivating	*Core*	*Desired*
Powerful	*Desired*	*Core*
Reuse	*Desired*	*Desired*

- **Core = Essential** = required, necessary, mandatory, compulsory, must have;
- **Desired** = not necessary, not mandatory, not compulsory but recommended, good to have.

Scenario-based Testing vs Other Test Techniques

- **scenario-based thinking** is just one technique among many discussed;
- the tester may apply as well:
  - **function testing, domain testing, specification testing;**
  - **quick-tests;**
  - **use cases, scenarios;**
- *the tester should not:*
  - use scenarios for everything;
  - combine all other tests into scenarios;
- ***the tester should:***
  - **use each test on its own;**
  - **use scenarios to reach tasks or benefits or situations that are more complex than he can reach by using simpler tests.**

Test Design. Early Testing vs Late Testing

- **Designing for early testing – simple tests:**
  - E.g.: test techniques:
    - **Function testing;**
    - **Domain testing;**
    - **Use-case testing;**
    - **Simple combination testing;**
    - **Quick-tests;**
- **Designing for late testing – complex tests:**
  - E.g.: test techniques:
    - **Complex combination testing;**
    - **Meaningful scenarios;**
    - **Data-intense** (or otherwise complex-to-test) **risk based-testing.**

Scenario-based Testing. Conclusions

- **A scenario is**
  - **coherent, credible,**
  - **motivating, complex,**
  - **easy to evaluate story;**

- **in scenario-based testing the 17 lines of inquiries (scenario types) represent distinct test techniques;**

- many test techniques tell the tester how the program will behave in the first few days that someone uses it;
- **Scenario-based testing teach testers** that:
  - good scenario tests go beyond the simple uses of the program to ask **whether the program is delivering the benefits it should deliver**;
  - good scenarios often give tester **insight into frustrations that an experienced user will face**—someone who has used the program for a few months and is now trying to do significant work with the program.

OTHER ACTIVITY-BASED TECHNIQUES

- Guerilla testing, All-pairs testing, Random testing, Installation testing Regression testing, Long sequence testing, Dumb monkey testing Load testing, Performance testing

Guerilla Testing. Definition

- **Guerilla Testing** allows
  - to run exploratory tests that are usually **time-boxed and done by *an experienced explorer***;
- **goal:** to perform a **fast and vicious** attack on some part of the program.

- E.g.: a senior tester might spend a day testing an area that is seen as low priority and would otherwise be ignored;
  - he tries out his most powerful tests;
    - **if he finds significant problems, the area will be re-budgeted** and the overall test plan might be affected;
    - **if he finds no significant problems, the area will hereinafter be ignored or only lightly tested.**

Activity: Time-boxed, risk-based testing focused on one part of the program.

All-Pairs Testing. Definition

- **all-pairs criterion** specifies
  - that **one value** for each investigated item **is paired with every value of other investigated items;**
  - **a coverage criterion**;
- **All-Pairs Testing** is
  - the testing performed to achieve **all-pairs criterion**;
- E.g.: the tester intends to test ***N*** variables together;
  - he picks a few values to test from each variable;
  - under the all-pairs coverage criterion, the tests must include one value for each variable and, **across the set of tests**, every value of each variable is paired with every value of every other variable;
  - usually, the tester relies on **a tool** for picking the combinations of values for the variables.

Activity: following the algorithms (or using tools) to generate tests that meet this criterion.

Random Testing. Definition

- **Random Testing** allows
  - the tester to use a random number generator to determine:
    - **values to be assigned to some variables**, or
    - **the order in which tests will be run**, or
    - **the selection of features to be included in a test**.

- it means **the decisions are made by a random number generator** rather than by a human as part of a detailed plan, but testing is performed by tester.

Activity: Drive test decisions with a random number generator.

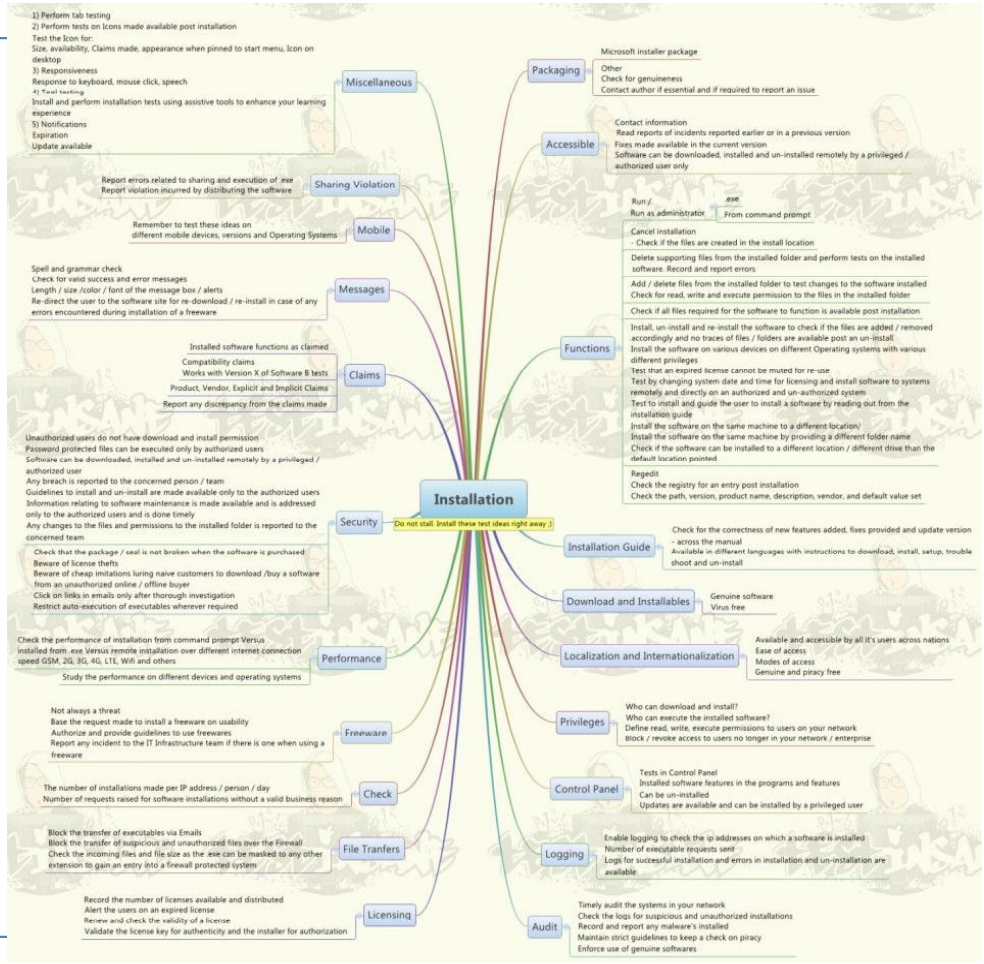
Installation Testing. Definition

- **Installation Testing** allows
  - to perform checks whether the product (part of product) can be **installed, uninstalled, reinstalled or updated**;
- **installation is often one of the least well-tested parts of a program, and therefore a good place to hunt bugs;**
- *approaches:* **risk-based** [Bach1999], procedural, **activity-based** [Agruss2000];
- E.g.: checks on whether:
  - a new product, a new version of the product, or a patch **installs well**, without interfering with other software;
  - a virgin **installation or re-installation works**;
  - a **uninstallation works** and then if the **reinstallation after uninstallation is possible** (or impossible if it the Digital Rights Management (DRM) forbids it);

Activity: How to automate much of the installation-test process.

Installation Testing. Test Ideas

- some test ideas for installation testing are [TestInsameApps2019]:
  - **Functions;**
  - **Claims;**
  - **Security;**
  - **Performance;**
  - **Installation guide;**
  - **Download and Installables;**
  - **Localization and Internationalization;**
  - **Privileges;**
  - **Check;**
  - **Control Panel;**
  - **File Transfer;**
  - **Freeware;**
  - **…**



Regression Testing. Definition

- **Regression Testing** allows
  - **to run/reuse the same tests, i.e., old tests, after a change occurred**;
- **goal:** to detect possible **side effects** of applied changes;
- **specific types of regression** based on type of the performed changes:
  - **bug fix regression** is **to prove a bug fix was ineffective**;
  - **old bugs regression** is **to prove a software change caused a fixed bug to become unfixed**;
  - **side-effect regression (stability regression)** is **to prove a change has caused something that used to work but now is broken**.
- most discussions of regression testing as a technique consider:
  - **how to automate the tests**, or
  - **how to use tools to select a subset of the tests that might be the most interesting for the current build**.
- particularity: ***write and fix and fix and fix and fix and fix and fix the "automation" code to do the same test over and over.***

Activity: Do the same boring tests over and over or, write once and fix if required.

Regression Testing. Details

- characteristics of a good regression test suite:
  - the most worthwhile test are selected (from the old tests);
  - redundant tests are eliminated;
  - tests are recoded to be maintainable and efficiently reusable.
- **unit regression testing** vs **system regression testing**:
  - **unit testing:** creating automated regression tests at this level is part of usual, **efficient configuration management process;**
  - **system regression testing:** regression testing at this level is **less valuable, as it is expensive and unlikely to reveal new information;**
- some project teams consider large regression test suites because old bugs keep reappearing, recurrent bugs;
  - this indicates **a *broken development process* – it cannot be fixed by regression testing, especially at system level.**

Long Sequence Testing. Definition

- **Long Sequence Testing (LST)** allows
  - **to hunt for bugs, typically intermittent failures, that are hard to expose by running *one test a time techniques;***
- it is performed overnight or for several days;
- it is known as:
  - **duration testing**, or
  - **life testing**, or
  - **reliability testing**, or
  - **soak testing**, or
  - **endurance testing**;
- **goals:** discover *bugs that short sequence tests miss*.

Long Sequence Testing. Details

- E.g.:
  - long-sequence regression testing;
  - long sequences of tests that have never been run previously;
- Types of bugs revealed:
  - **bugs missed by short sequence tests**;
  - basic **functional bugs** that are unnoticed when they occur but **show up as violations of a precondition for a later test**;
  - bugs related to: **stack overflows, wild pointers, memory leaks, and bad interactions among several features**.
- designers of LSTs have **to write a framework for running all these tests**, **randomizing their order or parameters and building support for troubleshooting failures;**
  - otherwise, **it is hard to see what went wrong first and then led to the failure the tester can see** ==> good diagnosis is required, thorough logs and good logs analyzer.

Activity: Creating software to execute LSTs, with diagnostics to help troubleshoot the failures they trigger, i.e., developing support technology.

Dumb Monkey Testing. Definition

- **Dumb Monkey Testing** is
  - **similar to *state model-based testing* except that the test execution program does not work from a verifiable state model;**
- it drives the program from state to state by **simulating random user input**;
  - the testing continues until the program crashes or fails in some obvious way;

Activity: Random state-transition tests programmed to run until they crash.

Dumb Monkey Testing. Details

- **Dumb Monkey Testing vs State Model-based Transition Testing**
  - when a program is in any given state, it will ignore some inputs (or other events) and respond to others;
    - state transition consists of the program's response to some inputs or events, i.e., takes the system to its next state;
    - **if the tester feeds random inputs to the program to force state transitions** ==> **monkey testing** [Nyman1998];
    - if the tester has
      - a state model that ties inputs to transitions, and
        - *an oracle* (the ability to tell whether the program transitioned to the correct state)

          then the tester can do state-model-based testing ==> **smart monkey testing** [Nyman1998];
        - *no oracle* (no state model is available)

          then the tester can still run the monkey testing, waiting to see if the program crashes or corrupts data in some obvious way ==> **dumb monkey testing** [Nyman1998].

Performance Testing. Definition

- **Performance Testing** is
  - **a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload;**
- it is typically done [Meier2007]:
  - to help **identify bottlenecks in a system**;
  - to **establish a baseline for future testing**;
  - to **support a performance tuning effort**;
  - to **determine compliance with performance goals and requirements**;
  - to **collect performance-related data** to help stakeholders make informed decisions related to the overall quality of the application being tested.
- performance, stress and load testing are
  - *risk-based techniques* – if the focus is on **ways to make the product to fail**;
  - *activity-based techniques* – if the focus is on **the technology they use**, i.e., **how they are performed.**

Activity: Coding and then executing input streams activities, followed by execution-timing monitoring activities.

Lecture Summary

- We have discussed:
  - **Activity-based testing.** Focus
    - Use case testing;
    - Scenario testing;
    - Other activity-based techniques:
      - Guerilla testing;
      - All-pairs testing;
      - Random testing;
      - Installation testing;
      - Regression testing;
      - Long sequence testing;
      - Dumb monkey testing;
      - Performance testing.

Next Lecture

- Bug reporting
  - Bug Types;
    - Implementation
    - Design
  - RIMGEN/RIMGEA investigation and reporting strategy;
  - Examples of bug reports from BBST Bug Advocacy;
  - Bugs taxonomy by Claudiu Draghia.

References I

- **[Kaner2003]** Cem Kaner, An introduction to scenario testing, http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf, 2003.
- **[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 .
- **[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
- **[Bach1999]** Bach, J. (1999), Heuristic risk-based testing, Software Testing & Quality Engineering, http://www.satisfice.com/articles/hrbt.pdf

References II

- **[Agruss2000]** Agruss, C. (2000), Software installation testing: How to automate tests for smooth system installation, Software Testing & Quality Engineering, 2 (4), http://www.stickyminds.com/getfile.asp?ot=XML&id=5001&fn=Smzr1XDD1806filelistfilename1%2Epdf
- **[TestInsaneApps]** Test Insame Apps, 2019, http://apps.testinsane.com/mindmaps
- **[Nyman1998]** Nyman, N. (1998), Application testing with dumb monkeys, International Conference on Software Testing Analysis & Review (STAR West);
- **[Meier2007]** Meier, J.D., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007), Performance Testing Guidance for Web Applications. Redmond: Microsoft Press.
- **[Jacobson1995]** Jacobson, I. (1995), The use-case construct in object-oriented software engineering, In John Carroll (ed.) (1995), Scenario-Based Design. Wiley.
- **[Cockburn2001]** Cockburn, A.(2001). Writing Effective Use Cases Addison-Wesley.
- **[Collard1999]** Collard, R. (July/August 1999). Test design: Developing test cases from use cases, Software Testing & Quality Engineering, 31-36.
- **[Kahn1967]** Kahn, H. (1967), The use of scenarios, in Kahn, Herman & Wiener, Anthony (1967), The Year 2000: A Framework for Speculation on the Next Thirty-Three Years, pp. 262-264. https://www.hudson.org/research/2214-the-use-of-scenarios
- **[Carroll1999]** Carroll, J.M. (1999), Five reasons for scenario-based design, Proceedings of the 32nd Hawaii International Conference on System Sciences, http://www.massey.ac.nz/~hryu/157.757/Scenario.pdf