CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
REASONING I

Autumn 2011

Proofs in the ADT world

- Prove that the system does what you want
  - 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
- Proof can be formal or informal
- Complementary to testing

Rep invariant

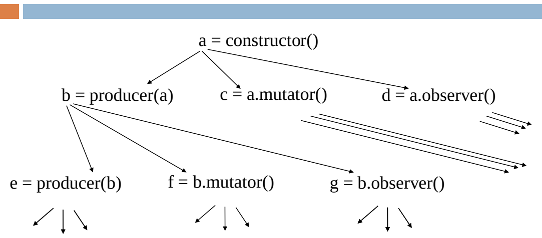
- Prove that all objects of the type satisfy the rep invariant
- Sometimes easier than testing, sometimes harder
- Every good programmer uses it as appropriate

- The follow techniques are used more broadly than for proving rep invariants – many proofs about programs have this flavor

All possible instances of a type

- Make a new object
  - constructors
  - producers
- Modify an existing object
  - mutators
  - observers, producers
- 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: prove infinitely many facts using a finite proof
- 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?

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`
  - observers: value
- Axioms
  1. `succ(0) > 0`
  2. `(succ(i) > succ(j))  ⇔  i > j`
- Goal: prove that for all natural numbers **`x`**, **`succ(x) > x`**
- Possibilities
  - **`x`** is 0 is true: `succ(0) > 0` — by axiom #1
  - **`x`** is `succ(y)` for some y
    - `succ(y) > y` — by assumption
  - `succ(succ(y)) > succ(y)` — by axiom #2
    - `succ(x) > x` — by def of x = succ(y)

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 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
- But why do we even need to check `member`?
  - The specification says that it does not mutate set
  - Reasoning must account for all possible arguments; the specification might be wrong; etc.

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 `c` is in $\texttt{elts}_{pre}$
  - `elts` is unchanged $\Rightarrow$ rep invariant is preserved
- If `c` is not in $\texttt{elts}_{pre}$
  - new element is not `null` (`Character` constructor cannot return null) or a duplicate (`insert` won't call `elts.add`) $\Rightarrow$ rep invariant is preserved

Reasoning about mutations

- 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
  - Basically, representation exposure allows side-effects on instances of the representation that are not easily visible

Reasoning about ADT uses

- Induction on specification, not on code
- Abstract values may differ from concrete representation
- Can ignore observers, since they do not affect abstract state
- Axioms
  - specs of operations
  - axioms of types used in overview parts of specifications

LetterSet (case-insensitive char set)

```
// LetterSet: mutable finite set of case-insensitive characters
// effects: creates an empty LetterSet
public LetterSet ( );
// Insert c if this contains no char with same lower-case rep
// modifies: this
// effects: this_post = if (∃c1∈ this_pre |
//                          toLowerCase(c1)=toLowerCase(c))
//                          then this_pre else this_pre ∪ {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 ( );
```

Prove some LetterSet contains two different letters

- Prove
  $|S|>1 \Rightarrow (\exists c_1, c_2 \in S \mid [\texttt{toLowerCase}(c_1) \neq \texttt{toLowerCase}(c_2)])$
- How might S have been made?

constructor → S

T —T.insert(c)→ S

constructor → S

T —T.insert(c)→ S = T

T —T.insert(c)→ S = T ∪ {c}

Base case

Inductive case #1

Inductive case #2

Full proof: two slides from Ernst

20

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

Prove: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [toLowerCase(c_1) ≠ toLowerCase(c_2)])
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: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T$ [toLowerCase(c_3) ≠ toLowerCase(c_4)])
Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [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}"
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.)

21

Goal: a large enough LetterSet contains two different letters.

Inductive case: S = T.insert(c)

Goal (from previous slide):
Assume: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T$ [toLowerCase(c_3) ≠ toLowerCase(c_4)])
Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [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, the theorem holds by induction hypothesis
   (The assumption above)
2. If S = T U {c}, there are three cases to consider:
   - |T| = 0: Vacuous case, since hypothesis of theorem ("|S| > 1") is false
   - |T| ≥ 1: We know that T did not contain a char of toLowerCase(h), so the theorem holds by the meaning of union
   - Bonus: |T| > 1: By inductive assumption, T contains different letters, so by the meaning of union, T U {c} also contains different letters

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

Prove: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [toLowerCase(c_1) ≠ toLowerCase(c_2)])
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: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T$ [toLowerCase(c_3) ≠ toLowerCase(c_4)])
Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [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}"
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: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T$ [toLowerCase(c_3) ≠ toLowerCase(c_4)])
Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S$ [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, the theorem holds by induction hypothesis
   (The assumption above)
2. If S = T U {c}, there are three cases to consider:
   - |T| = 0: Vacuous case, since hypothesis of theorem ("|S| > 1") is false
   - |T| ≥ 1: We know that T did not contain a char of toLowerCase(h), so the theorem holds by the meaning of union
   - Bonus: |T| > 1: By inductive assumption, T contains different letters, so by the meaning of union, T U {c} also contains different letters

Conclusion

- A proof is a powerful mechanism for ensuring correctness of code
- 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
  - Verify that the implementation satisfies the spec
  - Verify that client code behaves correctly

Next steps

- Friday: usability; Monday: UML; Wednesday: TBA
- A5 and A6