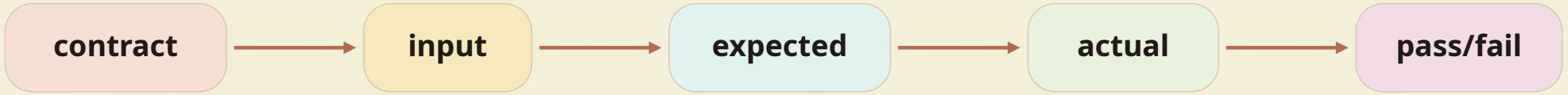
L09

AI Game Development & Computer Science Summer Program

Lecture 9: Testing, Debugging, and Program Design

Testing asks whether the program matches the contract

A test compares expected behavior with actual behavior.



Core question

- For this input, what should the function return?
- If actual differs from expected, the test has found useful information.

Write tests before or while you write code

Tests are design notes that can run.

```
const startsWithA = (name: string): boolean => {
  return name.trim().toUpperCase().startsWith("A");
};

expect(startsWithA("Ada Lovelace")).toBe(true);
```

Good first tests

- One normal case.
- One boundary case.
- One case that might break your assumption.

A test is strongest when you can explain why the expected result is correct.

Unit tests check one small piece

Start with pure functions because they are easiest to test.

Unit test

- Checks one function or small module.
- Uses direct inputs and expected outputs.
- Runs fast and often.

Integration test

- Checks pieces working together.
- Finds wiring and boundary bugs.
- Usually slower and broader.

Regression test

- Captures a bug after you fix it.
- Prevents the same mistake from returning.

Black-box and glass-box testing answer different questions

Use both when the risk is high.

Black-box

- Use the contract only.
- Do not care how the function is written.
- Best for checking user-visible behavior.

Glass-box

- Look inside the implementation.
- Choose tests that exercise branches and loops.
- Best for finding hidden paths.

Black-box asks, does it do the right thing? Glass-box asks, did we check the risky paths?

Edge cases expose hidden assumptions

A passing normal case is not enough.

Kind	Example for initials(name)	Why it matters
normal	Ada Lovelace	expected everyday use
single word	Ada	shorter structure
extra spaces	Ada Lovelace	messy input
empty		smallest input

Edge-case habit

- Minimum.
- Maximum.
- Empty.
- Repeated.
- Unexpected but possible.

Debug by shrinking the failing case

Make the bug smaller until the cause is visible.

Step 1

- reproduce

Step 2

- remove noise

Step 3

- trace one value

Step 4

- write the failing test

Debugging rule

Do not fix a bug you cannot reproduce.
Once reproduced, turn it into a test before editing.

Monolithic code hides bugs

Break a large workflow into named, testable helpers.

Hard to debug

- One long function.
- Many responsibilities.
- No clear place to test.

Easier to debug

- Small helpers with contracts.
- Each helper has tests.
- Integration is checked separately.

A useful abstraction gives you a smaller place to look when something fails.

Immutable updates make tests easier

Return a new value instead of quietly changing the old one.

```
type Task = { title: string; done: boolean };

const markDone = (task: Task): Task => {
  return { ...task, done: true };
};
```

Why it helps

- The input stays available for comparison.
- The output is a clear new value.
- Tests do not depend on hidden side effects.

Test question

- Did the returned task change while the original task stayed the same?

AI can brainstorm tests

Use AI for breadth. You still decide correctness.

Prompt

- Here is a function contract.
- List normal, boundary, invalid, and regression test cases.
- Do not write implementation code yet.

Your review

- Check whether each case follows the contract.
- Write expected outputs yourself.
- Add one case from your own trace.

Generated tests are useful only when the expected result is justified.

Studio build: tested formatter

Write tests first, then implement the smallest helper that passes them.

```
const displayName = (name: string): string => {
  // trim spaces, collapse blanks, and return "Anonymous" for
empty input
};

expect(displayName(" Ada  Lovelace ")).toBe("Ada Lovelace");
expect(displayName("")).toBe("Anonymous");
```

Build order

- Write the contract in one sentence.
- Pick three tests.
- Implement the simplest version.
- Add one failing edge case.
- Turn the fix into a regression test.

Lab plan: practice, build, apply

Use the main lab block to turn the concept into working evidence.

Guided practice

- Turn expected behavior into tests.
- Shrink a failing case.
- Separate pure helper bugs from state bugs.

Build target

- Write tests for a formatter or rule helper.
- Implement only enough to pass the tests.

Project time

- Add at least three P1 tests.
- Record one bug and its smallest reproduction.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Add a regression test for a fixed bug.
- Test a boundary value.
- Ask AI for test ideas and delete duplicates.

Done when

- The main lab still works.
- You can explain the tradeoff or edge case.
- You saved one proof object: test, trace, screenshot, or note.

P1: Game Rules Milestone

L03-L15

Checkpoint

Move on when every bug becomes a smaller, testable claim.

Exit ticket

- One unit test from a contract.
- One edge case and why it matters.
- One bug shrunk to a small input.
- One regression test for a fixed bug.
- One AI-suggested case you accepted or rejected.

Next class preview

- Mutability.
- Immutability.
- References and shared state.