Topic 06 - Testing & QA

COMPSCI 320 - Perretta

Overview

There is no single best way to test software.

Consider the **tradeoffs** of different **approaches** and **tools used**

- **Confidence gained** vs. **Initial cost** vs. **Repeated cost**
- "Traditional" automated testing (unit, integration, end-to-end)
- Randomized approaches
- Manual testing
- Testing for quality (non-functional) requirements

Tradeoffs to Consider

For a given approach/technique:

- How much **confidence** do we gain that our software is **correct**?
- What are the **initial costs** (human & computational)?
- What is the cost of **repeating** the approach?

Unit/Integration/End-to-End Testing

Widely-used automated testing approaches.

Unit testing:

- Test components in **isolation** (how much isolation?)
- Tests are **small** and run **fast**

Integration testing:

- Test interactions between components
- Relatively fast (depends on dependencies)

Both good for **thorough, granular** tests.

Unit vs. Integration Testing: Where's the line?

Consider the tradeoffs of mocking a dependency vs using the real thing.

Mock it if:

- It's expensive/limited (computation or $)
- Cumbersome to set up test environment
- Need to test specific/uncommon states (e.g., network error)

Too much mocking makes tests **brittle.**



End-to-End Testing

Test **full** start to finish paths through your application.

Pros:

- Closest automated substitute for a real user
- Gain confidence system components interact correctly

Cons:

- More challenging/time-consuming to write
- Take longer to run
- Too many possible paths to cover all

My Guidelines for Unit/Integration/End-to-End Testing

- Only mock when necessary
- Thorough Foundation of **unit/integration tests**
- High-importance paths? Add **end-to-end** tests

Aside: Regression Testing

When you find a bug:

- Write a test **before** fixing the bug:
  - should **fail before** fixing
  - should **pass after** fixing
- Leave a comment pointing to the issue/ticket for the bug



Aside: Test-Driven Development (TDD) - My Take

Central point: Write tests **before** you write the code.

Main benefits:

- Forces you to **actually write tests**
- Better understanding of the code when you do implement it

(There are a bunch of other things prescribed by TDD I won't get into).

Are Unit/Integration/End-to-End Testing Good Enough?

What if we want to test:

- Whether a webserver has a buffer overflow bug?
- A compiler or interpreter?

Randomized Testing

- Randomly generate program inputs
- Assess behavior with an **oracle**

Fuzz testing

- Oracle: Does the program crash?

Property-based testing

- Define less-specific traits of program output
  - E.g., "Output is always sorted"
  - E.g., "If input is divisible by 3, output is divisible by 3"

Randomized Testing

When are we done testing?

Effort required:

- Writing input generator (can be simple or complex)
- Computation: Run **many** inputs

Differential Testing

Compare the behavior of **different implementations**.

Good for **highly complex** but **standardized** programs.

Used for compilers/interpreters.

Check-In

What are the tradeoffs of the approaches we've discussed so far?

Tradeoffs:

- How much **confidence** do we gain that our software is **correct**?
- What are the **initial costs** (human & computational)?
- What is the cost of **repeating** the approach?

- Unit/integration testing?
- End-to-end testing?
- Fuzz testing?
- Property-based testing?
- Differential testing?

What About Quality/Non-Functional Requirements?

- Performance?
  - How do we test if our system is fast enough?
- Availability?
  - How do we test our system's ability to recover from unexpected errors?
- Usability & Accessibility?
  - How do we test whether users can actually use our application?

Another Creative Example

Cloudberry Kingdom (2013) is a procedurally-generated platformer video game.

How to make sure that levels are:

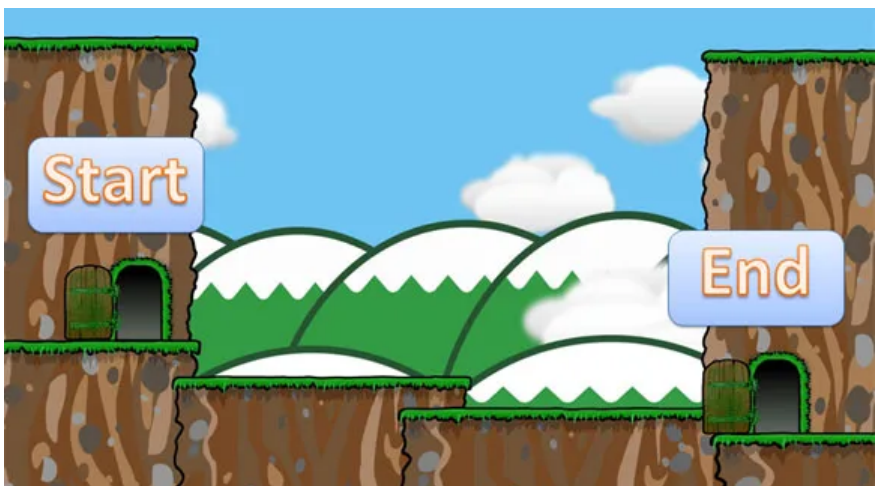
- Possible to complete?
- Interesting?
- Appropriately challenging?

https://www.gamedeveloper.com/design/how-to-make-insane-procedural-platformer-levels

Another Creative Example

Cloudberry Kingdom (2013) is a procedurally-generated platformer video game.

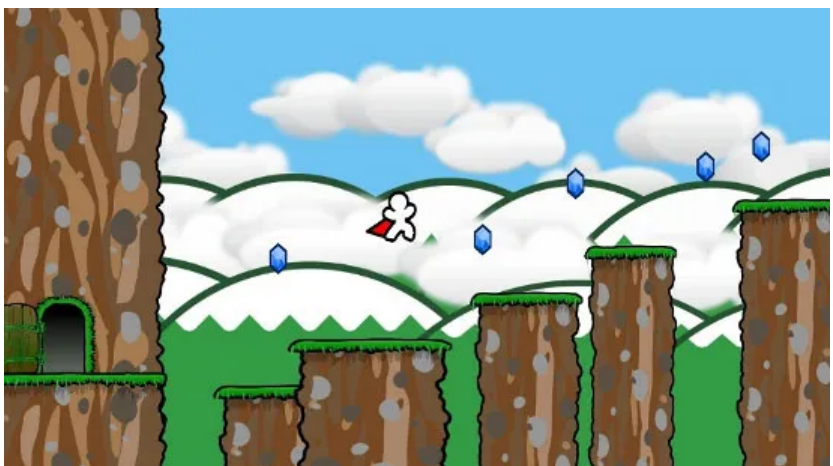
Not very interesting



Another Creative Example

Cloudberry Kingdom (2013) is a procedurally-generated platformer video game.

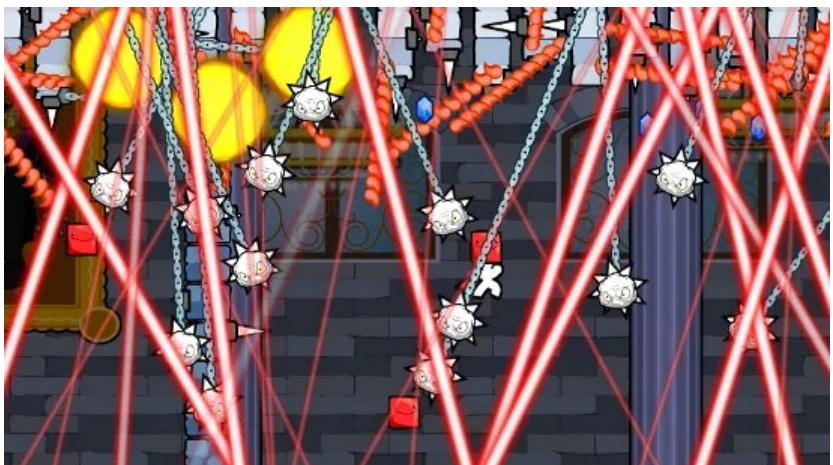
Fairly easy



Another Creative Example

Cloudberry Kingdom (2013) is a procedurally-generated platformer video game.

Way too difficult



Discussion: Tradeoffs in Different Settings?

How would our priorities/choice of tradeoffs change if we were:

- A one-person development team?
- An open-source project with many contributers?
- A small startup?
- A small team of volunteers for a nonprofit?
- A medium-sized company?
- A large company?

Conclusion

Every testing approach has **tradeoffs**.

Confidence gained vs. **Initial cost** vs. **Repeated cost**

Consider:

- What are the right approaches for your application?
- What are the right approaches for your team/resources?