Software Quality

17-313 Fall 2024

Foundations of Software Engineering

https://cmu-17313q.github.io

Eduardo Feo Flushing

Sources:

- Effective Software Testing: A developer's guide. Maurizio Aniche
- Software Quality and Testing - TU Delft
- Introduction to Combinatorial Testing. Rick Kuhn

Administrivia

- P3 is out
- Exam grades released
  - You can collect your cheat sheets
- Gradebook is live
- There will be a post-midterm activity with bonus points

Smoking Section

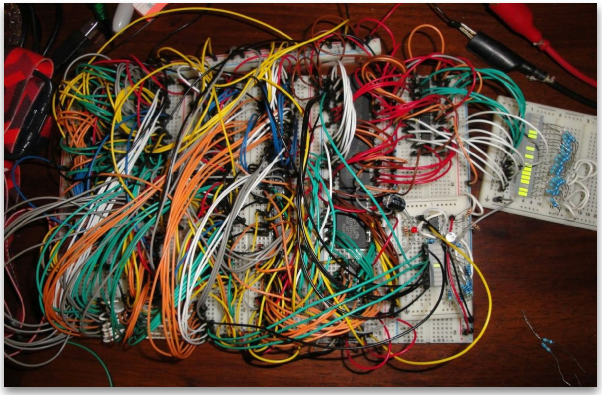
- Last **two** full rows

Learning Goals

- Understand the concepts of software quality and technical debt
- Reflect on personal experiences of technical debt
- Learn best practices for proactively ensuring quality
- Learn techniques for creating functional tests
- Explain the importance of technical debt management
- Learn techniques for managing technical debt

Software Quality

Internal Quality



- Is the code well structured?
- Is the code understandable?
- How well documented?

External Quality

- Does the software crash?
- Does it meet the requirements?
- Is the UI well designed?

Testing

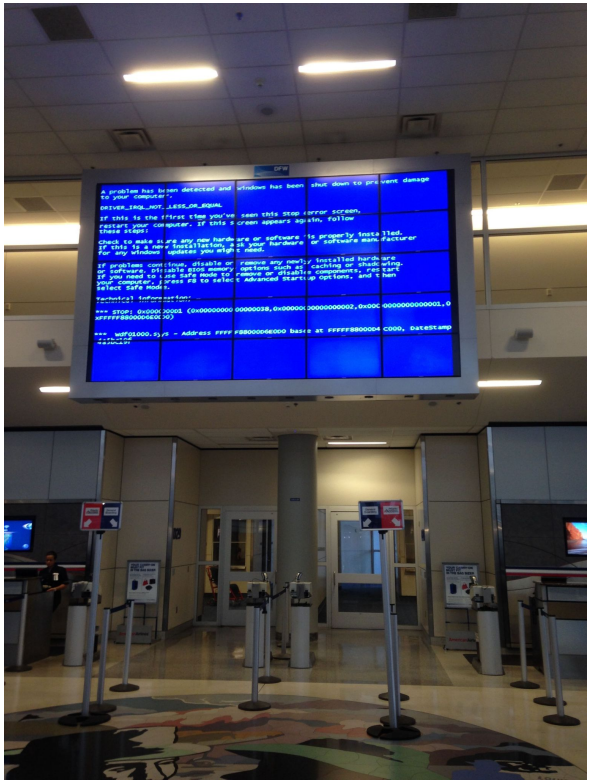
Assuring external quality

Terminology

Failure:

“Deviation of the component or system from its expected delivery, service or result”

“Manifested inability of a system to perform required function”



Terminology

Fault / Defect:

“Flaw in component or system that can cause the component or system to fail to perform its required function”

“A defect, if encountered during execution, may cause a failure of the component or system”

Terminology

Error:

“A human action that produces an incorrect result”

Terminology

Failure:

- Manifested inability of a system to perform required function.

Defect (fault):

- missing / incorrect code

Error (mistake)

- human action producing fault

Bug

And thus:

- Testing: Attempt to trigger failures
- Debugging: Attempt to find faults given a failure

Principles of Testing #1: Avoid the *absence of defects* fallacy

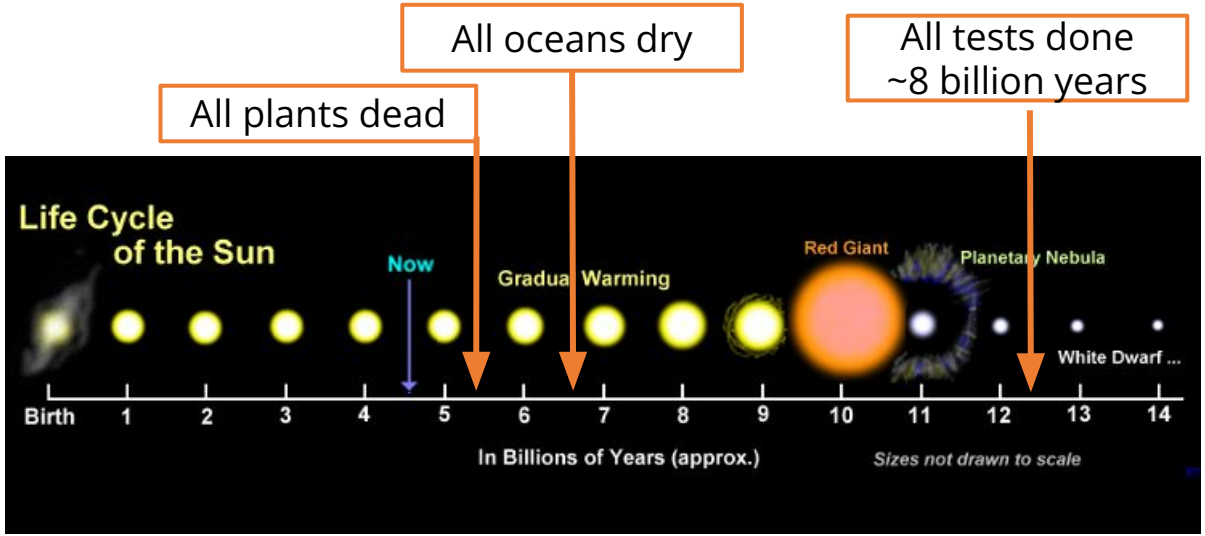
- Testing shows the presence of defects
- Testing does not show the absence of defects!
- “no test team can achieve 100% defect detection effectiveness”

Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #2: Exhaustive testing is impossible

```python
def is_valid_email(email: str) -> bool:
    ...
```

- A simple function, 1 input, string, max. 26 lowercase characters + symbols (@, ., _, -)
- Assume we can use 1 zettaFLOPS: 10^{21} tests per second



Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #3: Start testing early

- To let tests guide design
- To get feedback as early as possible
- To find bugs when they are cheapest to fix
- To find bugs when have caused least damage

Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #4: Defects are usually clustered

- “Hot” components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, …
- Use as heuristic to focus test effort

Effective Software Testing: A developer's guide. Maurizio Aniche

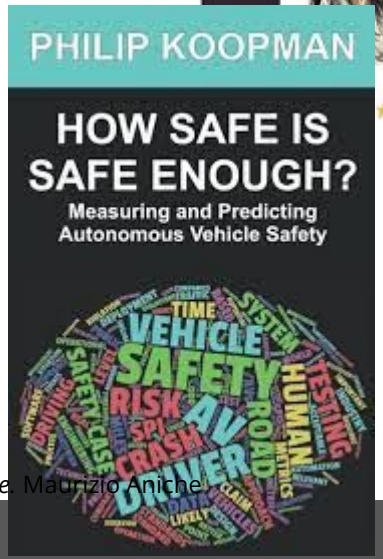
Principles of Testing #5: The pesticide paradox

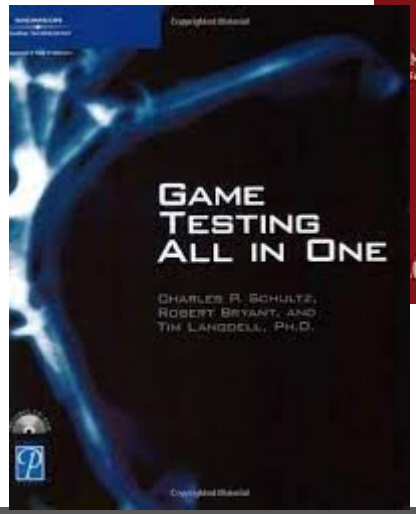
"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."

- Re-running the same test suite again and again on a changing program gives a false sense of security
- Variation in testing

Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #6: Testing is context-dependent





Effective Software Testing: A developer's guide, Maurizio Aniche

Principles of Testing #7: Verification is not validation

Verification

- Does the software system meet the requirements specifications?
- Are we building the **software right**?

Validation

- Does the software system meet the user's real needs?
- Are we building the **right software**?

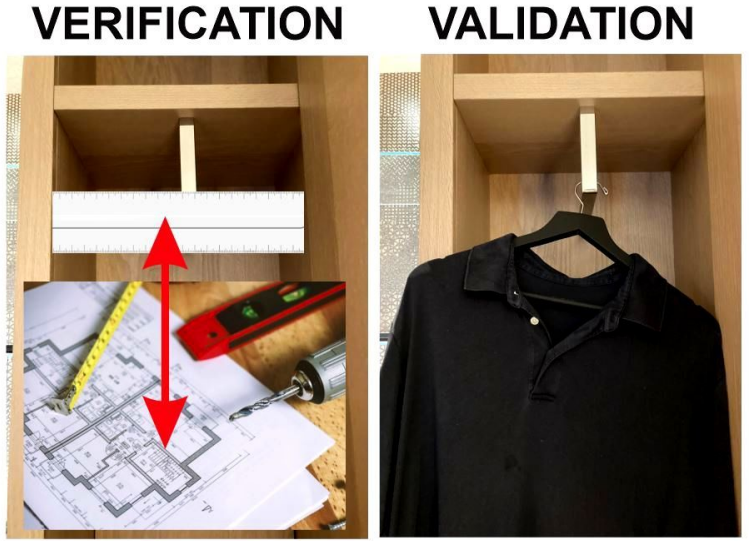


Image Credit: Philip Koopman

Effective Software Testing: A developer's guide. Maurizio Aniche

How to create tests?

Test design techniques

- **Opportunistic/exploratory testing:** Add some unit tests, without much planning
- **Structural testing ("white box"):** Derive test cases to cover implementation paths
  - Line coverage, branch coverage
- **Specification-based testing ("black box"):** Derive test cases from specifications
  - Boundary value analysis
  - Equivalence classes
  - Combinatorial testing
  - Random testing

Specification Testing

Tests are based on the specification

Advantages:

- Avoids implementation bias
- Robust to changes in the implementation
- Tests don’t require familiarity with the code
- Tests can be developed before the implementation

```
"""
 Compute the price of a bus ride:
   - Children under 2 ride for free.
   - Children under 18 and senior citizens over 65 pay half the fare
   - All others pay the full fare of $3.
   - On weekdays (Monday to Friday), between 7am and 9am and
     between 4pm and 6pm, a peak surcharge of $1.5 is added
     to the fare.
  -  During weekends (Saturday and Sunday), there is a flat rate
     of $2 for all riders, except for children under 2.
  -  Short trips under 5 minutes during off-peak times are free,
     except on weekends.
   - If the trip occurs on a public holiday, a special holiday surcharge
     of $2 is added, ignoring other surcharges and the weekend flat rate.
 """
def bus_ticket_price(age: int,
                     ride_datetime: datetime,
                     ride_duration: int,
                     is_public_holiday: bool) -> float:
    ...
```

What about exhaustive testing?

Idea: Try all values!

- **age: int** (2 - 117) years
- **datetime: DateTime** (hh:mm + M/D/Y)
- **rideTime: int** (in minutes, 1 - 2 Hours)
- **is_public_holiday: bool** (2 values)

116 x 1440 (minutes per day) x 1826 (days in the next 5 years) x 120 (ride time) x 2

~ 72 Billion test cases

What about exhaustive testing?

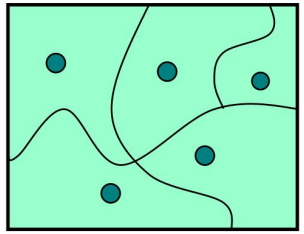
Exhaustive testing is usually impractical – even for trivially small problem

Key problem: choosing test suite

- **Small enough** to finish in a useful amount of time
- **Large enough** to provide a useful amount of validation

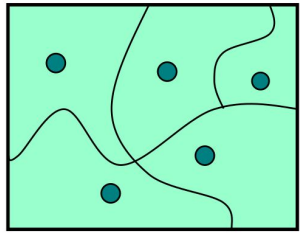
Alternative: **Heuristics**

Equivalence Partitioning



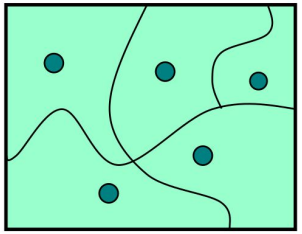
- Identify sets with same behavior (**equivalence class**)
- Try one input from each set
- Equivalence classes derived from specifications (e.g., cases, input ranges, error conditions, fault models)
- Requires domain-knowledge

Example: Equivalence Classes?



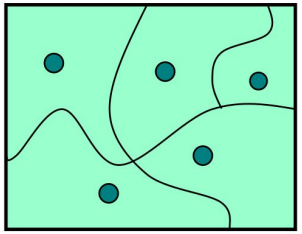
```
"""
Compute the price of a bus ride:
  - Children under 2 ride for free.
  - Children under 18 and senior citizens over 65 pay half the fare
  - All others pay the full fare of $3.
  - On weekdays (Monday to Friday), between 7am and 9am and
    between 4pm and 6pm, a peak surcharge of $1.5 is added
    to the fare.
  -  During weekends (Saturday and Sunday), there is a flat rate
    of $2 for all riders, except for children under 2.
  -  Short trips under 5 minutes during off-peak times are free,
    except on weekends.
  - If the trip occurs on a public holiday, a special holiday surcharge
    of $2 is added, ignoring other surcharges and the weekend flat rate.
"""
def bus_ticket_price(age: int,
                     ride_datetime: datetime,
                     ride_duration: int,
                     is_public_holiday: bool) -> float:
    ...
```

The category-partition method



- Identify the parameters
- The domains of each parameter
  - From the specs
  - Not from the specs
- Add constraints (minimize)
- Remove invalid combinations
- Reduce number of exceptional behaviors
- Generate combinations

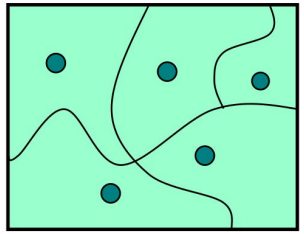
The category-partition method



```
"""
 Compute the price of a bus ride:
   - Children under 2 ride for free.
   - Children under 18 and senior citizens over 65 pay half the fare
   - All others pay the full fare of $3.
   - On weekdays (Monday to Friday), between 7am and 9am and
     between 4pm and 6pm, a peak surcharge of $1.5 is added
     to the fare.
  -  During weekends (Saturday and Sunday), there is a flat rate
     of $2 for all riders, except for children under 2.
  -  Short trips under 5 minutes during off-peak times are free,
     except on weekends.
   - If the trip occurs on a public holiday, a special holiday surcharge
     of $2 is added, ignoring other surcharges and the weekend flat rate.
 """
def bus_ticket_price(age: int,
                     ride_datetime: datetime,
                     ride_duration: int,
                     is_public_holiday: bool) -> float:
    ...
```

Variable	Domains
age	<2, [2,17], [18,65], >65
ride_datetime	weekdays peak and off-peak, weekends peak and off-peak …
ride_duration	<5, >=5
is_public_holiday	F, T

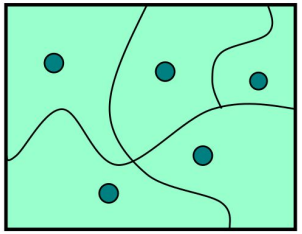
Boundary-value analysis



Key Insight: Errors often occur at the boundaries of a variable value

- For each variable, select:
  - minimum,
  - min+1,
  - medium,
  - max-1,
  - maximum;
  - possibly also invalid values min-1, max+1

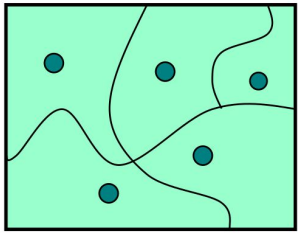
Boundary-value analysis



```
"""
 Compute the price of a bus ride:
   - Children under 2 ride for free.
   - Children under 18 and senior citizens over 65 pay half the fare
   - All others pay the full fare of $3.
   - On weekdays (Monday to Friday), between 7am and 9am and
     between 4pm and 6pm, a peak surcharge of $1.5 is added
     to the fare.
  -  During weekends (Saturday and Sunday), there is a flat rate
     of $2 for all riders, except for children under 2.
  -  Short trips under 5 minutes during off-peak times are free,
     except on weekends.
   - If the trip occurs on a public holiday, a special holiday surcharge
     of $2 is added, ignoring other surcharges and the weekend flat rate.
 """
def bus_ticket_price(age: int,
                     ride_datetime: datetime,
                     ride_duration: int,
                     is_public_holiday: bool) -> float:
    ...
```

Variable	Domains
age	<2, [2,17], [18,65], >65
ride_datetime	weekdays peak and off-peak, weekends peak and off-peak …
ride_duration	<5, >=5
is_public_holiday	F, T

Pairwise testing



Key Insight: some problems only occur as the result of an interaction between parameters/components

- Examples of interactions:
  - The bug occurs for senior citizens traveling on weekends (pairwise interaction)
  - The bug occurs for senior citizens traveling on weekends during peak hours (3-way interaction)
  - The bug occurs for adults traveling long trips during public holidays that are weekends. (4-way interaction)
- **Claim: Considering pairwise interactions finds about 50% to 90% of defects**

Group Activity:

- Use specification testing to create a test suite for the `bus_ticket_price` example
- Explain the heuristics you use to create your test cases
- BONUS: Test the program and find some bugs!

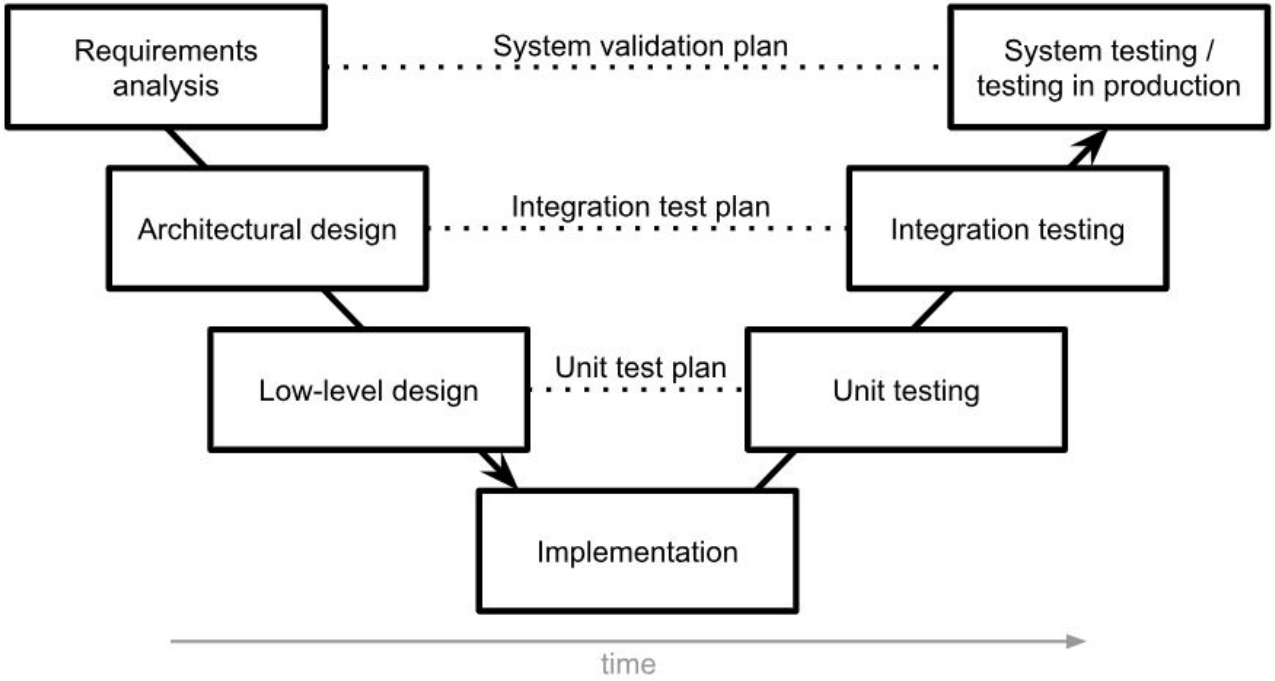
Bus Ticket Pricing Rules

- **Children under 2 ride for free.**
- **Children under 18 and senior citizens over 65 pay half the fare.**
- **All others pay the full fare of $3.**
- **On weekdays (Monday to Friday), between 7am and 9am and between 4pm and 6pm, a peak surcharge of $1.5 is added to the fare.**
- **During weekends (Saturday and Sunday), there is a flat rate of $2 for all riders, except for children under 2 who still ride for free.**
- **Short trips under 5 minutes during off-peak times are free, except on weekends.**
- **If the trip occurs on a public holiday, a special holiday surcharge of $2 is added, ignoring other surcharges and the weekend flat rate.**

https://bit.ly/CMU313-activity

When to create and run tests?

The V-Model


Requirements analysis
System validation plan
System testing / testing in production
Architectural design
Integration test plan
Integration testing
Low-level design
Unit test plan
Unit testing
Implementation
time

Test Driven Development

Tests first!

Popular agile technique

Write tests as specifications before code

Never write code without a failing test

Claims:

- Design approach toward testable design
- Avoid writing unneeded code
- Higher product quality (e.g. better code, less defects)
- Higher test suite quality
- Higher overall productivity