

Exceptions and Testing



6.170
Michael Ernst
Saman Amarasinghe

Generic Types

Bugfix to the last lecture

```
public class List<T> {
    private T lst[];
    public void add(int, T) { ... }
    public T get(int) { ... }
    ...
}

List<Integer> lst = new List<Integer>();
Integer fst = lst.get(0);
```

How to handle abnormal situations?

```
public static int Search(int[] lst, int key)

requires: none
modifies: none
effects: none
returns: some i such that lst[i] = key if such an i exists,
        otherwise -1
```

Returning $\{(insertion\ point), -1\}$ is very ugly, and an invitation to bugs and confusion.

How to handle abnormal situations?

```
public static int Search(int[] lst, int key)

requires: key is in lst
modifies: none
effects: none
returns: i where lst[i] = key
```

*Partial procedures make the world more complicated.
Who is going to check if key is in a?
Too much burden on the client.*

Use an exception

```
public static int Search(int[] lst, int key)

requires: none
modifies: none
effects:
throws: NoSuchElementException if key is not in lst
returns: some i such that lst[i] = key if such an i exists
```

How to use an Exception

Declaration:

```
public static int Search(int[] lst, int key)
    throws NoSuchElementException {
    for(int i = 0; i < lst.length; i++)
        if(lst[i] == k) return i;
    throw new NoSuchElementException();
}
```

Use:

```
try {
    j = s.Search(mylist, k);
} catch (NoSuchElementException e) {
    Handle the exception
}
... continue for both normal and exceptional cases
```

Catching Exceptions

Caught by catch associated with nearest enclosing try

If no such catch

Exception propagated up call stack

If not caught at all

Program terminates

Two Kinds of Exceptions

Checked exceptions

E.g., class `MissingException` extends `Exception`

Compiler error unless

There exists a catch clause, or
Caller declared to throw that exception

∴ There is guaranteed to be a dynamically enclosing catch

Unchecked exceptions

E.g., class `ArithmeticException` extends `RuntimeException`

Compiler doesn't complain

Rule of thumb

Stick to checked exceptions most of the time

Use unchecked exceptions to mean failure

Expect program to terminate

Why Catch Exceptions Locally?

Failure to catch exceptions violates modularity

Call chain: `A` → `IntegerSet.insert` → `IntegerList.insert`

`IntegerList.insert` throws an exception

Implementer of `IntegerSet.insert` knows how list is being used

Implementor of `A` may not even know that `IntegerList` exists

Procedure up the line may think that it is handling an exception raised by a different call

Even if exception is better handled up a level

May be better to catch it and throw it again ("chaining")

Makes it clear to reader of code that it was not an omission

Exceptions in Review

Use an exception when

Used in a broad or unpredictable context

Checking the condition is feasible

Use a precondition when

Checking would be prohibitive

E.g., requiring that a list be sorted

Used in a narrow context in which calls can be checked

Preconditions should be avoided because

Caller may violate precondition

Program can fail in uninformative or dangerous way

Want program to fail as early as possible

Exceptions in Review, cont.

Use checked exceptions most of the time

Handle exceptions sooner rather than later

Don't think of them as errors

A program structuring mechanism

Used for exceptional (unpredictable) circumstances

Documentation standards and conventions

Will be covered in recitation

Also see Bloch's *Effective Java*

Building Quality Software

What Impacts the Software Quality?

External

Correctness *Does it do what it suppose to do?*

Reliability *Does it do it accurately all the time?*

Efficiency *Does it do with minimum use of resources?*

Integrity *Is it secure?*

Internal

Portability *Can I use it under different conditions?*

Maintainability *Can I fix it?*

Flexibility *Can I change it or extend it or reuse it?*

Quality Assurance

The process of uncovering problems and improving the quality of software.

Testing is a major part of QA.

The Phases of Testing

Unit Testing

Is each module doing what it is supposed to do?

Integration Testing

Do you get the expected results when the parts are put together work?

Validation Testing

Does the program satisfy the requirements

System Testing

Does it work within the overall system

Unit Testing

A test is at the level of a method/class/interface

Check if the implementation matches the specification.

Black box testing

Choose test data *without* looking at implementation

Glass box (white box) testing

Choose test data *with* knowledge of implementation

How is testing done?

Basic steps of a test

- 1) Choose input data/configuration
- 2) Define the expected outcome
- 3) Run program/method against the input and record the results
- 4) Examine results against the expected outcome

What's So Hard About Testing ?

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

```
int procl(int x, int y, int z)
// requires: 1 <= x,y,z <= 1000
// effects:  computes some f(x,y,z)
```

Exhaustive testing would require 1 billion runs!

Sounds totally impractical

Could see how input set size would get MUCH bigger

Key problem: choosing test suite (set of partitions of inputs)

Small enough to finish quickly

Large enough to validate the program

Approach: Partition the Input Space

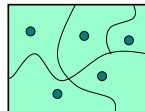
Input space very large, program small

==> behavior is the "same" for sets of inputs

Ideal test suite:

Identify sets with same behavior

Try one input from each set



Two problems

1. Notion of **the same behavior** is subtle
 - Naive approach: execution equivalence
 - Better approach: revealing subdomains
2. Discovering the sets requires perfect knowledge
 - Use heuristics to approximate cheaply

Naive Approach: Execution Equivalence

```
int abs(int x) {
// returns:  x < 0   => returns -x
//           otherwise => returns 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 $x \geq 0$ are execution equivalent

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

Why Execution Equivalence Doesn't Work

Consider the following buggy code:

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

Two executions:

x < -2 x >= -2

Three behaviors:

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

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

Revealing Subdomain Approach

“Same” behavior depends on specification

Say that program has “same behavior” on two inputs if

- 1) gives correct result on both, or
- 2) gives incorrect result on both

Subdomain is a subset of possible inputs

Subdomain is revealing for an error, E, if

- 1) Each element has same behavior
- 2) If program has error E, it is revealed by test

Trick is to divide possible inputs into sets of revealing subdomains for various errors

Example

For buggy abs, what are revealing subdomains?

```
int abs(int x) {  
  if (x < -2) return -x;  
  else return x;  
}
```

~~{-1} {-2} {-2, -1} {-3, -2, -1}~~

{-2, -1}

Which is best?

Heuristics for Designing Test Suites

A good heuristic gives:

few subdomains

∀ errors e in some class of errors E,

high probability that some subdomain is revealing for e

Different heuristics target different classes of errors

In practice, combine multiple heuristics

Black Box Testing

Heuristic: Explore alternate paths through specification

Procedure an interface is a **black box**, internals hidden

Example

```
int max(int a, int b)  
  // effects: a > b => returns a  
  //           a < b => returns b  
  //           a = b => returns a
```

3 paths, so 3 test cases:

(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)

Black Box Testing: Advantages

Process not influenced by component being tested

Assumptions embodied in code not propagated to test data.

Robust with respect to changes in implementation

Test data need not be changed when code is changed

Allows for independent testers

Testers need not be familiar with code

More Complex Example

Write test cases based on paths through the specification

```
int find(int[] a, int value) throws Missing
// returns: the smallest i such
//          that a[i] == value
// throws:  Missing if value not in a
```

Two obvious tests:

([4, 5, 6], 5) => 1

([4, 5, 6], 7) => throw Missing

Have I captured all the paths?

([4, 5, 5], 5) => 1

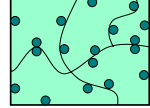
Must hunt for multiple cases in effects or requires

Heuristic: Boundary Testing

Create tests at the edges of subdomains

Why do this?

- off-by-one bugs
- forget to handle empty container
- overflow errors in arithmetic
- program does not handle aliasing of objects



Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors

Boundary Testing

To define boundary, must define adjacent points

One approach:

Identify basic operations on input points

Two points are adjacent if one basic operation away

A point is isolated if can't apply a basic operation

Example: list of integers

Basic operations: create, append, remove

Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>

Isolated point: [] (can't apply remove integer)

Point is on a boundary if either

There exists an adjacent point in different subdomain

Point is isolated

Other Boundary Cases

Arithmetic

Smallest/largest values

Zero

Objects

Null

Circular

Same object passed to multiple arguments (aliasing)

boundary cases: Arithmetic Overflow

```
public int abs(int x)
// returns: |x|
```

Tests for abs

what are some values or ranges of x that might be worth probing?

x < 0 (flips sign) or x ≥ 0 (returns unchanged)

around x = 0 (boundary condition)

Specific tests: say x = -1, 0, 1

How about...

```
int x = -2147483648; // this is Integer.MIN_VALUE
System.out.println(x<0); // true
System.out.println(Math.abs(x)<0); // also true!
```

From Javadoc for Math.abs:

Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative

boundary cases: duplicates and aliases

```
<E> void appendList(List<E> src, List<E> dest) {
// modifies:  src, dest
// effects:   removes all elements of src and
//            appends them in reverse order to
//            the end of dest
//
// while (src.size()>0) {
//     E elt = src.remove(src.size()-1);
//     dest.add(elt);
// }
}
```

What happens if src and dest refer to the same thing?

This is aliasing – often forgotten

Watch out for shared references in inputs

Glass-box Testing

Goal:

Ensure test suite covers (executes) all of the program
Measure quality of test suite with % coverage

Assumption:

high coverage →
(no errors in test suite output
→ few mistakes in the program)

Focus: features not described by specification

Control-flow details
Performance optimizations
Alternate algorithms for different cases

glass-box motivation

There are some subdomains that black-box testing won't give:

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

Important transition around $x = \text{CACHE_SIZE}$

Glass Box Testing: Advantages

Insight into test cases

Which are likely to yield new information

Finds an important class of boundaries

Consider `CACHE_SIZE` in `isPrime` example

Need to check numbers on each side of `CACHE_SIZE`

`CACHE_SIZE-1`, `CACHE_SIZE`, `CACHE_SIZE+1`

If `CACHE_SIZE` is mutable, we may need to test with different `CACHE_SIZE`'s

Glass-box Challenges

Definition of all of the program

What needs to be covered?

Options:

Statement coverage
Decision coverage
Loop coverage
Condition/Decision coverage
Path-complete coverage

↑ increasing
number of
Cases
(more or
less)

100% coverage not always reasonable target

100% may be unattainable (dead code)
High cost to approach the limit

pragmatics: regression testing

Whenever you find and fix a bug

Store input that elicited that bug
Store correct output
Put into test suite

Why is this a good idea

Helps to populate test suite with good tests
Protects against reversions that reintroduce bug
Arguably is an easy error to make (happened at least once, why not again?)

Rules of Testing

First rule of testing: *Do it early and do it often*

Best to catch bugs soon, before they have a chance to hide.
Automate the process if you can
Regression testing will save time.

Second rule of testing: *Be systematic*

If you randomly thrash, bugs will hide in the corner until you're gone
Writing tests is a good way to understand the spec

Think about revealing domains and boundary cases
If the spec is confusing → write more tests

Spec can be buggy too

Incorrect, incomplete, ambiguous, and missing corner cases

When you find a bug → fix it first and then write a test for it

summary

Testing matters

You need to convince others that module works

Catch problems earlier

Bugs become obscure beyond the unit they occur in

Don't confuse volume with quality of test data

Can lose relevant cases in mass of irrelevant ones

Look for revealing subdomains

Choose test data to cover

Specification (black box testing)

Code (glass box testing)

Testing can't generally prove absence of bugs

But can increase quality by reducing the bugs