**Today**

- System testing

**Homework**

- HW 11 (sprint plan due)
- Prototype demo sign up
Independent System Testing

"Testing conducted on a complete, integrated system to evaluate the system’s compliance with its specified requirements."


System testing ...

- Testing the system as a whole
- As the user would use it (as opposed to the developer)

Exercise 1: Come up with some system tests for your project
System Testing: Exploratory Testing versus Test Plan

Test Plans

- Pre-defined set of test cases
- Try to be exhaustive
- Typically based on requirements
- Boundary values, base cases, edge cases, etc.
  - base case = under normal use (case on a “happy path”)
  - edge case = outside normal use, limits of what system can handle (but not nec. unreasonable)
  - corner case = many things go wrong, riducously out of range of what is expected
- For system tests, test all requirements and various combinations
- Follow test plan during testing

Exploratory Testing

- Similar in terms of trying to exhaustively test system
- However, don’t restrict to “rigidly” following test plan
- Instead, let intuition drive testing
- Assumes you learn more about the system as you go
  - Which following a test plan can hinder (since just following a set plan)
- Follow “odd” behavior (that aren’t necessarily bugs)
Combinatorial, Pairwise, and $n$-Way Testing

Combinatorial Testing

- **Challenge:** exhaustive testing over multiple combinations of features, cases, parameters
  - if $n$ features, then at least $2^n$ combinations
  - becomes very expensive quickly!

Example: in a text editor BOLD, ITALIC, UNDERLINE text  
[from: B. Laboon]

- Can represent combinations as a **truth table** (feature on/off)
- Where T is true and F is false

<table>
<thead>
<tr>
<th>BOLD</th>
<th>ITALIC</th>
<th>UNDERLINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

- Instead of “truth tables” these are called **covering arrays**
Pairwise Testing

- Instead of testing all possible combinations, just test pairs
  - also called “2-way interaction testing”, or “all pairs” testing

- Can significantly reduce number of tests to run (i.e., reduce truth table)

Q: What test cases do we need to run for the editory example?

<table>
<thead>
<tr>
<th>BOLD</th>
<th>ITALIC</th>
<th>UNDERLINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

- Similar approach can be used if you have more than two possible outcomes of an item
  - e.g., for SIZE could be normal, sub-heading, heading

Exercise 2 & 3: Come up with example pairwise tests for your project
$n$-Way Testing

- Similar idea for testing more than just two features at a time

- E.g., 3 out of 4 (or more) possible features at once
  - In example could add STRIKETHROUGH

- Underlying idea though is that bugs found with smaller numbers of $n$
  - NIST “interaction rule” studies\(^1\)
    - showed that most bugs caused by 1 or 2 parameters
    - and progressively fewer with 3 or 4 parameters
    - where no bugs were found with over 6 parameters

\(^1\)See https://csrc.nist.gov/publications/detail/itl-bulletin/2016/05/combinatorial-testing-for-cybersecurity-and-reliability/final
Exploring Different “Paths” through the System

A path is a sequence of user interactions

- for example, search for item, add to cart, checkout

Bugs often found when not looking at just the “Happy Paths”

- Could be normal, often used sequences, but with extreme or odd inputs
- Could be unexpected or non-expected sequences

Exercise 3 & 4: What are “happy” and “unhappy” paths for your system?
Issues in System Testing

Testers aren't finding the important bugs
- i.e., important to the customer
- usability problems are valid bugs!
- so are security, performance, etc.

Narrow functional tests don’t always find critical bugs
- i.e., isolated tests of a single feature
- users will use features together
- often means different sequences of operations
- e.g., open, edit, print, edit, print (fails)

While focusing on narrow functional tests ...
- Not testing installation
- Not testing documentation
- Not focusing on risky parts of the system
- Checking system does what it is supposed to ...
  - but not what it is not supposed to
- Not noticing and exploring odd (“funny”) behavior
- Waiting to stress test until it is too late
For Thursday ...

- Bring a computer with the latest version of your application
- Make sure it is runnable
- Bring a short description (list) of the features of your application
- Mark which of the features are currently supported in your application

Demo sign up ...

- Will send email with details and sign up times
- Will have 15 minute slot to demo and describe quality and engineering practices