Test Driven Development

Building a fortress in a greenfield (or fortifying an existing one)

Dr. Hale
University of Nebraska at Omaha

Today's topics:

Software Testing and Test driven development

- Unit / integration / acceptance testing
- Think-test-build-test-repeat
- QUnit Demo

Blackbox and Whitebox testing

Vulnerability surface and testing strategies

Agile scrum/sprint setup/project management tools

Getting started:

- Team meetings – Define user stories and use cases
  – Plan sprint

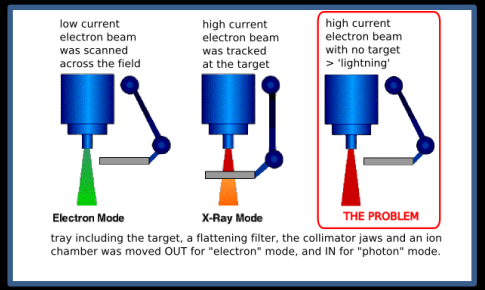
Test-driven Development

Some Material from Bernd Bruegge and Allen Dutoit Object-Oriented SE: Using UML, Patterns, and Java (because their slides are hilarious)

Famous Problems

- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software

- The Therac-25 accidents (1985-1987), one of the most serious non-military computer-related failure in terms of human life (at least five died)
  - Reason: Bad event handling in the GUI
- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem.



Terminology

- Failure: Any deviation of the observed behavior from the specified behavior
- Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure
- Fault: The mechanical or algorithmic cause of an error (“bug”)
- Validation/testing: Activity of checking for deviations between the observed behavior of a system and its specification.

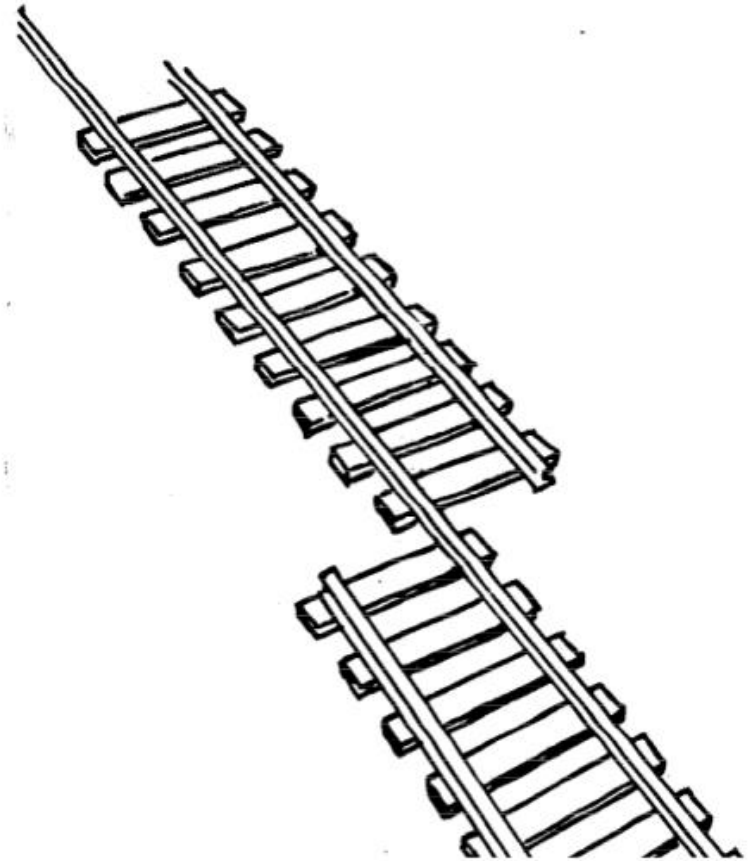
What is this?

A failure?

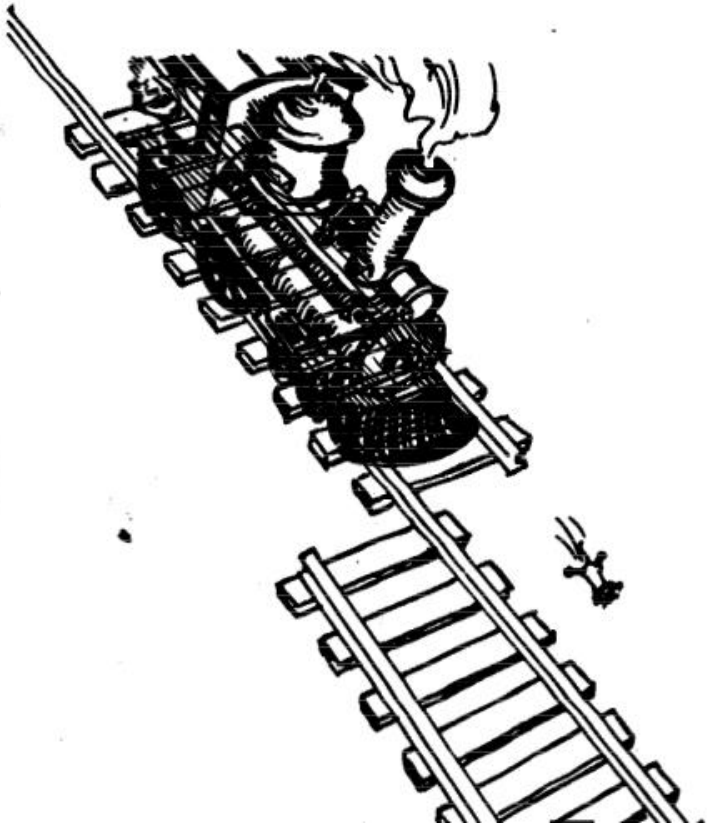
An error?

A fault?

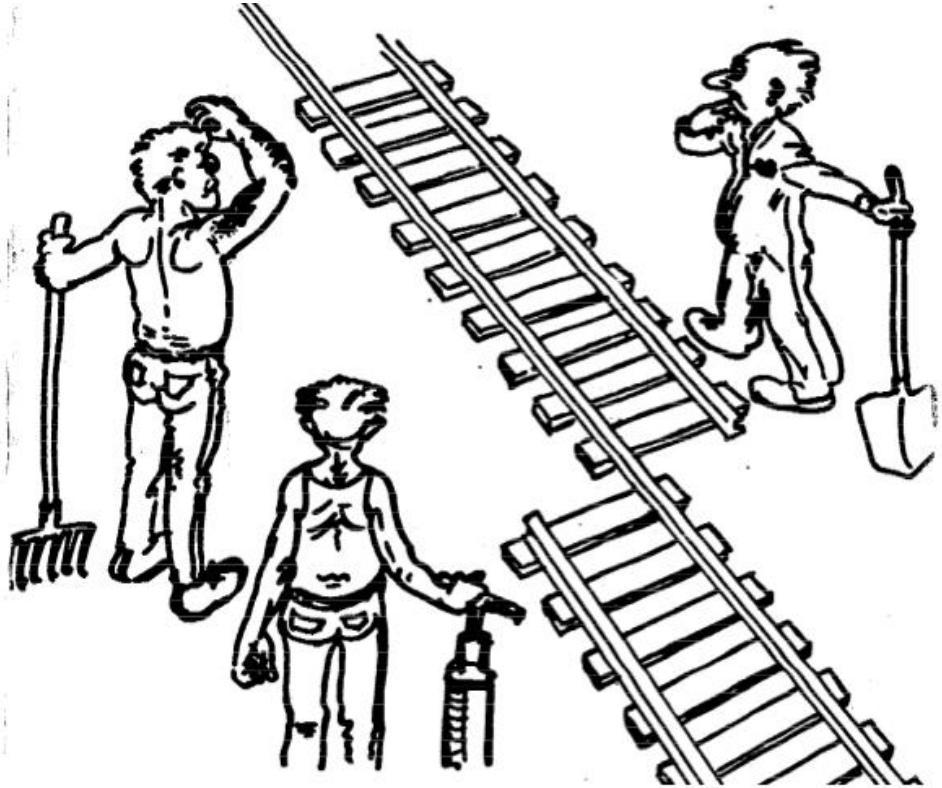
We need to describe specified and desired behavior first!



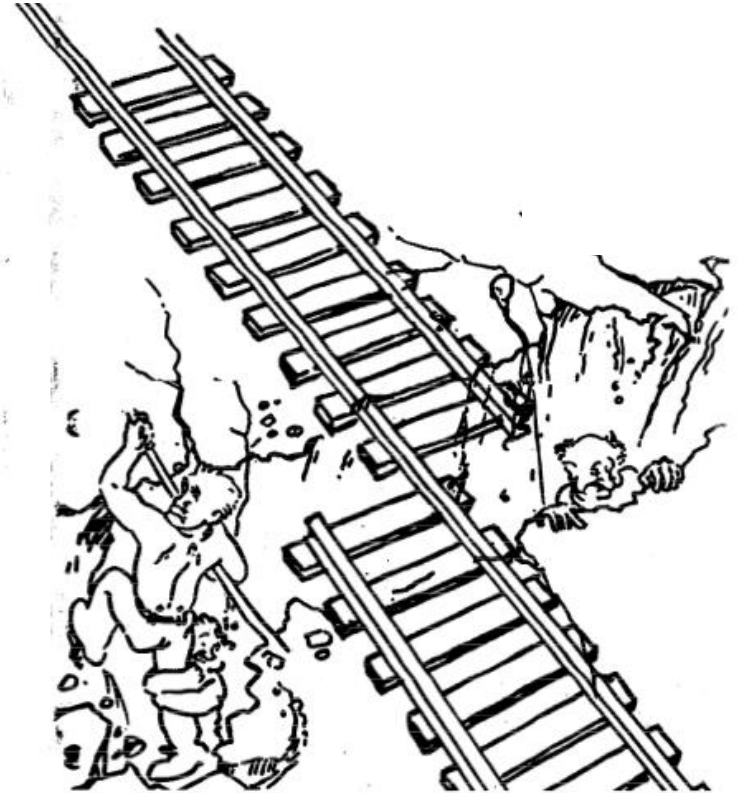
Erroneous State ("Error")



Algorithmic Fault



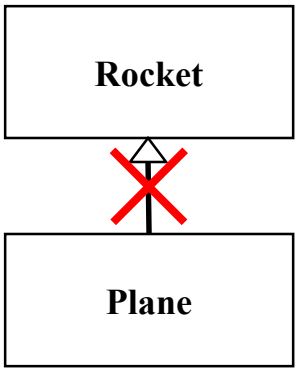
Mechanical Fault



F-16 Bug

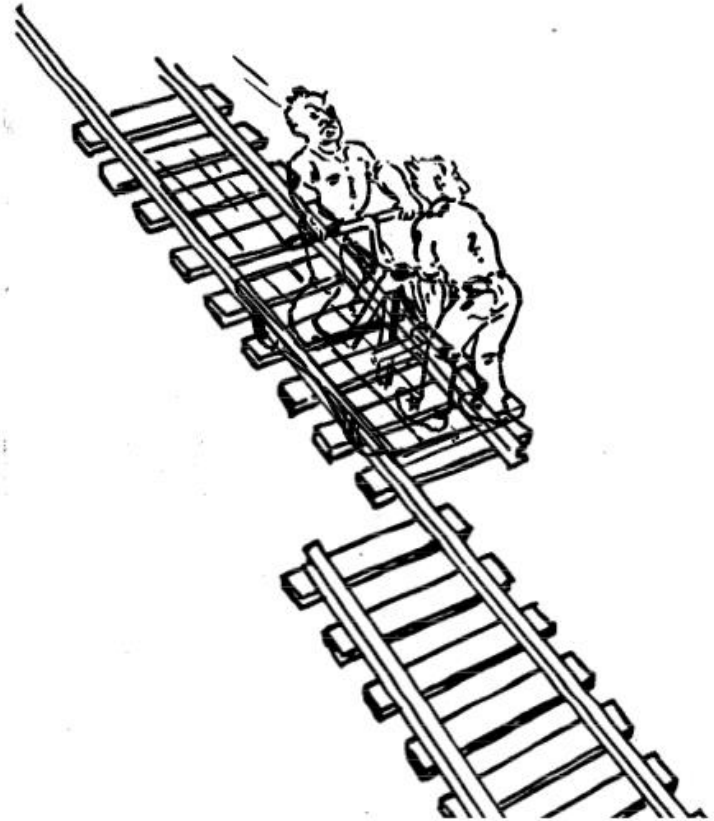


- What is the failure?
- What is the error?
- What is the fault?
  - Bad use of implementation inheritance
  - A Plane is **not** a rocket.

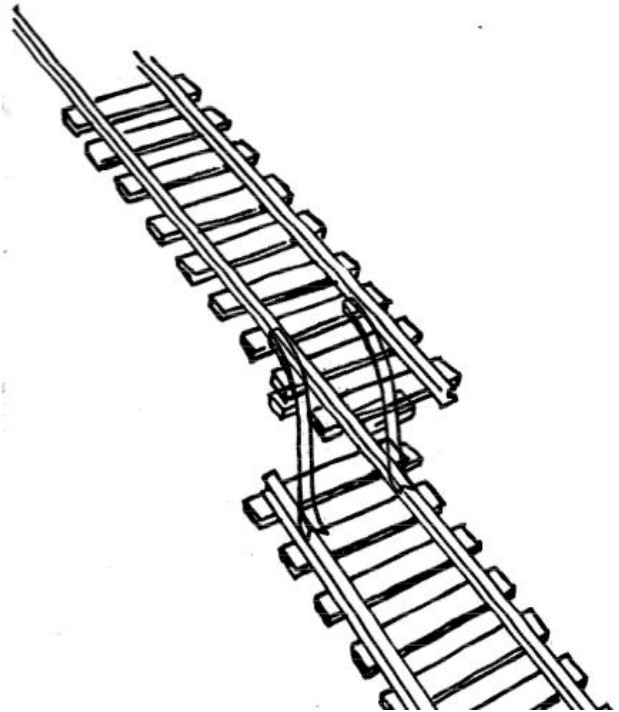


How do we deal with Errors, Failures and Faults?

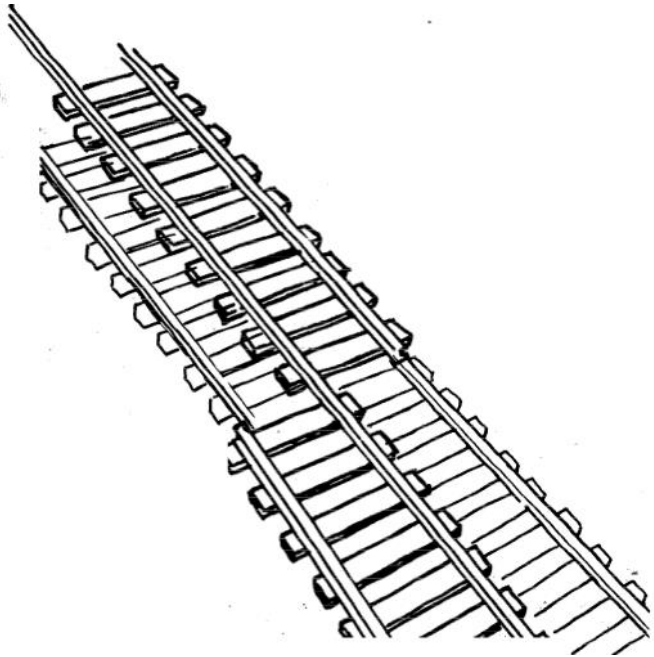
Testing



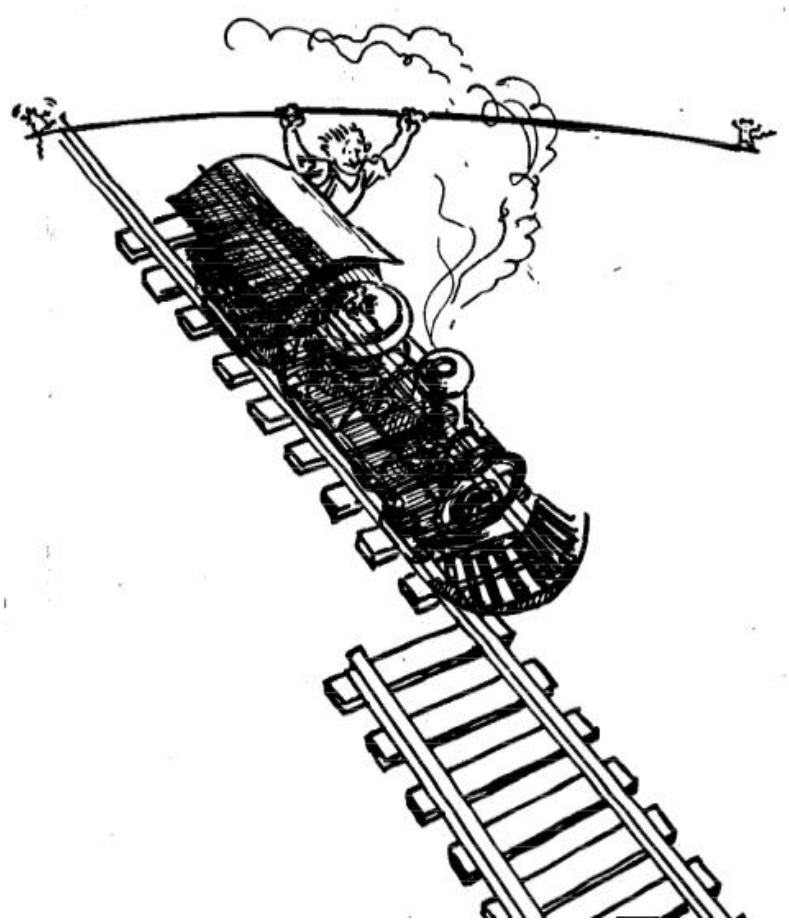
Patching



Building Modular Redundancy



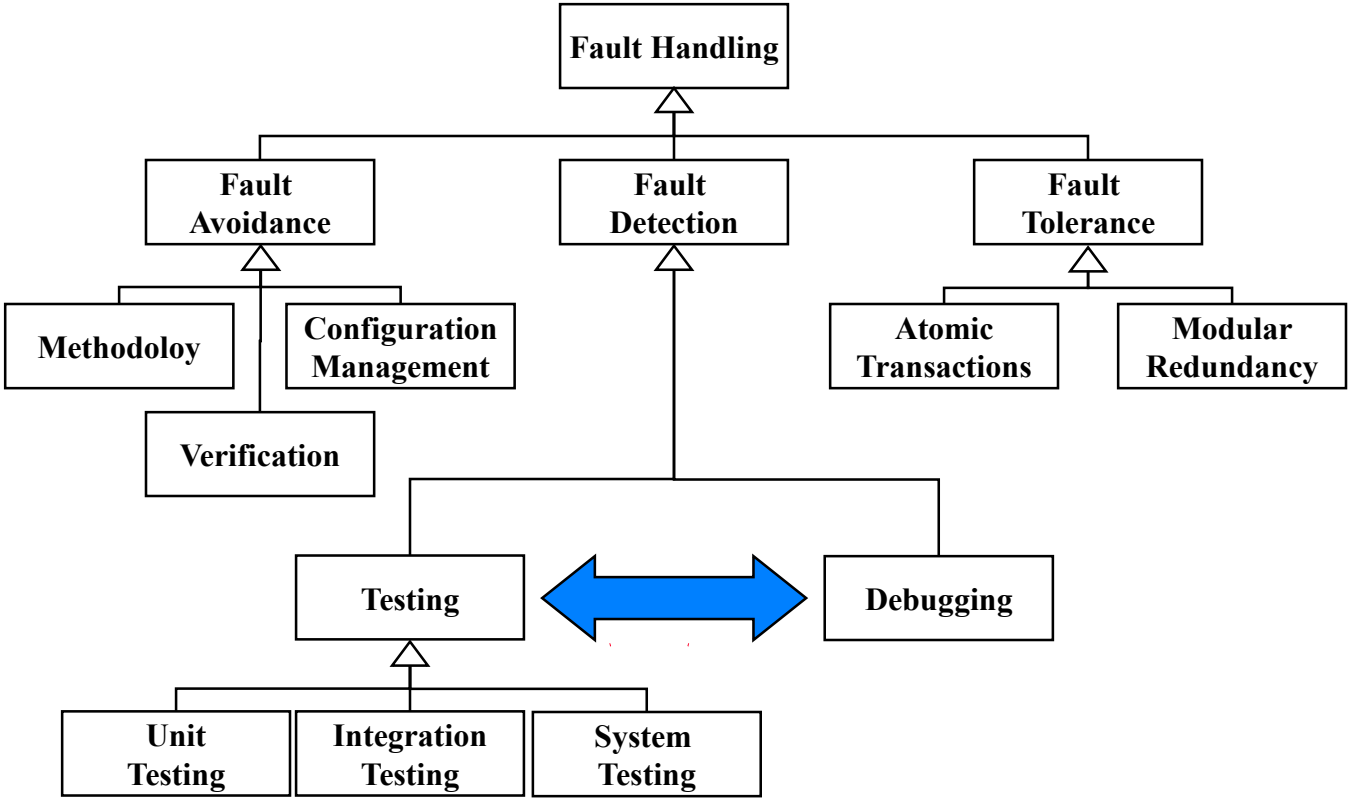
Declaring the Bug as a Feature



Another View on How to Deal with Faults

- Fault avoidance
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (Faults) of an observed failure
  - Monitoring: Collecting and Delivering information about state => Used during debugging
- Fault tolerance
  - Exception handling
  - Modular redundancy.

Taxonomy for Fault Handling Techniques

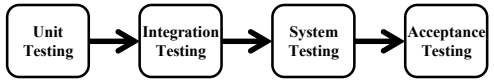


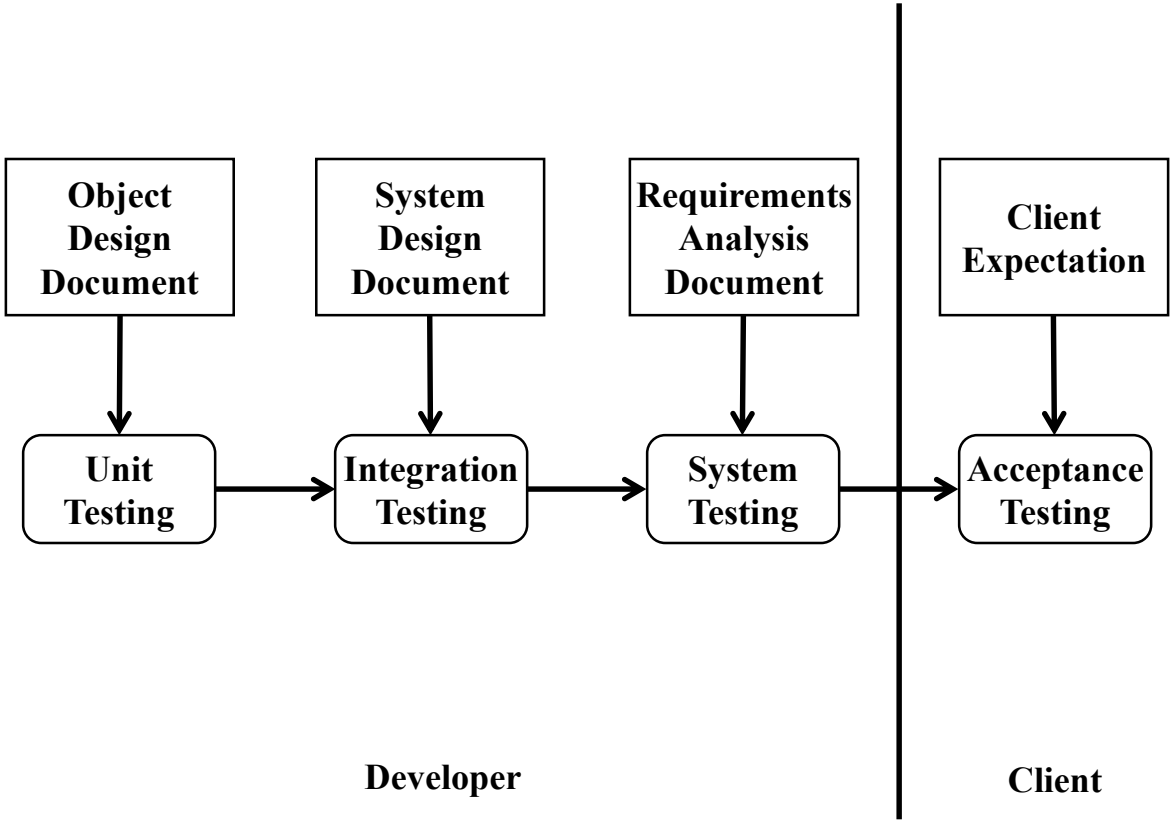
Observations

- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra).
- Testing is not for free

=> Define your goals and priorities

Testing Activities





Types of Testing

Acceptance Test – A measure that ensures that a feature meets functional demands. Usually acceptance tests are tied to user stories or use cases.

Unit test – A smaller test that ensures isolated chunks of functionality (known as units) are functional and operating as expected.

Integration tests – Between unit tests and acceptance tests. Focuses on ensuring that different units function together (said to be integrable).

UNIT Testing

Can be done manually or programmatically – want to define them programmatically since your components may change and manually testing each time is onerous

Basically you boil down exactly what a feature or component should be doing and you logically state these criteria. Each time you modify the feature/component you run the unit tests to see if they pass. When they all pass you move on to integration tests.

Integration Testing

Can be done manually or programmatically

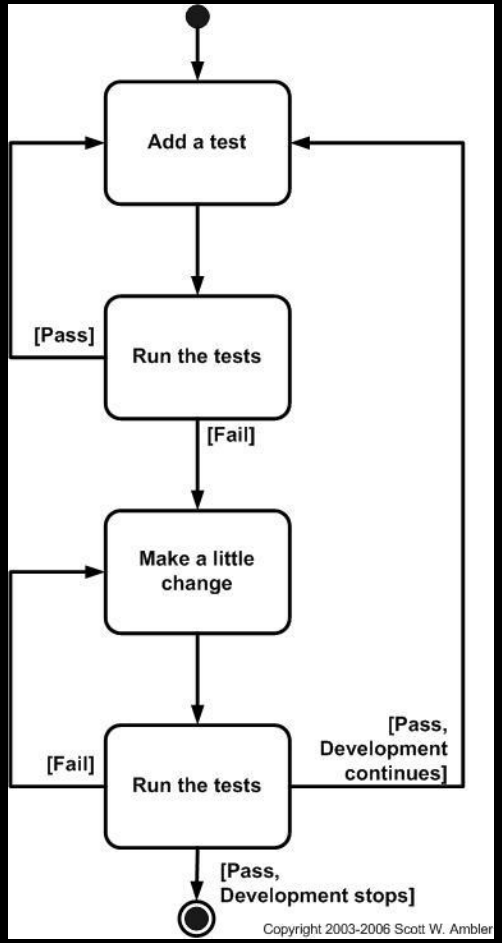
Here you define how different components need to interact and state those constraints logically. When all of the integration tests work – it means you move on to acceptance tests and make sure the collected components satisfy the original goals in the user story or use cases.

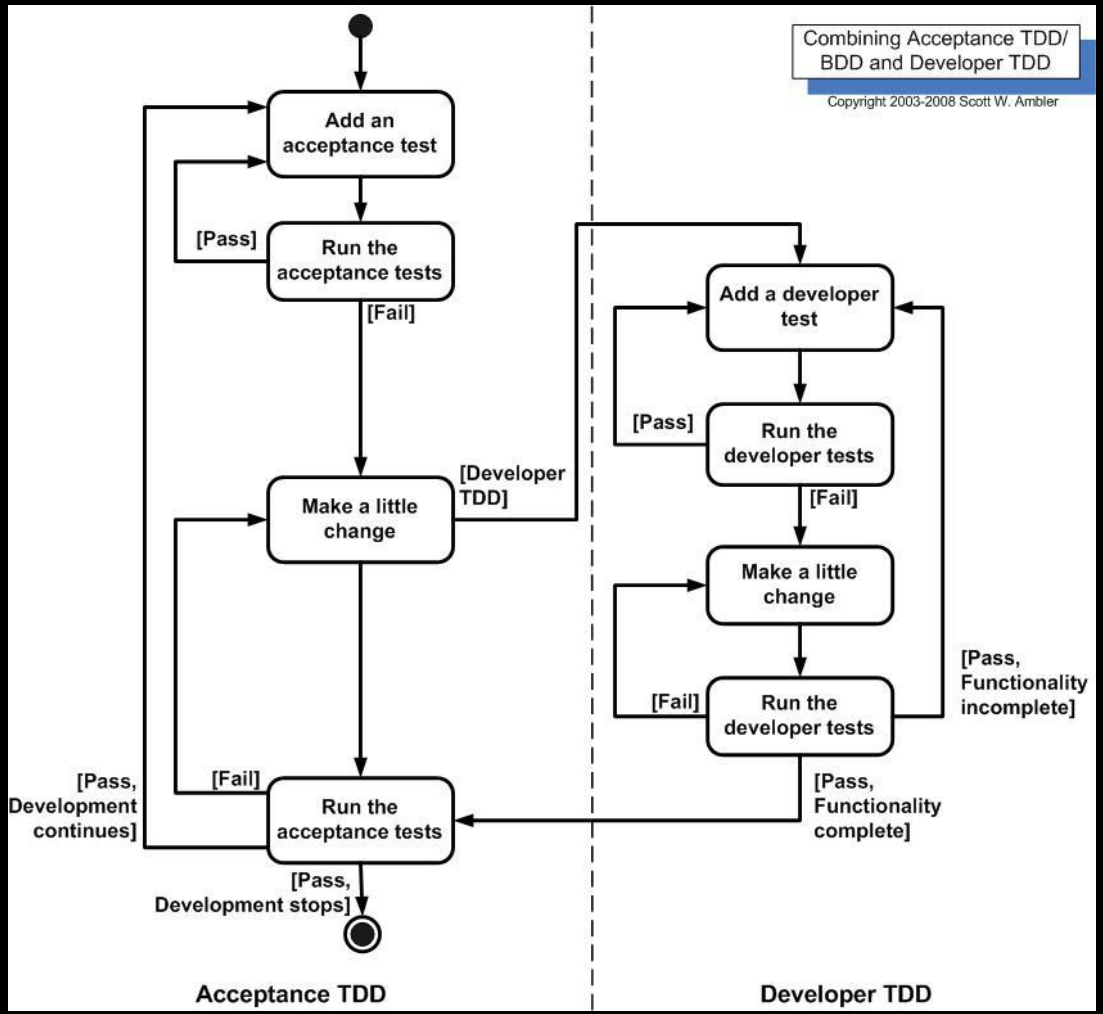
Acceptance Testing

Can be done manually or programmatically – often the former

You basically define the set of all acceptance tests related to your user stories and use cases and – when you demonstrate the app passes all of the tests you are done!

Test Driven Development Core Philosophy




Combining Acceptance TDD/ BDD and Developer TDD
Copyright 2003-2008 Scott W. Ambler
Add an acceptance test
[Pass]
Run the acceptance tests
[Fail]
Make a little change
[Developer TDD]
[Pass, Development continues]
[Fail]
Run the acceptance tests
[Pass, Development stops]
Add a developer test
[Pass]
Run the developer tests
[Fail]
Make a little change
[Fail]
Run the developer tests
[Pass, Functionality incomplete]
[Pass, Functionality complete]
Acceptance TDD
Developer TDD

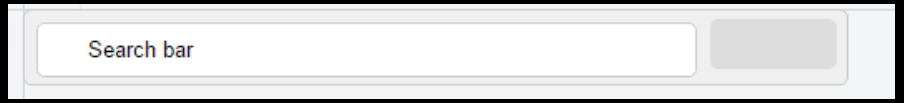
Blackbox and Whitebox testing

Blackbox Testing

Testing a component, feature, or system without knowledge of the inner workings of the entity.

Blackbox Testing

Some random garbage code from:
http://jsfiddle.net/SjafT/1055/

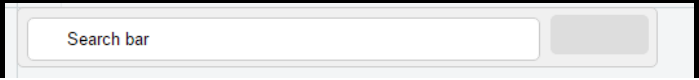


Whitebox Testing

Testing a component, feature, or system with knowledge of the inner workings of the entity.

Whitebox Testing

Some random garbage code from:
http://jsfiddle.net/SjafT/1055/



HTML

```
<div class="wrap left rounded">
    <input id='search' type="text" class="search left rounded" value="Search bar" />
    <button class="go left rounded" onclick="search"><span></span></button>
</div>
```

JAVASCRIPT

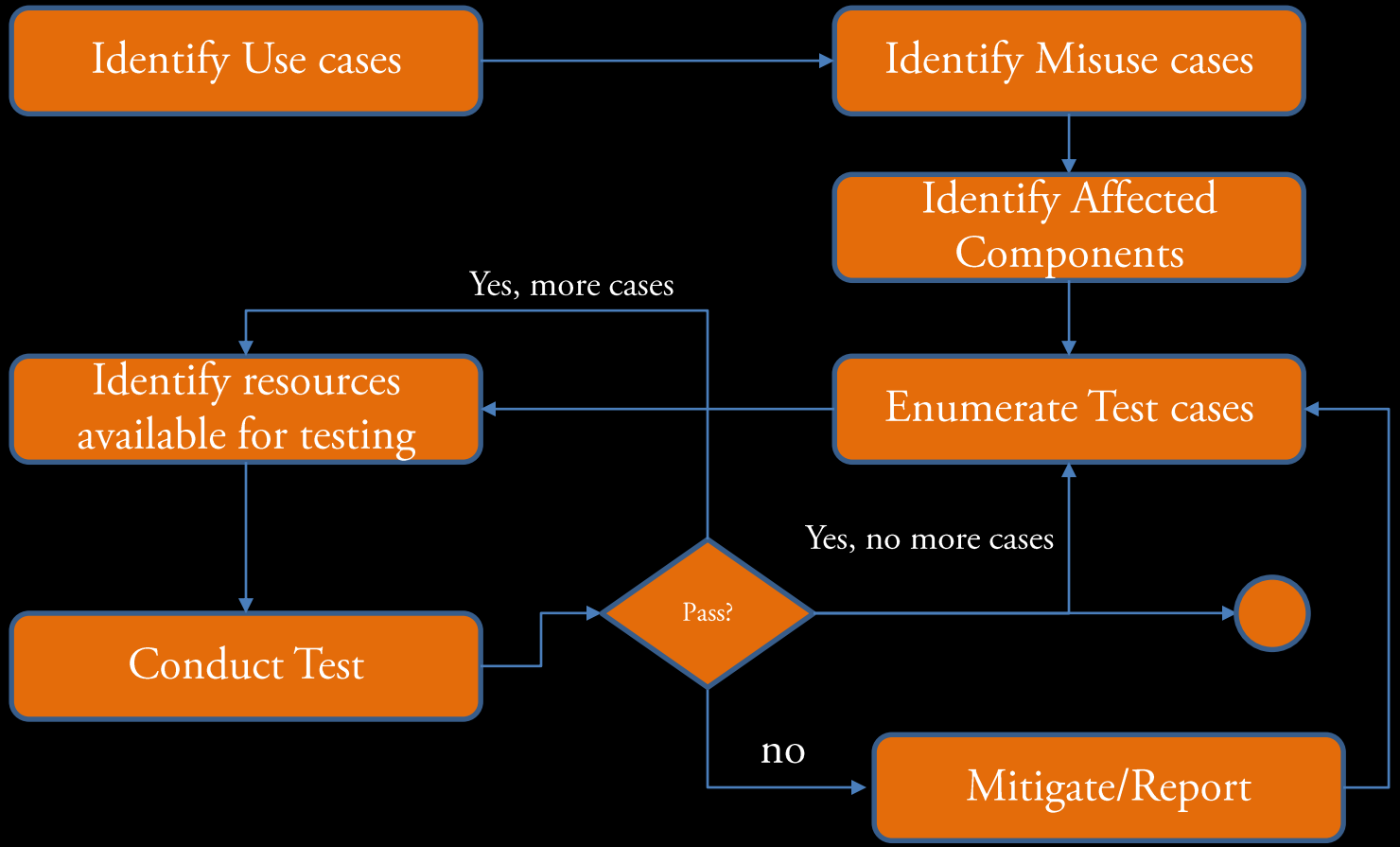
```
function search (){
   var search_string = $('#search').val().toString();
   $.ajax({
           method: 'POST',
           url: 'mybackendserver.com/search',
           data: {searchfield: search_string},
   });

}
```

```
.left {
    float:left;
}
.rounded {
    -webkit-border-radius:5px;
    -moz-border-radius:5px;
    border-radius:5px;
}
.wrap {
    position:relative;
    padding:5px 6px 6px 7px; /* readjust in jsfiddle*/
    background:#f0f0f0;
    border:1px solid #ccc;
    overflow:hidden;
}
.search {
    width:360px;
    position:relative; top:2px; /* readujst in jsfiddle */
    padding:8px 5px 8px 30px;
    border:1px solid #ccc;

}
.go {
    position:relative; top:0;
    margin-left:8px;
    border:none;

}
.go span {
    display:block;
    width:64px; height:28px;
```

Same basic idea:
Understand what can go wrong so you can mitigate the problem or vulnerability.

Conducting an Evaluation

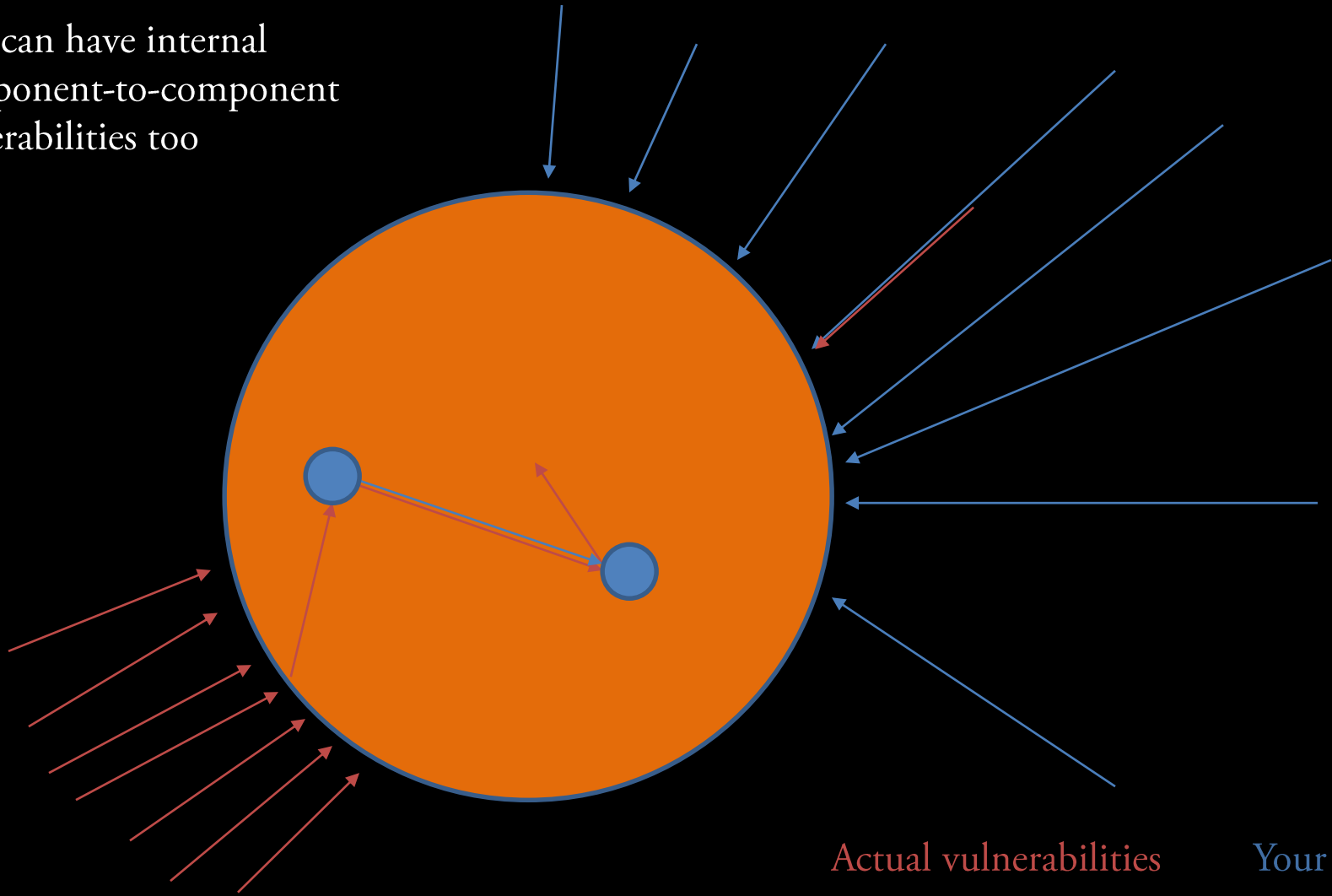
Suggested workflow for security evaluation



Conceptualizing testing strategies

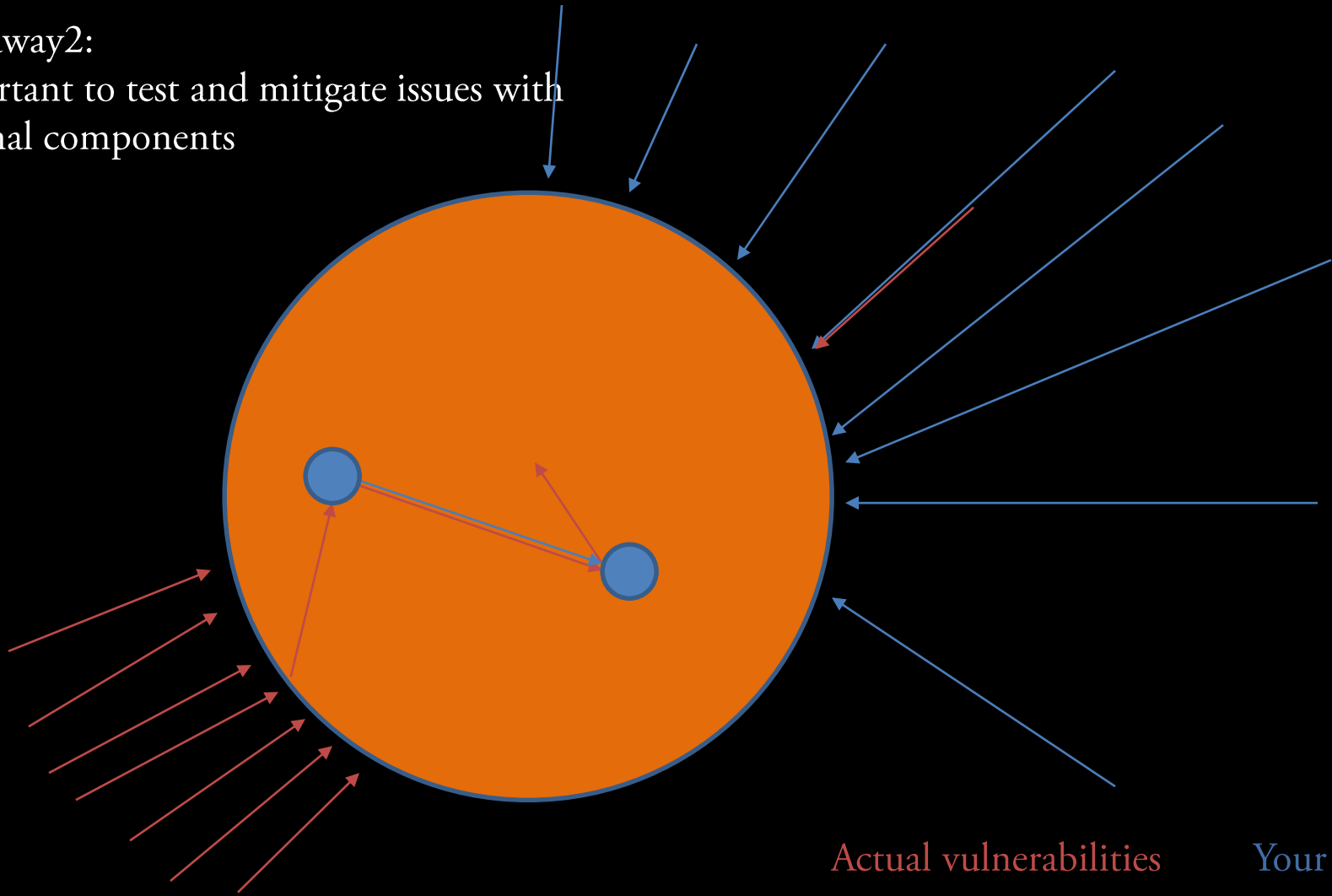
Takeaway:
Having coverage AND
Depth is important

Apps can have internal
Component-to-component
Vulnerabilities too



Actual vulnerabilities Your tests

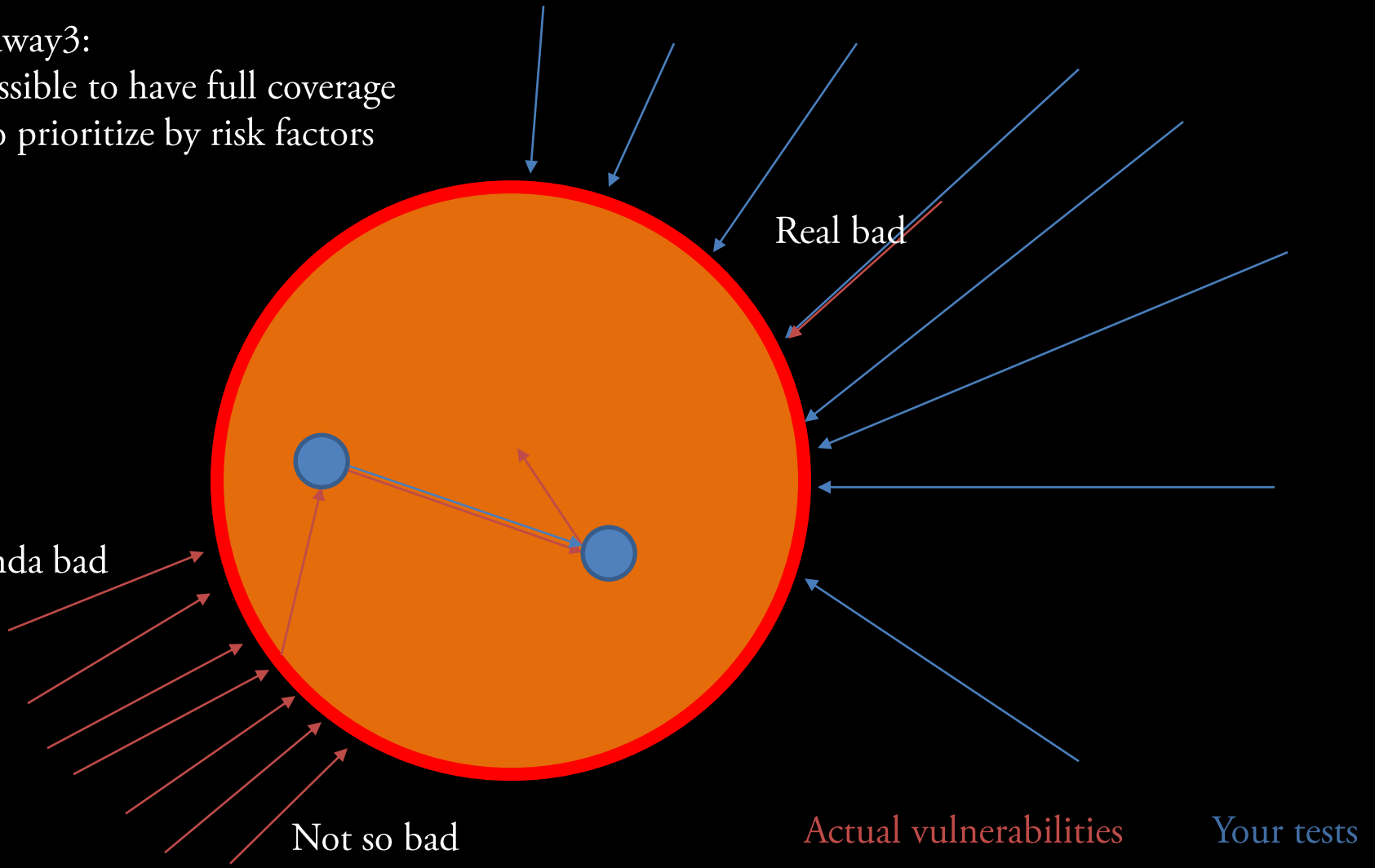
Takeaway2:
Important to test and mitigate issues with internal components



Actual vulnerabilities

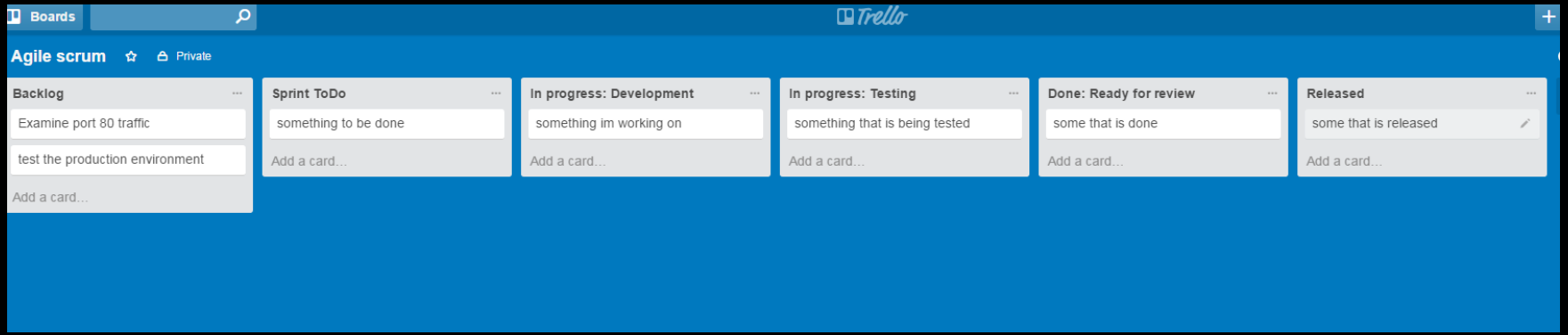
Your tests

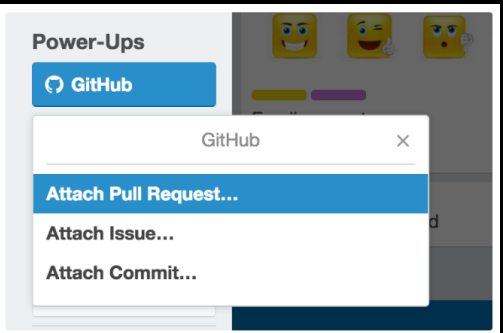
Takeaway3:
Impossible to have full coverage
Try to prioritize by risk factors



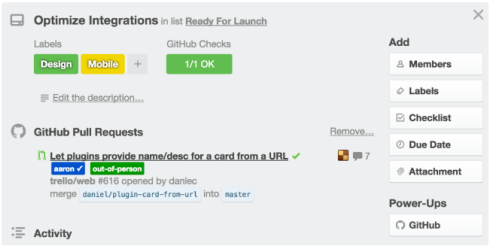
Agile/scrum setup

1. Create a Kanban board in Trello (at least for this class) to keep track of your tasks in the backlog and in the project
2. Create a product backlog from your user stories and use cases
3. Prioritize the tasks in the backlog by the importance of the feature or test for the overall project
4. Select tasks from the backlog for a particular sprint and mark them as 'todo'
5. As you work on tasks move them from todo into 'in progress' or 'done/ready for test'
6. Review/test them as necessary and move to release
7. Link to github commits or issues using the Trello 'github powerup'
8. Update your backlog as you flesh out new tasks





Once you attach a Pull Request (PR) to a Trello card all the relevant information about that branch will be loaded directly onto the card. This includes the name of the PR, whether it has passed checks, any labels (if applicable), who opened the PR, whether it was merged, as well as the person assigned and the number of comments.



Getting Started: Today

1. Break into groups
2. Create a github repo and trello space for collaboration
3. Form your first trello board: call it project requirement elicitation
4. Discuss project goals and start recording ideas as ‘cards’ in trello
5. (on paper or using a charting tool)
   Identify use and misuse cases and create a use-case diagram

Questions?

Matt Hale, PhD

University of Nebraska at Omaha

Interdisciplinary Informatics

faculty.ist.unomaha.edu/mhale/

mlhale@unomaha.edu

Twitter: @mlhale_