Understanding ADTs

CSE 331
University of Washington

Michael Ernst

Ways to get your design right

- The hard way
  - Start hacking
  - When something doesn't work, hack some more
    - How do you know it doesn't work?
    - Need to reproduce the errors your users experience
  - Apply caffeine liberally

- The easier way
  - Plan first (specs, system decomposition, tests, ...)
  - Less apparent progress upfront
  - Faster completion times
  - Better delivered product
  - Less frustration

Ways to verify your code

The hard way: hacking
- Make up some inputs
- If it doesn't crash, ship it
- When it fails in the field, attempt to debug

An easier way: systematic testing
- Reason about possible behaviors and desired outcomes
- Construct simple tests that exercise all behaviors

Another way that can be easy: reasoning
- Prove that the system does what you want
  - Rep invariants are preserved
  - Implementation satisfies specification
- Proof can be formal or informal (we will be informal)
- Complementary to testing

Uses of reasoning

Goal: correct code

- Verify that rep invariant is satisfied
- Verify that the implementation satisfies the spec
- Verify that client code behaves correctly

  Assuming that the implementation is correct

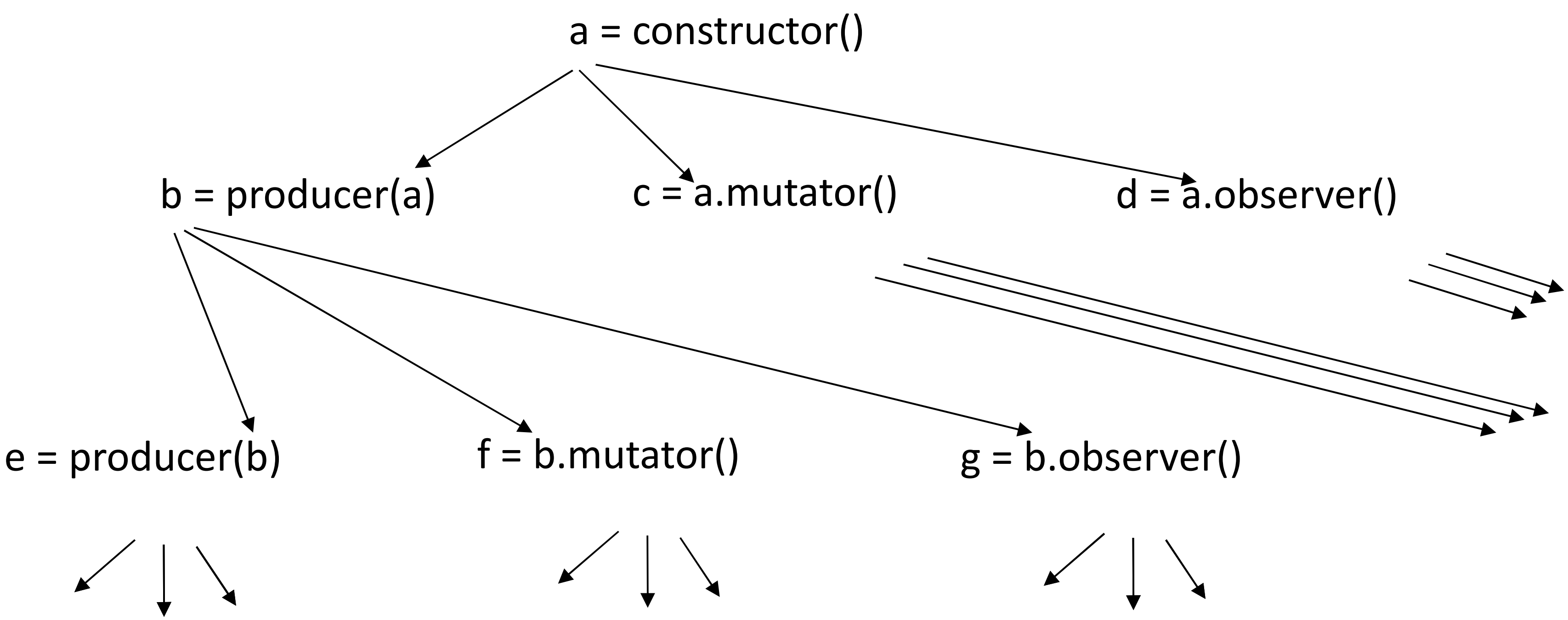
Goal: Demonstrate that rep invariant is satisfied

- Exhaustive testing
  - Create every possible object of the type
  - Check rep invariant for each object
  - Problem: impractical
- Limited testing
  - Choose representative objects of the type
  - Check rep invariant for each object
  - Problem: did you choose well?
- Reasoning
  - Prove that all objects of the type satisfy the rep invariant
  - Sometimes easier than testing, sometimes harder
  - Every good programmer uses it as appropriate

All possible objects (and values) of a type

- Make a new object
  - constructors
  - producers
- Modify an existing object
  - mutators
  - observers, producers (why?)
- Limited number of operations, but infinitely many objects
  - Maybe infinitely many values as well

Examples of making objects



Infinitely many possibilities

We cannot perform a proof that considers each possibility case-by-case

Solution: induction

Induction: technique for proving infinitely many facts using finitely many proof steps

For constructors ("basis step")

- Prove the property holds on exit

For all other methods ("inductive step")

- Prove that:
  - if the property holds on entry, then it holds on exit

If the basis and inductive steps are true:

- There is no way to make an object for which the property does not hold
- Therefore, the property holds for all objects

A counter class

```
// spec field: count
// abstract invariant:  count ≥ 0
class Counter {
  // counts up starting from 0
  Counter();
  // returns a copy of this counter
  Counter clone();
  // increments the value that this represents:
  // count_post = count_pre + 1
  void increment();
  // returns count
  BigInteger getValue();
}
```

Is the abstract invariant satisfied by these method specs?

Proof by contradiction: where was the invariant first violated?

Inductive proof

- Base case: invariant is satisfied by constructor
- Inductive case:
  - If invariant is satisfied on entry to clone, then invariant is satisfied on exit
  - If invariant is satisfied on entry to increment, then invariant is satisfied on exit
  - If invariant is satisfied on entry to getValue, then invariant is satisfied on exit
- Conclusion: invariant is always satisfied

Inductive proof that $x+1 > x$

ADT: the natural numbers (non-negative integers)

- constructor: 0 (zero)
- producer: succ (successor: succ(x) = x+1)
- mutators: none
- observers: value

Axioms:

1. $succ(0) > 0$
2. $(succ(i) > succ(j)) \Leftrightarrow i > j$

Goal: prove that for all natural numbers x, $succ(x) > x$

Possibilities for x:

1. x is 0
   - $succ(0) > 0$ — axiom #1
2. x is succ(y) for some y
   - $succ(y) > y$ — assumption
   - $succ(succ(y)) > succ(y)$ — axiom #2
   - $succ(x) > x$ — def of $x = succ(y)$

Outline for remainder of lecture

1. Prove that rep invariant is satisfied
2. Prove that client code behaves correctly

   (Assuming that the implementation is correct)

CharSet abstraction

```
// Overview: A CharSet is a finite mutable set of chars.

// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: this_post = this_pre U {c}
public void insert (char c);

// modifies: this
// effects: this_post = this_pre - {c}
public void delete (char c);

// returns: (c ∈ this)
public boolean member (char c);

// returns: cardinality of this
public int size ( );
```

Implementation of CharSet

```
// Rep invariant:  elts has no nulls and no duplicates
List<Character> elts;

public CharSet() {
  elts = new ArrayList<Character>();
}
public void delete(char c) {
  elts.remove(new Character (c));
}
public void insert(char c) {
  if (! member(c))
    elts.add(new Character(c));
}
public boolean member(char c) {
  return elts.contains(new Character(c));
}
…
```

Proof of CharSet representation invariant

Rep invariant: elts has no nulls and no duplicates

Base case: constructor

```
public CharSet() {
  elts = new ArrayList<Character>();
}
```

This satisfies the rep invariant

Inductive step:

For each other operation:

Assume rep invariant holds before the operation

Prove rep invariant holds after the operation

Inductive step, `member`

Rep invariant: elts has no nulls and no duplicates

```
public boolean member(char c) {
  return elts.contains(new Character(c));
}
```

`contains` doesn't change **`elts`**, so neither does **`member`**.
Conclusion: rep invariant is preserved.

Why do we even need to check **`member`**?

- After all, the specification says that it does not mutate set.

Reasoning must account for all possible arguments

- It's best not to involve the specific values in the proof

Inductive step, `delete`

Rep invariant: elts has no nulls and no duplicates

```
public void delete(char c) {
  elts.remove(new Character(c));
}
```

`List.remove` has two behaviors:

- leaves **`elts`** unchanged, or
- removes an element.

Rep invariant can only be made false by adding elements.

Conclusion: rep invariant is preserved.

Inductive step, `insert`

Rep invariant: elts has no nulls and no duplicates

```
public void insert(char c) {
  if (! this.member(c))
    elts.add(new Character(c));
}
```

If $\mathbf{c} \in \mathbf{elts}_{pre}$:

elts is unchanged $\Rightarrow$ rep invariant is preserved

If $\mathbf{c} \notin \mathbf{elts}_{pre}$:

new element is not null or a duplicate $\Rightarrow$ rep invariant is preserved

Reasoning about mutations to the rep

- Inductive step must consider all possible changes to the rep
  - A possible source of changes: representation exposure
  - If the proof does not account for this, then the proof is invalid
- An important reason to protect the rep:
  - Compiler can help verify that there are no external changes

Induction for reasoning about uses of ADTs

- Induction on specification, not on code
- Abstract values (e.g., specification fields) may differ from concrete representation
- Can ignore observers, since they do not affect abstract state
  - How do we know that?
- Axioms
  - specs of operations
  - axioms of types used in overview parts of specifications

LetterSet (case-insensitive character set)

```
// A LetterSet is a mutable finite set of characters.
// No LetterSet contains two chars with the same lower-case representation.

// effects: creates an empty LetterSet
public LetterSet ( );

// Insert c if this contains no other char with same lower-case representation.
// modifies: this
// effects: this_post = if (∃c_1 ∈ this_pre s.t.  toLowerCase(c_1) = toLowerCase(c) )
//                          then this_pre
//                          else this_pre U {c}
public void insert (char c);

// modifies: this
// effects: this_post = this_pre - {c}
public void delete (char c);

// returns:  (c ∈ this)
public boolean member (char c);

// returns:  |this|
public int size ( );
```

Attempt #1

Goal: prove that no LetterSet contains upper- and lower-case versions of a letter

Property $P(X) = \neg\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) = \text{toLowerCase}(c_2)]$

Prove $P(S)$; that is: $\neg\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) = \text{toLowerCase}(c_2)]$

How might S have been made?

$\xrightarrow{\text{constructor}} S$ Base case

$T \xrightarrow{\text{T.insert(c)}} S$ Inductive case

$T \xrightarrow{\text{T.delete(c)}} S$ Inductive case

Goal: prove no case-insentitive duplicates

Property P(X) = $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

Prove P(S); that is: $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

How might S have been made?

Consider two possibilities for how S was made: by the constructor, or by `insert`

Base case: S = { }, (S was made by the constructor):

property holds (vacuously true)

Inductive case (S was made by a call of the form "T.insert(c)"):

Show: P(S), that is, $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

where S = T.insert(c)

= "if ($\exists c_5 \in$ T s.t. toLowerCase(c_5) = toLowerCase(c))
then T **else** T U {c}"

The value for S came from the specification of insert, applied to T.insert(c):

// modifies: this

// effects: this_{post} = ***if (***$\exists c_1 \in$***S s.t. toLowerCase(***c_1***) = toLowerCase(c))***
then $\textit{\textbf{this}}_{pre}$
else $\textit{\textbf{this}}_{pre}$ ***U {c}***

public void insert (char c);

(Inductive case is continued on the next slide.)

Goal: no case-insensitive duplicates.

Inductive case: S = T.insert(c)

Goal (from previous slide):

Assume: P(T), that is: $\neg\exists c_3, c_4 \in T$ [toLowerCase(c_3) = toLowerCase(c_4)]

Show: P(S), that is: $\neg\exists c_1, c_2 \in S$ [toLowerCase(c_1) = toLowerCase(c_2)]

where S = T.insert(c)

= "if ($\exists c_5 \in T$ s.t. toLowerCase(c_5) = toLowerCase(c))

then T **else** T U {c}"

Consider the two possibilities for S (from "if ... then T else T U {c}"):

1. If S = T, then we have not introduced a duplicate (duh)
   and T had no duplicate to begin with
2. If S = T U {c}, then P(S) holds because of the if statement in the specification and the definition of union

Therefore, P(S) holds

Attempt #2

Goal: prove that no LetterSet contains upper- and lower-case versions of a letter

Property $P(X) = \neg\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) = \text{toLowerCase}(c_2)]$

Prove $P(S)$; that is: $\neg\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) = \text{toLowerCase}(c_2)]$

Use induction on the size of S.

How big is S?

Size 0	Base case
Size >0	Inductive case

Goal: prove no case-insentitive duplicates

Property $P(X) = \neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

Prove $P(S)$; that is: $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

How might S have been made?

Consider three possibilities for how S was made: by the constructor, or by **`insert`**, or by **`delete`**

Base case: S = { }, (S was made by the constructor):

property holds (vacuously true)

Inductive case (S was made by a call of the form "T.insert(c)"):

Assume: $P(T)$ for all T such that $|T| < |S|$

Show: $P(S)$, that is, $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

Tricky because it's possible that |T.insert(c)| = |T|

Inductive case (S was made by a call of the form "T.delete(c)"):

Assume: $P(T)$ for all T such that $|T| > |S|$

Show: $P(S)$, that is, $\neg\exists c_1, c_2 \in X$ [toLowerCase(c_1) = toLowerCase(c_2)]

Proof by induction over the computation

Any LetterSet was constructed by a sequence of calls like this:

LetterSet().insert(c1).insert(c2). … .delete(c3).insert(c4)

Assume property holds for this object

Prove property holds for this object

Also prove the base case

A new goal

Goal: prove that a large enough LetterSet contains two different letters

Property $P(X) = |X| > 1 \Rightarrow (\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

Prove $P(S)$; that is: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S\ [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

How might S have been made?

$\xrightarrow{\text{constructor}} S$ Base case

$T \xrightarrow{\text{T.insert(c)}} S$ Inductive case

ignore delete(c) to keep the proof short

Goal: prove that a large enough LetterSet contains two different letters

Property $P(X) = |X| > 1 \Rightarrow (\exists c_1, c_2 \in X\ [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

Prove $P(S)$

Two possibilities for how S was made: by the constructor, or by `insert`

Base case: S = { }, (S was made by the constructor):

property holds (vacuously true)

Inductive case (S was made by a call of the form "T.insert(c)"):

Assume: $P(T)$, that is, $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T\ [\text{toLowerCase}(c_3) \neq \text{toLowerCase}(c_4)])$

Show: $P(S)$, that is, $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S\ [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

where S = T.insert(c)

= "if $(\exists c_5 \in T$ s.t. $\text{toLowerCase}(c_5) = \text{toLowerCase}(c))$

then T **else** T U {c}"

The value for S came from the specification of insert, applied to T.insert(c):

// modifies: this

// effects: $this_{post}$ = ***if*** $(\exists c_1 \in$ ***S s.t. toLowerCase(***c_1***) = toLowerCase(c))***

then $this_{pre}$

else $this_{pre}$ ***U {c}***

public void insert (char c);

(Inductive case is continued on the next slide.)

Goal: a large enough LetterSet contains two different letters.

Inductive case: S = T.insert(c)

Goal (from previous slide):

Assume: P(T), that is, $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T\ [\text{toLowerCase}(c_3) \neq \text{toLowerCase}(c_4)])$

Show: P(S), that is, $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S\ [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

where S = T.insert(c)

= "if ($\exists c_5 \in T$ s.t. toLowerCase(c_5) = toLowerCase(c))

then T **else** T U {c}"

Consider the two possibilities for S (from "if ... then T else T U {c}"):

1. If S = T, then P(S) holds by the induction hypothesis or assumption that P(T)
2. If S = T U {c}, there are three cases to consider:
   - |T| = 0: P(S) holds vacuously, since hypothesis ("|S| > 1") is false
   - |T| ≥ 1: We know that T did not contain a char of toLowerCase(c), so P(S) holds by the meaning of union

   We didn't need to use the induction hypothesis for this case

   - Bonus: |T| > 1: By inductive assumption, T contains different letters, so by the meaning of union, T U {c} also contains different letters

Conclusion

The goal is correct code

A proof is a powerful mechanism for ensuring correctness

Formal reasoning is required if debugging is hard

Inductive proofs are the most effective in computer science

Types of proofs:

- Verify that rep invariant is satisfied (today)
- Verify that the implementation satisfies the spec ("reasoning about code" lectures)
- Verify that client code behaves correctly (today)