CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal

Fall 2022

Testing

Representation exposure

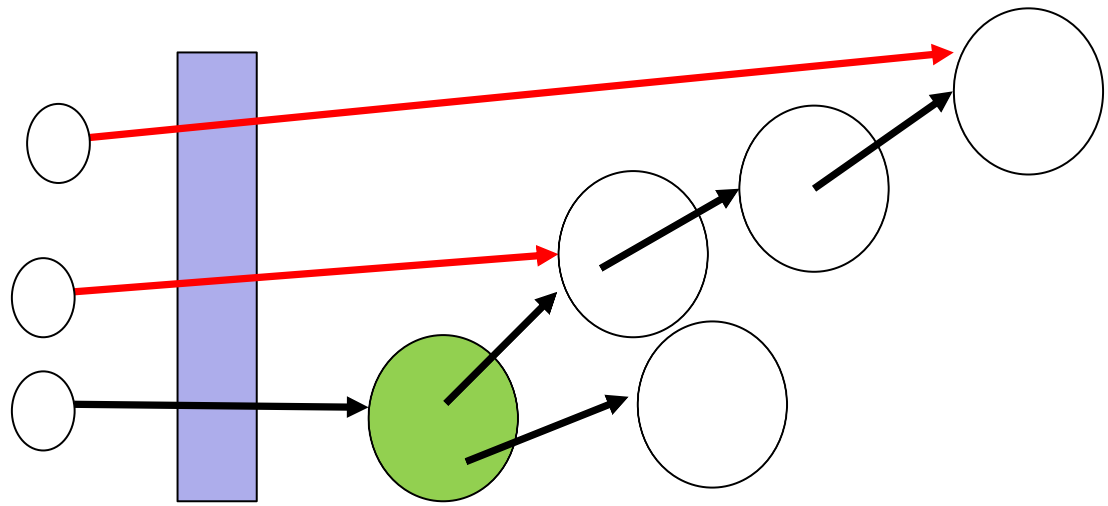
Consider this client code (outside the `CharSet` implementation):

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Representation exposure is external access to the rep
- Representation exposure is almost always **bad**
  - can cause bugs that will be **very hard to detect**
- Rule #1: Don't do it!
- Rule #2: If you do it, document it clearly and then feel guilty about it!

`private` is not enough

- Making fields **`private`** does *not* suffice to prevent rep exposure
  - see our example
  - issue is ***aliasing of mutable data outside the abstraction***



- So **`private`** is a hint to you: no aliases outside abstraction to references to mutable data reachable from **`private`** fields
- Three general ways to avoid representation exposure…

Avoiding rep exposure (way #1)

- One way to avoid rep exposure is to make copies of all data that cross the abstraction barrier
  - Copy in [parameters that become part of the implementation]
  - Copy out [results that are part of the implementation]
- Examples of copying (assume **`Point`** is a mutable ADT):

```
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = new Point(s.x,s.y);
        this.e = new Point(e.x,e.y);
    }
    public Point getStart() {
        return new Point(this.s.x,this.s.y);
    }
    …
```

Avoiding rep exposure (way #2)

- One way to avoid rep exposure is to exploit the immutability of (other) ADTs the implementation uses
  - aliasing is no problem if nobody can change data
    - have to mutate the rep to break the rep invariant

- Examples (assuming **`Point`** is an *immutable* ADT):

```
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = s;
        this.e = e;
    }
    public Point getStart() {
        return this.s;
    }
    …
```

Alternative #3

```
// returns: elts currently in the set
public List<Character> getElts() { // version 1
  return new ArrayList<Character>(elts);//copy out!
}

public List<Character> getElts() { // version 2
  return Collections.unmodifiableList(elts);
}
```

From the JavaDoc for **`Collections.unmodifiableList`**:

Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list… result in an **`UnsupportedOperationException`**.

The good news

```
public List<Character> getElts() { // version 2
  return Collections.unmodifiableList(elts);
}
```

- Clients cannot *modify (mutate)* the rep
  - cannot break the rep invariant
- (For long lists,) more efficient than copy out
- Uses standard libraries

The bad news

```
public List<Character> getElts() { // version 1
 return new ArrayList<Character>(elts);//copy out!
}
```

```
public List<Character> getElts() { // version 2
 return Collections.unmodifiableList(elts);
}
```

The two implementations do not do the same thing!

- both avoid allowing clients to break the rep invariant
- both return a list containing the elements

But consider:

```
xs = s.getElts();
s.insert('a');
xs.contains('a');
```

Version 2 is *observing* an exposed rep, leading to different behavior

Different specifications

Ambiguity of "returns a list containing the current set elements"

"returns a fresh mutable list containing the elements in the set *at the time of the call*"

versus

"returns read-only access to a list that the ADT *continues to update to hold the current elements in the set*"

A third spec weaker than both [but less simple and useful!]

"returns a list containing the current set elements. *Behavior is unspecified (!) if* client attempts to mutate the list or to access the list after the set's elements are changed"

Also note: Version 2's spec also makes changing the rep later harder

– only "simple" to implement with rep as a **`List`**

Suggestions

Best options for implementing `getElts()`

- if O(n) time is acceptable for relevant use cases, copy the list
  - safest option
  - best option for changeability
- if O(1) time is required, then return an unmodifiable list
  - prevents breaking rep invariant
  - clearly document that behavior is unspecified after mutation
  - ideally, write a your own unmodifiable view of the list that throws an exception on all operations after mutation
- if O(1) time is required and there is no unmodifiable version and you don't have time to write one, expose rep and feel guilty

Testing

How do we ensure correctness?

Best practice: use three techniques

1. **Tools**
   – e.g., type checking, @Override, libraries, etc.
2. **Inspection**
   – think through your code carefully
   – have another person review your code
3. **Testing**
   – usually >50% of the work in building software

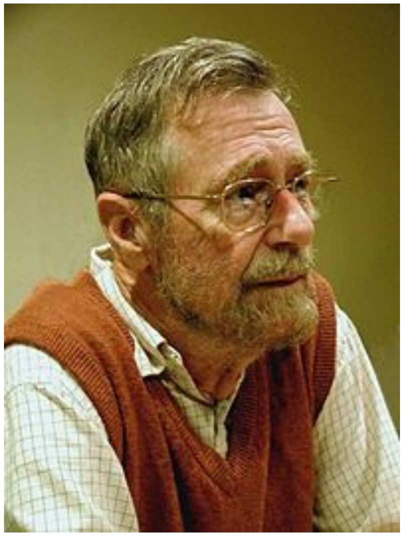
Each removes ~2/3 of bugs. Together >97%

What can you learn from testing?

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsgar Dijkstra

Notes on Structured Programming, 1970



Testing is *essential* but it is insufficient by itself

Only **reasoning** can prove there are no bugs. Yet...

How do we ensure correctness?

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
-Donald Knuth, 1977

<u>Trying it</u> is a surprisingly useful way to find mistakes!

No **single activity** or approach can guarantee correctness

We need tools **and** inspection **and** testing to ensure correctness

Why you will care about testing

In all likelihood, you will be expected to **test your own code**

- Industry-wide trend toward developers doing more testing
  - 20 years ago, we had large test teams
  - now, test teams are small to nonexistent

- Reasons for this change:
  1. easy to update products after shipping (users are testers)
  2. often lowered quality expectations (startups, games)
     - some larger companies want to be more like startups

This has positive and negative effects…

It’s hard to test your own code

Your **psychology** is fighting against you:

- confirmation bias
  - tendency to avoid evidence that you’re wrong
- operant conditioning
  - programmers get cookies when the code works
  - testers get cookies when the code breaks

You can avoid some effects of confirmation bias by

writing most of your tests before the code

Not much you can do about operant conditioning

Kinds of testing

- Testing field has terminology for different kinds of tests
  - we won't discuss all the kinds and terms
- Here are three orthogonal dimensions [so 8 varieties total]:
  - *unit* testing versus *system / integration / end-to-end* testing
    - one module's functionality versus pieces fitting together
  - *clear-box* testing versus *opaque-box / black-box* testing
    - did you look at the code before writing the test?
  - *specification* testing versus *implementation* testing
    - test only behavior guaranteed by specification or other behavior expected for the implementation

Unit Testing

- A unit test focuses on one class / module (or even less)
  - could write a unit test for a single method
- Tests a single unit in isolation from all others
- Integration tests verify that the modules fit together properly
  - usually don't want these until the units are well tested
    - i.e., unit tests come first

How is testing done?

Write the test

1) Choose input / configuration
2) Define the expected outcome

Run the test

3) Run with input and record the actual outcome
4) Compare *actual* outcome to *expected* outcome

What's So Hard About Testing?

"Just try it and see if it works..."

```
// requires: 1 ≤ x,y,z ≤ 100,000
// returns:  computes some f(x,y,z)
int func1(int x, int y, int z){…}
```

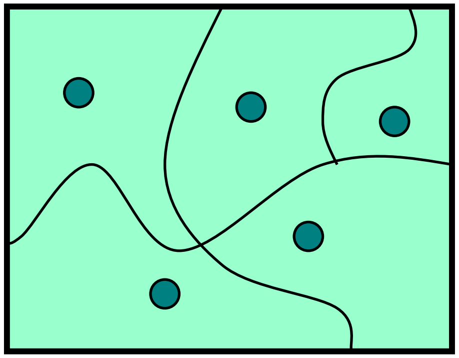
Exhaustive testing would require 1 quadrillion cases!

- impractical even for this trivially small problem

Key problem: choosing test suite

- Large/diverse enough to provide a useful amount of validation
- (Small enough to write in reasonable amount of time.)
  - need to think through the expected outcome
  - very few software projects have *too many* tests

Approach: Partition the Input Space



Ideal test suite:

Identify sets with "same behavior"

(actual and expected)

Test **at least** one input from each set

(we call this set a *subdomain*)

Two problems:

1. Notion of same behavior is subtle
   - Naive approach: execution equivalence
   - Better approach: revealing subdomains
2. Discovering the sets requires perfect knowledge
   - If we had it, we wouldn't need to test
   - Use heuristics to approximate cheaply

Naive Approach: Execution Equivalence

```
// returns:  x < 0      => returns -x
//           otherwise => returns x
int abs(int x) {
   if (x < 0) return -x;
   else       return x;
}
```

All x < 0 are execution equivalent:

- Program takes same sequence of steps for any x < 0

All x ≥ 0 are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite

Execution Equivalence Can Be Wrong

```
// returns:  x < 0      => returns -x
//           otherwise => returns x
int abs(int x) {
   if (x < -2) return -x;
   else        return x;
}
```

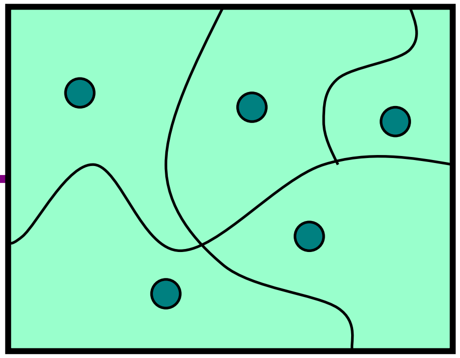
{-3, 3} does not reveal the error!

Two possible executions: x < -2 and x >= -2

Three possible behaviors:

- x < -2 OK, x = -2 or x= -1 (BAD)
- x >= 0 OK

Revealing Subdomains



- A *subdomain* is a subset of possible inputs

- A subdomain is *revealing* for error *E* if either:
  - *every* input in that subdomain triggers error E, *or*
  - *no* input in that subdomain triggers error E

- Need test at least one input from a revealing subdomain to find bug
  - if you test one input from every revealing subdomain for E, you are guaranteed to find the bug

- The trick is to *guess* revealing subdomains for **the errors present**
  - even though your reasoning says your code is correct, make educated guesses where the bugs might be

Testing Heuristics

- Testing is *essential* but difficult
  - want set of tests likely to reveal the bugs present
  - but we don't know where the bugs are
- Our approach:
  - split the input space into enough subsets (subdomains) such that inputs in each one are likely all correct or incorrect
  - can then take just one example from each subdomain
- Some heuristics are useful for choosing subdomains...

Heuristics for Designing Test Suites

A good heuristic gives:

- for all errors in some class of errors E:
  high probability that some subdomain is revealing for E
- not an *absurdly* large number of subdomains

Different heuristics target different classes of errors

- in practice, combine multiple heuristics
  - (we will see several)
- a way to think about and communicate your test choices

Specification Testing

Heuristic: Explore alternate cases in the specification

Procedure is opaque-box: specification visible, internals hidden

Example

```
// returns:  a > b => returns a
//           a < b => returns b
//           a = b => returns a
int max(int a, int b) {…}
```

3 cases lead to 3 tests

(4, 3) => 4 *(i.e. any input in the subdomain a > b)*
(3, 4) => 4 *(i.e. any input in the subdomain a < b)*
(3, 3) => 3 *(i.e. any input in the subdomain a = b)*

Specification Testing: Advantages

Process is not influenced by component being tested

- avoids psychological biases we discussed earlier
- can only do this for your own code if you **write tests first**

Robust with respect to changes in implementation

- test data need not be changed when code is changed

Allows others to test the code (rare nowadays)

Heuristic: Clear-box testing

Focus on features not described by specification

- control-flow details (e.g., conditions of “if” statements in code)
- performance optimizations
- alternate algorithms for different cases

Example: `abs` from before (different behavior < 0 and >= 0)

```
// @return |x|
int abs(int x) {
   if (x < 0) return -x;
   else       return x;
}
```

Clear-box Example

There are some subdomains that opaque-box testing won't catch:

```
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i=2; i*i <= x; i++) {
            if (x % i == 0)
             return false;
        }
        return true;
    } else {
        return primeTable[x];
    }
}
```

Clear Box Testing: [Dis]Advantages

- Finds an important class of boundaries
  - yields useful test cases
  - wouldn't know about `primeTable` otherwise

Disadvantage:
  - buggy code tricks you into thinking it's right once you look at it
    - (confirmation bias)
  - can end up with tests having same bugs as implementation
  - so also write tests **before** looking at the code

Clear-box Example

There are some subdomains that opaque-box testing won't catch:

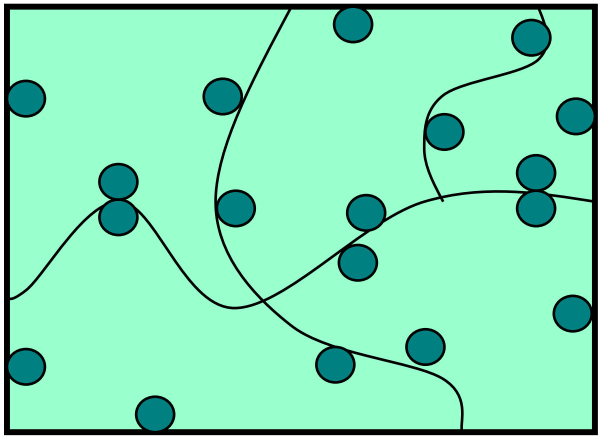
```
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i=2; i*i <= x; i++) {
            if (x % i == 0)
             return false;
        }
        return true;
    } else {
        return primeTable[x];
    }
}
```

Where is the bug?

Heuristic: Boundary Cases

Create tests at the boundaries between subdomains



Edges of the "main" subdomains have a high probability of revealing errors

- e.g., off-by-one bugs

Include one example on each side of the boundary

Also want to test the side edges of the subdomains…

Summary of Heuristics

Before you write the code (part of ”test-driven development”):

- split subdomains on boundaries appearing in the specification
- choose a test along both sides of each boundary

After you write the code:

- split further on boundaries appearing in the implementation

More next time…

On the other hand, don't confuse *volume* with *quality* of tests

- look for revealing subdomains
- want tests in every revealing subdomain not **just** lots of tests

Testing Tools

- Modern development ecosystems have built-in support for testing
- Your homework introduces you to Junit
  - standard framework for testing in Java
- Continuous integration
  - ensure tests pass **before** code is submitted
- You will see more sophisticated tools in industry
  - libraries for creating mock implementations of other modules
  - automated tools to test on every platform
  - automated tools to find severe bugs (using AI)
  - …