Software Quality

17-313 Fall 2026

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

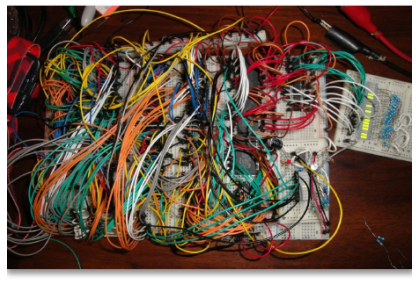
- Don’t forget: team surveys, P2B.

Learning Goals

- Understand the concepts of software quality and testing.
- Distinguish errors, faults, and failures.
- Explain what testing can and cannot establish.
- Derive tests using equivalence partitioning and boundary-value analysis.
- Explain how pairwise testing reduces combinations.
- Compare when tests are created in the V-model and TDD.

Software Quality

You cannot always tell from the outside whether a system is well built.



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?

NOTES

Internal quality is what developers experience when they work on the code. External quality is what users experience when they use the software. A system can look polished externally while being unmaintainable internally, or vice versa.

Testing

Assuring external quality

NOTES

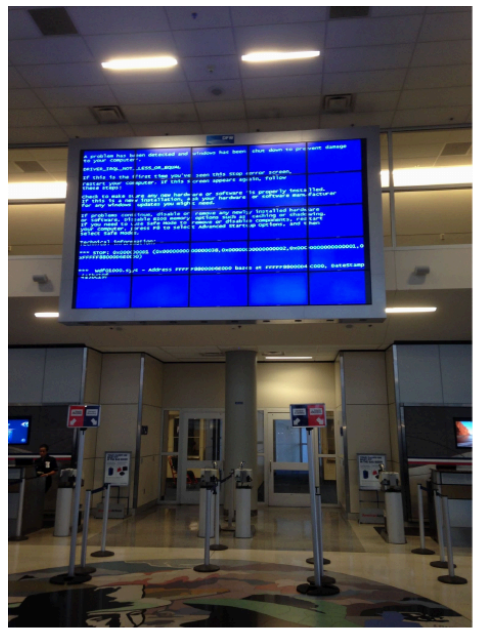
The "Blue Screen of Death" is the critical-error screen Windows shows when the operating system hits a fatal error it cannot safely recover from. To an ordinary user, the cause never matters: all they see is that the software crashed. This particular photo is a BSOD that took over a public digital billboard, a reminder that these failures escape far beyond desktop PCs into embedded and public-facing displays.

Terminology

Failure:

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

"Manifested inability of a system to perform required function"



NOTES

Having a common terminology is important in any field. "Failure," "defect," and "error" get used interchangeably in casual conversation, but distinguishing them precisely lets us talk about what went wrong, where, and why, without ambiguity.

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 or incorrect code.

Error (mistake):

- Human action producing a fault.

Bug is the informal, everyday term for a defect (fault).

And thus:

- **Testing:** attempt to trigger failures.
- **Debugging:** attempt to find faults given a failure.

NOTES

Side note: Edsger W. Dijkstra didn't like the term "software bug." He felt it made defects sound like something that crawled in from outside, rather than a mistake the programmer made.

Principles of Testing #1: Testing Shows the Presence, Not the Absence, of Defects

- Testing shows the presence of defects.
- Testing does not show the absence of defects!
- *"No test team can achieve 100% defect detection effectiveness."*

Edsger W. Dijkstra

Dijkstra on Bugs

NOTES

This is the most common misconception about testing: passing tests is evidence of correctness for the cases you tried, not proof that no bugs remain. "Testing shows the presence of bugs, but never their absence", a quote attributed to computer scientist Edsger W. Dijkstra (The Humble Programmer, a must-read). Side Note: Dijkstra didn't like the term "software bug." He felt it made defects sound like something that crawled in from outside, rather than a mistake the programmer made.

Principles of Testing #2: Exhaustive Testing Is Impossible

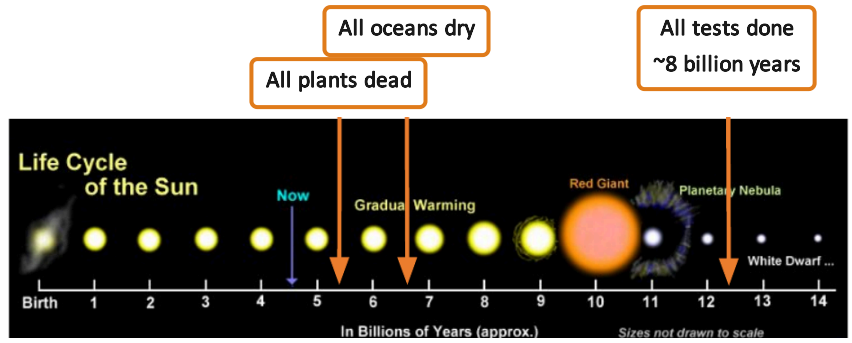
```
def is_valid_email(email: str) -> bool:
    ...
```

- A simple function, 1 input, string, max.

  26 lowercase characters plus symbols

  `(@, ., _, -)`
- Assume we can use 1 zettaFLOPS: 10^{21}

  tests per second



NOTES

Even if we used all the CPU power on the planet, we would not be around by the time the tests finished running.

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 they have caused the least damage.

NOTES

Self-explanatory slide. Start testing early, period.

Principles of Testing #4: Defects Are Usually Clustered

- "Hot" components requiring frequent change, bad habits, poor developer practices, tricky logic, business uncertainty, innovative code, size, etc.
- Use as a heuristic to focus test effort.

NOTES

Bugs are also social animals: they like to be together. This is related to the 80/20 rule: a small fraction of files tends to account for a disproportionate share of defects. See Walkinshaw, Neil, and Leandro Minku, "Are 20% of Files Responsible for 80% of Defects?", Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2018, for a closer look at how well that specific ratio actually holds up.

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 is needed.

NOTES

Boris Beizer described the pesticide paradox in Software Testing Techniques (2nd ed., 1990): every testing method leaves behind subtler defects that it is less effective at finding. As a result, test suites "wear out" and must evolve over time.

Principles of Testing #6: Testing Is Context-Dependent

- A safety-critical system, a mobile game, and a one-off script all warrant very different testing effort and rigor.
- What counts as "enough testing" depends on the cost of failure, not on a fixed checklist.

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**?

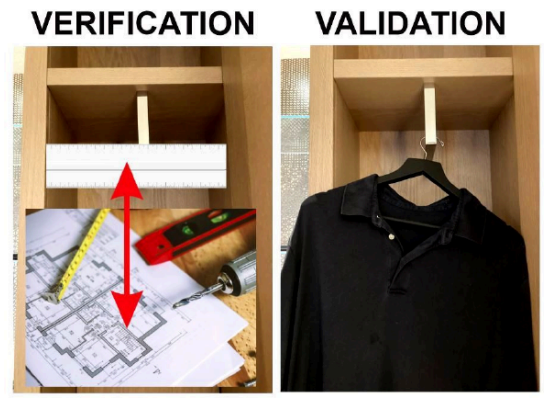


Image credit: Philip Koopman

NOTES

Verification asks whether we built the system according to its specification. Validation asks whether that specification describes what users actually need. A system can pass every test and still fail its users if the requirements were incomplete, ambiguous, or simply wrong.

How to Create Tests?

Test design techniques

Test Design Techniques

- **Ad hoc testing:** add some tests without a systematic strategy.
- **Exploratory testing:** deliberate testing where learning, test design, and execution happen together.
- **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.

NOTES

Because exhaustive testing is impossible, we need systematic ways to choose a small but useful set of test cases. We can derive tests from the implementation, from the specification, or through exploration; in practice, strong test suites usually combine all three approaches.

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:
  - Base fare is QAR 3
  - Children under 2 ride for free
  - People under 18 and senior
    citizens over 65 pay half the fare
  - On weekdays (Sun-Thu), between
    7AM - 9AM and 4PM - 6PM a peak surcharge
    of QAR 1.5 is added.
  - On weekends (Fri-Sat), there is a
    flat rate of QAR 2 for all riders,
    except children under 2 still free.
  - Short trips under 5 minutes during
    off-peak time are free, except
    on weekends.
"""
def busTicketPrice(age: int,
                   ride_datetime: datetime,
                   ride_duration: int) -> float:
    ...
```

What About Exhaustive Testing?

Idea: try all values!

- `age` : `int` (2 to 117) years
- `ride_datetime` : `datetime` (hh:mm + M/D/Y)
- `ride_duration` : `int` (in minutes, 1 to 2 hours)

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

~ 36 billion test cases

NOTES

Unlike the email example, 36 billion is actually doable on modern hardware. But please, don't do it: running (and maintaining) a suite that size buys you almost nothing beyond a much smaller, well-chosen set of cases, and it costs a lot of time.

What About Exhaustive Testing?

Exhaustive testing is usually impractical, even for a trivially small problem.

Key problem: choosing the 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 the 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:
  - Base fare is QAR 3
  - Children under 2 ride for free
  - People under 18 and senior citizens over 65 pay half the fare
  - On weekdays (Sunday to Thursday), between 7AM and 9AM
    and between 4PM and 6PM a peak surcharge of QAR 1.5 is added.
  - On weekends (Friday and Saturday), there is a flat rate of
    QAR 2 for all riders, except children under 2 still free.
  - Short trips under 5 minutes during off-peak time are free,
    except on weekends.
"""
def busTicketPrice(age: int,
                   ride_datetime: datetime,
                   ride_duration: int) -> float:
    ...
```

NOTES

Some example equivalence classes for this function: age (infant <2, child [2,17], adult [18,65], senior >65), ride_datetime (weekday off-peak, weekday peak, weekend), ride_duration (short <5 minutes, regular >=5 minutes).

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

Variable	Domains
age	<2, [2,17], [18,65], >65
ride_datetime	date: (weekdays, weekends) time: (peak and off-peak)
ride_duration	<5, >=5

Boundary-Value Analysis

Key insight: errors often occur at the boundaries of a variable's value.

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

Boundary-Value Analysis

Variable	Boundary values to test
age	1 / 2 (infant), 17 / 18 (child), 65 / 66 (senior)
ride_duration	4 / 5 minutes (short-trip cutoff)
ride_datetime (peak windows)	7AM / 9AM and 4PM / 6PM (start and end of each window)

Pairwise Testing

Key insight: some problems only occur as the result of an interaction between parameters or 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).
- Studies of some systems have found that pairwise testing detects many interaction faults, but effectiveness depends on the system and fault distribution.

Pairwise Testing: A Concrete Example

Three parameters from `busTicketPrice`: age class (4 values), day type (3 values), duration class (2 values). Full cross-product: 4 × 3 × 2 = **24**. Pairwise coverage: every pair of values at least once in just **12** rows.

#	age class	day type	duration class
1	infant	weekday off-peak	short
2	infant	weekday peak	regular
3	infant	weekend	short
4	child	weekday off-peak	regular
5	child	weekday peak	short
6	child	weekend	regular
7	adult	weekday off-peak	short
8	adult	weekday peak	regular
9	adult	weekend	short
10	senior	weekday off-peak	regular
11	senior	weekday peak	short
12	senior	weekend	regular

With only 3 parameters the saving is modest (24 → 12). Add one more 3-valued parameter, e.g. payment method, and full cross-product jumps to 72, while pairwise coverage grows to only around 15 rows. The gap widens with every parameter you add.

NOTES

This is a real pairwise-covering array: every age class appears with every day type (12 rows are needed just for that, since 4 times 3 is already 12), and duration alternates so that every age and every day type is also paired with both duration classes at least once. Row 9, an adult on the weekend, is the same kind of combination that exposed the actual bug in the debrief: a weekend row whose clock time happens to land in what would be a weekday peak window.

With only 3 parameters the saving looks unimpressive: 24 down to 12, a 2x reduction. The real payoff shows up as more parameters get added. Full cross-product grows multiplicatively with every new parameter (a 4th 3-valued parameter takes 24 to 72), while a pairwise-covering array grows much more slowly, since it only has to cover every pair of values, not every combination. That gap is the actual argument for pairwise testing; a 2x saving at n=3 parameters is just the smallest case of a pattern that gets dramatic fast.

Group Activity

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

Bus Ticket Pricing Rules

- Base fare: QAR 3.
- Children under 2: free.
- Under 18 or over 65: half fare.
- Weekdays 7AM-9AM and 4PM-6PM: +QAR 1.5 peak surcharge.
- Weekends: flat QAR 2 (children under 2 still free).
- Trips under 5 min off-peak: free (not on weekends).

bit.ly/313-blackbox

Debrief: A Reference Test Suite

Boundary-value analysis, equivalence partitioning, and pairwise testing, mixed together. Not the full cross-product.

Age	Day	Time	Duration	Technique / why this case
1	weekday	10:00	10 min	Boundary: infant, age < 2
2	weekday	10:00	10 min	Boundary: just not infant
17	weekday	10:00	10 min	Boundary: child, half fare
18	weekday	10:00	10 min	Boundary: adult, full fare
65	weekday	10:00	10 min	Boundary: "over 65" excludes 65
66	weekday	10:00	10 min	Boundary: senior, half fare
30	weekday	6:59 / 7:00 / 7:01	10 min	Boundary: peak window edge (same pattern at 9:00, 16:00, 18:00)
30	weekday	11:00	4 / 5 min	Boundary: short-trip edge, off-peak
30	weekend	10:00	10 min	Equivalence: weekend flat rate
30	weekend	8:00	10 min	**Pairwise: weekend × peak-hour clock time**

Full cross-product: 6 ages × 12 time points × 2 durations × 2 day types = 288 cases. This suite selects far fewer, chosen to cover every boundary plus one telling interaction.

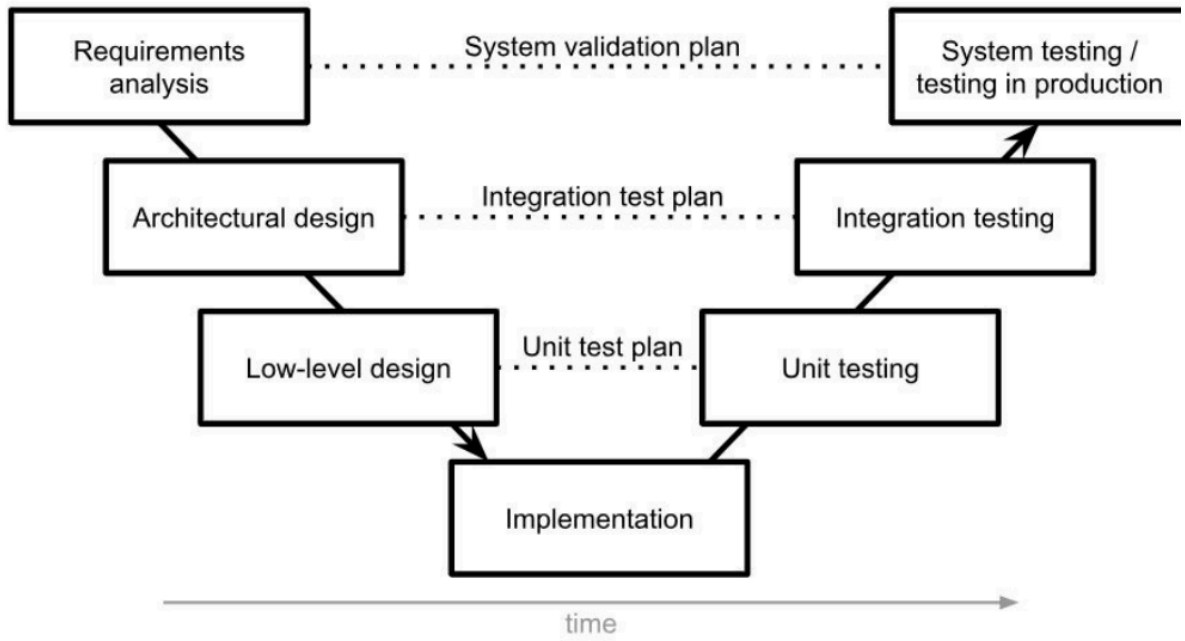
NOTES

The last row is the one that matters most: age and duration are unremarkable, but combining "weekend" with a clock time that falls in a weekday peak window exposes the bug from the implementation (the peak surcharge gets added on top of the flat weekend rate). Testing "a weekend case" and "a peak-hours case" separately, without ever combining them, would never have found it.

When to Create and Run Tests?

The V-Model

The V-Model



NOTES

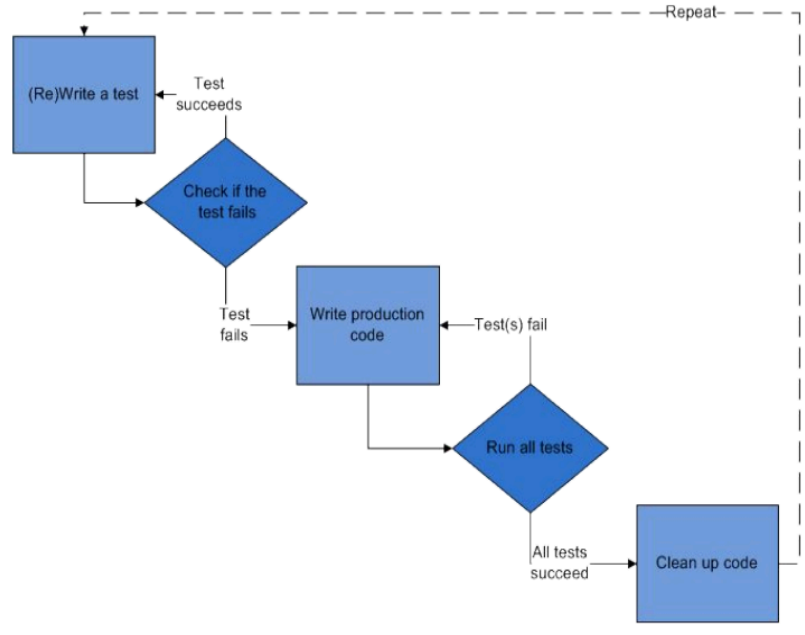
Unit, integration, and system tests are just a categorization by scope: how much of the system a test exercises, not when it runs. In plan-driven processes like the V-model, that scope maps to fixed stages: unit tests early, integration as the system takes shape, system tests once a prototype exists. In agile processes, prototypes are built incrementally, so all three can happen within the same sprint.

Test Driven Development

Tests first! A popular agile technique: write tests as specifications before code, and never write code without a failing test.

Claims:

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



NOTES

We will revisit test-driven development. For now, just remember that tests can be, and should be, a central piece of the software development process, not an afterthought bolted on at the end.

Takeaways

Testing shows the presence of defects, not their absence.

Good tests come from the specification, not from guessing at the implementation.

Start testing early: the earlier a defect is found, the cheaper it is to fix.