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

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

```
Example:  d = a.observer()
c = a.mutator()
b = producer(a)
a = constructor()
g = b.observer()
f = b.mutator()
e = producer(b)
```

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

- Infinitely many possibilities
- We cannot perform a proof that considers each possibility case-by-case
A counter class

```java
// 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

- ADD: the natural numbers (non-negative integers)
  - constructor: 0 // zero
  - producer: succ // successor: succ(x) = x+1
  - observer: value
- Axioms
  - succ(0) > 0
  - (succ(i) > succ(j)) → i > j
- Goal: prove that for all natural numbers x, succ(x) = x+1
- 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 characters.
  - constructor: CharSet() // creates a fresh, empty CharSet
  - public boolean member(char c); // modifies: this
    - public void insert(char c); // effects: creates a fresh, empty CharSet
      - public void delete(char c); // returns: cardinality of this
      - public int size();

 Implementation of CharSet

```java
// 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
  ```java
  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
  ```java
  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
  ```java
  public void insert(char c) {
    if (!this.member(c))
      elts.add(new Character(c));
  }
  ```
- If `c` is in `elts_pre`:
  - `elts` is unchanged \(\Rightarrow\) rep invariant is preserved
- If `c` is not in `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)

```java
// LetterSet: mutable finite set of case-insensitive characters
public LetterSet() {
  // Insert c if this contains no char with same lower-case rep
  // modifies: this
  // effects: this_post = if (\exists c_1 \in this_pre | toLowerCase(c_1) = toLowerCase(c))
  // then this_post else this_pre \cup \{c\}
  // public void insert(char c):
  // modifies: this
  // effects: this_post = this_pre \cup \{c\}
  public void delete(char c):
    // returns: (c \notin this)
    // public boolean member(char c):
    // returns: |this|
    public int size() {
    }
```
**Goal:** Prove some LetterSet contains two different letters.

- **Base case:** $\emptyset$ constructor
- $S = \emptyset 
- $T.insert(c) = S$

**Inductive case #1**

- $S = T.insert(c) = T 
- $T.insert(c) = S$

**Inductive case #2**

- $S = T \cup \{c\}$

---

**Goal:** Prove a large enough LetterSet contains two different letters.

**Inductive case:** $S = T.insert(c)$

**Goal (from previous slide):**

Assume: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T \ [toLowerCase(c_3) \neq toLowerCase(c_4)])$

Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S \ [toLowerCase(c_1) \neq toLowerCase(c_2)])$

where $S = T.insert(c) = \begin{cases} T & \text{if } (\exists c_5 \in T \ [toLowerCase(c_5) = toLowerCase(c)]) \\
T \cup \{c\} & \text{else} \end{cases}$

Consider two possibilities for $S$ from ‘if ... then $T \cup \{c\}$’:

1. If $S = T$, the theorem holds by induction hypothesis (The assumption above)
2. If $S = T \cup \{c\}$, 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(c)
   - Bonus: $|T| > 1$: By inductive assumption, $T$ contains different letters, so by the meaning of union, $T \cup \{c\}$ 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