Test-Driven Development

A different order of development – write tests *first*, before writing the code.

Steps – think, red, green, refactor, repeat

- list scenarios for the new behavior
  - what's the basic case? what are the variants or special cases?
- write a test that will pass when one of the behavior variants is met
- run all the tests – the new test should fail (red)
  - failure shows that new code is needed, and helps establish that the test is working correctly
- write the minimum code to pass the new test
  - hard coding is acceptable, elegance is not required
- run all the tests – the new test should pass (green)
- refactor to improve code, repeating tests to ensure nothing breaks
  - reorganize code – move code, split methods, rearrange inheritance hierarchies, remove duplicate code, etc
- repeat until all tests are implemented and pass
  - the new behavior is then complete

Limits of TDD

- primarily geared around unit testing
  - doesn't replace other kinds of testing or cover other aspects of the software
- a large suite of passing tests can result in false confidence that all is well
  - tests verify against the spec, but don't validate the spec
- tests tightly coupled to implementation details or with excessive setup and mocking have a high maintenance burden
- empirical studies have found mixed results regarding whether productivity is helped or harmed
  - TDD process involves overhead

Benefits of TDD

- writing the test first requires thinking about and fully defining the method's contract before implementing
- provides concrete feedback on progress
- results in a test suite supporting refactoring and continuous integration
- empirical studies have found evidence of improved software quality (fewer bugs) but results vary

Key Takeaways

TDD is a tool, not the only way.

- not all tasks are well-suited to TDD
- key takeaway: write tests early, not as an afterthought
  - focus on risk – lower-level tests when risk is highest

Why TDD matters here.

- thinking about specs early on prevents integration bugs from vague contracts
- a defense against AI slop and a safety net for refactoring
- when you implement first, a common instinct is to protect your code rather than try to break it
  - also satisfying to see tests pass

Debugging

A test failed. Now what?

- a failing test tells you *what* is wrong, but not *why*
- debugging is the process of finding the “why”

Debugging and testing go together.

- a failing test is a specific entry point for debugging
  - each test is for a specific behavior
- a passed test shows that the problem is actually fixed and guards against it returning

A Debugging Mindset

- goal: find the root cause of the failure, don’t just make the symptom disappear
- debugging is about finding a discrepancy
  - somewhere what you believe is true about your program and what’s actually happening have diverged
- work methodically
  - form a hypothesis about the cause of the failure, then check it
  - narrow the gap between where things are known to be correct and where something is known to be wrong
  - have a reason why your fix will fix things – don’t randomly change things hoping the problem goes away
- verify, don’t assume
  - wrong assumptions about program behavior hide bugs

Developing Hypotheses

- start with the most obvious, likely, and/or easy to test hypotheses
  - don’t stop if the first one isn’t it – what else could explain the problem?
- e.g. the program isn’t printing something it should
  - is there even code to print the line in question?
  - if yes, is that code even being run?
    - is it inside an ‘if’? maybe the condition is wrong…
    - is it in a method? is that method getting called?

Finding Bugs

- work backwards from where the failure was observed
  - trace the wrong value back through the calls that produced it – where did this value come from?
- trace forwards from a known good point
  - start where you’re confident things are correct, and step forward until something goes wrong
- narrow the gap between known-good and observed-bad
  - pick a point in between and check

Finding Bugs

- if you are stuck –
  - *expand* – widen your search – the bug may not be where you assumed
    - what other hypotheses can you come up with?
  - *augment* – add more instrumentation to gather more evidence before guessing even farther
  - *rewrite* – simplify or reimplement the suspect piece
    - if the bug disappears, you've confirmed where it was

Tool 2 – Debugger

- pause and inspect
  - a debugger lets you pause execution at an exact line (a *breakpoint*) and examine the program's full state at the moment
    - *conditional breakpoints* stop only if certain conditions are met
- stepping through code
  - *step over* – run the current line, executing any method calls, and move on to the next line
  - *step into* – if the current line calls a method, follow execution into it
  - *step out* – finish the current method, return to the caller
- observation
  - *watch* – track a specific variable or expression continuously as you step
  - *call stack* – shows the chain of method calls that got the program to this point

Tool 1 – Print Statements

- advantages
  - familiar
  - available anywhere, no external tools required
  - can see results of a whole run at once
- where they fall short
  - requires editing and rebuilding code each time
    - can introduce new bugs
  - can only examine what you thought to print
  - get the whole run's worth of output at once – can be overwhelming
  - easy to leave clutter behind
- guidelines
  - use print statements for quick, targeted checks
  - use the debugger for more complex situations or when output would be overwhelming
  - always remove ad-hoc prints before committing
    - convert anything worth keeping into real logging calls

Tool 3: Code Instrumentation – Assertions

- an *assertion* states something that must always be true at this point in the program

```
assert n >= 0 : "pickUp count must be non-negative, was " + n;
```

  - if the condition is false, an `AssertionError` is thrown
  - otherwise execution continues normally
- assertions are for documenting and catching *programmer errors* – things that should be impossible in correct code
  - invalid external input should be handled by throwing exceptions
    - use `IllegalArgumentException` for violated preconditions
- advantages
  - cheap – disabled by default in Java, so no impact on production code if left in
  - self-documenting
  - catch problems at the source
- limitations
  - disabled by default so can forget to turn them on and gain a false sense of security

Tool 4: Logging

- print statements and debugger sessions are temporary
  - they are gone once you close the terminal or stop the debugger
- print statements must be removed once debugging is done
- *logging* writes messages to a persistent record
  - can stay in the code permanently, providing info for future debugging or analysis
- logging is useful for situations that debuggers can't handle
  - e.g. issues that only show up after the program is running for a while or in production or in a context where you can't pause execution

Logging Levels

- logging frameworks organize log messages by *severity level*

Level	Meaning
`SEVERE`	A serious failure — something is broken
`WARNING`	Something unexpected, but not fatal
`INFO`	Routine, high-level status — "dealt 4 cards to column 2"
`CONFIG`	Configuration-related information
`FINE` / `FINER` / `FINEST`	Increasingly detailed debug-level tracing

  - the level assigned to a log message is a statement about how important it is to see, not whether it is "debug code" to be removed later
- logging output is controlled by adjusting the logging threshold and can be routed to different handlers (console, file, etc)
  - doesn't require editing and rebuilding code

Debugging Tool Summary

technique	applicability	stays in shipped code?	notes
print statements	a quick, isolated question timing-sensitive code	no – remove before committing	only ever meant as a temporary probe changes code – can introduce new bugs
assertions	assumptions that should always hold	can stay	disabled by default at runtime
logging calls	information worth keeping permanently – can support debugging production code	yes, permanently	verbosity controlled by configured level, not by editing code
debugger breakpoints	exploring code execution or state	n/a	not part of the code – set in the IDE