Testing and Debugging

Overview

- why we test
- testing foundations

- test-driven development
- debugging
- logging

- test planning and coverage
- test doubles

Why Test

It compiled, it ran, it's done.

- running once without crashing ≠ a correct program
- many bugs aren't crashes – they are wrong answers that look fine

The Cost Curve

- a bug found while you are writing the code
- a bug found by another developer on your team
- a bug found by a user after shipping

Testing is how you find bugs on the cheap end of the curve.

Levels of Testing

Four main levels of software testing –

- *unit* – do the individual pieces (one method or class) work correctly?
- *integration* – do multiple pieces work correctly together?
- *system* – does the whole application behave correctly end-to-end?

verification

- *acceptance* – does the software solve the intended problem?
  - there are various types of acceptance tests geared towards different aspects

validation

Unit Testing

- focuses on individual units (a single method or class)

Unit testing is used to –

- verify individual pieces of code –
  - the whole won't work if the individual parts don't work
- ensure business logic works correctly
- catch programming errors early
  - squash bugs while they are cheap

Who and when –

- developers generally write and execute unit tests during development
- unit tests are often automated
  - supports continuous integration and regression testing

Verification vs Validation

- *verification*: "did we build the thing right?"
  - does the program match the specification/design?
- *validation*: "did we build the right thing?"
  - does it actually meet the user's need?

Both are important –

Passing every verification check isn't enough by itself – correctly implementing a misunderstood requirement doesn't solve the problem.

Integration Testing

- focuses on interactions between modules

Integration is used to –

- *verify module interactions* – do the components cooperate correctly?
- *detect interface defects* – do components agree on how they are supposed to communicate?
- *validate API communication* – does our system communicate correctly with another system through its API?
- ensure database connectivity
- …

System Testing

- performed on the entire application
  - can reveal integration issues missed earlier
  - to ensure production readiness

System testing is used to –

- *validate complete workflows* – does the whole process work from start to finish?
- *verify system requirements* – does the complete system conform to its functional and non-functional requirements?
- *evaluate overall system behavior* – does the integrated system behave correctly as a whole?
  - including functional behavior, error handling, performance, security, compatibility, …

Who and when –

- typically done by QA teams in an environment that closely resembles production

What Makes a Good Test

- three components for defining a test
  - a description
  - a specific starting state and input
  - a known correct expected result
    - often called “expected output”, but it may be internal object state as well as return values
  - the implementation will contain a comparison between the actual result and the expected result (pass/fail)
- only one thing per test
  - each test should check one specific behavior
    - if a test fails, you know immediately what aspect of the behavior is wrong without needing to narrow things down farther
- test the edges too, not just the middle
  - “normal” input is usually the easiest case – and the least likely to reveal a bug
  - edge cases are things like empty input, boundary values, unusual-but-valid input

- `TableauColumn`

```
void deal(int n, Deck deck)        // deal initial cards into this column, drawn from deck
Card pickUp()                      // remove and return the top card
List<Card> pickUp(int n)           // remove and return the top n cards
void place(Card card)              // place a single card on top
void place(List<Card> cards)       // place a sequence of cards on top
void flipTopCard()                 // turn the top card face-up
boolean canPlace(Card card)        // is it legal to place this card on top?
boolean canPlace(List<Card> cards) // is it legal to place this sequence on top?
boolean isEmpty()                  // does this column have any cards?
```

- tests so far – what’s missing?

input	starting state	expected result	description
deal(4,deck)	empty column full deck	column has 4 cards	dealing cards into a column
pickUp()	column with several cards	correct card is returned number of cards in the column shrinks by 1	picking up a single card from a non-empty column
canPlace(card)	card is not legal to add	returns false	detect a card that isn’t legal to add
place(card)	a non-empty column card is legal to add	card is now on top	place a card on top of a non-empty column

So Many Test Cases…

Not all code deserves the same testing effort.

- test in proportion to risk – how easy is this to get wrong, and how costly if it is?
  - a one-line comparison or a trivial getter is low risk
    - usually fine to verify by reading – not worth a dedicated test
  - multi-condition logic like move validation, win-condition checks, anything combining several rules – high risk
    - write tests!
  - refactoring brings risk
    - can be worth testing even simple code to make sure nothing breaks

JUnit

- JUnit is a framework for automated testing
  - provides a standard way to write a test as a small method
  - automatically runs all your tests and reports pass/fail
  - removes tedious boilerplate common to every test
    - e.g. no need to hand-write `if` statements and print “pass”/”fail” yourself
- not limited to unit testing – “unit” vs “integration” a matter of scope rather than tools

JUnit Test Preview

```
@Test
void twoEqualCardsAreEqual() {
  Card c1 = new Card(Rank.QUEEN, Suit.HEARTS);
  Card c2 = new Card(Rank.QUEEN, Suit.HEARTS);
  boolean result = c1.equals(c2);
  assertTrue(result);
}
```

- three steps
  - set up the starting state and input for the test – create the two cards
  - generate the result – call `equals()`
  - compare the actual result to the expected result
    - `assertEquals`, `assertTrue`/`assertFalse`, …
- `@Test` marks the method as a test case
- features
  - test covers a single check – that two equivalent cards are detected
  - method has a descriptive name – gives you enough info to know what failed

Testing in the World of AI-Assisted Development

- AI tools are being increasingly used to generate and modify code – and can generate both code and tests
- human involvement in specifying tests helps rein in slop
  - defining tests requires developers to develop clear, complete specifications for code
  - prevents AI from rewriting tests when code changes
- testing provides a safety net when AI is to refactor legacy code bases
  - write tests to capture the current behavior, then use them after the refactoring to verify nothing broke
- testing is still essential, but there’s a shift in focus
  - unit tests serve as a contract in addition to catching small bugs
  - more focus on integration and end-to-end testing