topics for today

introduction
  - two problems, aims of testing

steps in testing
  - correctness criterion: writing a spec
  - defining the space, esp. inductively
  - generating cases, partitions
  - checking results, 3 approaches

- decide what method under test should do
- define test space
- generate test cases
- check results
introduction
two problems

often confused, but very different
• (a) problem of finding bugs in defective code
• (b) problem of showing absence of bugs in good code

approaches
• testing: good for (a), occasionally (b)
• reasoning: good for (a), also (b)

theory and practice
• for both, you need grasp of basic theory
• good engineering judgment essential too
aims of testing

what are we trying to do?
- find bugs as cheaply and quickly as possible

reality vs. ideal
- ideally, choose one test case that exposes a bug and run it
- in practice, have to run many cases that “fail”

in practice, conflicting desiderata
- increase chance of finding bug
- decrease cost of test suite (cost to generate, cost to run)

essential characteristics
- modularity: no dependence of driver on internals of unit being tested
- automation: must be able to run (and check results) without manual effort
the correctness criterion
what does correct mean?

ADT works if

・ each method works -- ie, meets specification

general form of spec: pre and post conditions

\[ T \text{ } m \text{ } (T_1 \text{ } a_1, \text{ } T_2 \text{ } a_2, \text{ } ...) \]

\textbf{requires} some constraint involving args this, a1, a2, ...

\textbf{returns} some constraint involving result and args

interpretation

・ if pre holds on call, then method must ensure that post holds on return
・ precondition obligates caller, so strengthening pre \textbf{weaken} the spec
・ postcondition can be \textbf{non-deterministic}, allowing more than one result
・ implementation can \textbf{weaken} pre and \textbf{strengthen} post
from java.util.Collections, simplified

```java
static <T> int binarySearch(List<T> list, T key)
requires list is sorted
returns list.get(result).equals(key) || result=-1 && !list.contains(key)
```

note

• precondition: if not sorted, any behaviour is possible
• non-determinism in postcondition (where?)

why the precondition and non-determinism?

• weakening spec gives implementation freedom
• allows simpler and faster code (why?)

testing implications?
space of values

what are the cases a method should work for?

\[ T \text{ m} (T_1 a_1, T_2 a_2, \ldots) \]
\textbf{requires} ... \textbf{returns} ...

cases are defined by space

- all values \( T_i \) of each argument \( a_i \)
- in every combination
- that satisfy the precondition

fundamental problem of testing

- space is generally too big to cover exhaustively
- Intel Teraflop Research Chip can do 2TFlop (\( 2 \times 10^{12} \) floating point ops/s)
- arithmetic op on 32-bit float: \(~100\) days at 2TFlop
generating inputs
how do we enumerate the values of an argument?

- integer: 0, 1, -1, 2, -2, 3, -3, ...
- but what about lists? trees? expressions? sets?

a constructive view of datatype definitions

- recall from last lecture
  \[
  \text{List}<E> = \text{NonEmptyList}(E, \text{List}<E>) \cup \text{EmptyList}()
  \]
- can view as **recipe for generating values**
  \[
  \text{ls} = \{\text{EmptyList}()\}; \ \text{used} = \{\};
  \text{while} \ (\text{true}) \ {\}
  \quad \text{choose} \ l \in \text{ls} \setminus \text{used};
  \quad \text{used} = \text{used} \cup \{l\};
  \quad \text{for} \ (e: E) \ {\}
  \quad \text{ls} = \text{ls} \cup \{\text{NonEmptyList}(e,l)\} \}
  \]
inductive generators

even if not a recursive type, can still write inductive definition

classifying methods

• recall: constructors, producers, observers

• identify: generators ⊆ constructors ∪ producers

• a generator set is sufficient to generate all values

text

example: set

• constructors: emptySet

• producers: add, remove

• observers: isEmpty, contains

• generators: emptySet, add

\[ \text{Set}<E> = \text{add}(E, \text{Set}<E>) \cup \text{emptySet}(E) \]
selecting inputs

full space is too large
' so need to select inputs

idea: partitioning
' partition space for each argument into classes*
' pick one value from each class, and take cross product

example: java.math.BigInteger.multiply (BigInteger val)
' two arguments, this and val, drawn from BigInteger
' partition BigInteger into BigNeg, SmallNeg, -1, 0, 1, SmallPos, BigPos
' pick a value from each class, eg -2^{65}, -123, -1, 0, 1, 123, 2^{65}
' test the 7 \times 7 = 49 combinations

*ie, equivalence classes (subsets), not Java classes! often these classes are called “partitions”, but the partition is actually the set of classes
relationships

**partition the product**

- so far, partition the classes and then take product
- when relationship matters, just partition the product

**example:** `static int java.Math.max(int a, int b)`

- product is `int \times int`
- partition into: `a < b, a = b, a > b`
- pick value from each product class, eg `(1, 2), (1, 1), (2, 1)`

**example:** `Set intersect (Set that)`

- product is `Set \times Set`
- partition into: `this \subseteq that, that \subseteq this, this \cap that = \emptyset, this \cap that \neq \emptyset^*`

^*each constraint implicitly conjoined with negation of preceding constraints
additional strategies

get partitions from structure of postcondition

› example: `java.lang.Math.abs`

```java
public static long abs(long a)
```

Returns the absolute value of a `long` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Note that if the argument is equal to the value of `Long.MIN_VALUE`, the most negative representable `long` value, the result is that same value, which is negative.

combine partitions

› take partitions formed by more than one approach
› basic: “cover” all the partitions
› more ambitious: attempt to satisfy all combinations of partitions

example: `Set.intersect`

› one method gives class `this ⊆ that`, another gives classes `∅`, singleton, small
› basic: cover `this ⊆ that` with one case, `this = ∅`, `that = singleton` with another
› advanced: generate cases for `this ⊆ that` with values from `∅`, singleton, small
checking results
how to check results

now we have the inputs, but how to check the outputs?

• problem: an ADT’s abstract values aren’t visible!

3 approaches

• use toString to get abstract value as a string
• compare to known value using equals method
• apply observers to determine value indirectly
approach 1: toString

**approach:** use `toString` to get abstract value as a string

- example: `assertEquals("{1,2}", new Set().add(1).add(2).toString())`
- pros: fairly direct check that abstract value is right
- cons: gets tedious, fragile (breaks when type is enriched)

**why it’s fragile**

- `toString` is not usually specified precisely
- if type is modified or enriched, very likely that format of string changes
- tests that previously succeeded will now fail

**note**

- abstraction function maps rep to abstract value, not to a string!
- this value is not directly represented -- that’s why it’s abstract!
- good `toString` is essential for ensuring readable reports when unit test fails
approach 2: abstract equality

**approach:** compare to known value using `equals` method

• example: `assertEquals(set12, new Set().add(1).add(2))`

• pros: easy to automate, scalable

• cons: no good if method under test is non-deterministic

**note:** must write `equals` method!

• `equals` inherited from `Object` is no good: compares references

• need to compare **contents** of objects

• look at `equals` methods in sample code from last lecture

• read Bloch, Items 6 and 7
approach 3: observers

**approach:** apply observers to determine value indirectly

- example: `assertTrue (new Set().add(1).add(2).contains(1))`
- pros: simple approach
- cons: each check is only partial, might require many checks

**adequate observers**

- set of observers is adequate if any pair of values can be distinguished
- eg, `{contains, size}` is adequate for `Set`
- if no adequate set, cannot check results completely

**example:** checking values of `Expr` (from last lecture)

- observers: just `eval`
- not adequate; have to apply both `subst` and `eval`
checking rep

so far
• only checking that abstract value is right
• how to check rep value?

dilemma
• can’t check rep value in test
• would expose rep (and make test dependent on choice of implementation)

strategy
• write assertion for checking rep invariant
• insert in code at end of every method
• gets checked automatically during tests
summary

main ideas

• test space as product of argument values
• inductive definitions of types give generation recipes
• partitioning, a form of spec-based coverage
• checking abstract values by (necessarily) indirect means

a good test suite

• takes effort and careful thought to develop
• but can save a lot of time in the long run