Representation Invariants and Abstraction Functions

Announcements
- Exam 1 on Tuesday Oct. 6th
  - Closed book/phone/laptop, 2 cheat pages allowed
  - Reasoning about code, Specifications, ADTs
  - I will post Review slides and practice tests off Announcements page by Tuesday
- HW2 due today
- HW3 out today and due on Fri, before test
- Quiz 3 today, end of class

Outline
- Reasoning about ADTs
  - Representation invariants (rep invariants)
  - Representation exposure
  - Checking rep invariants
- Abstraction functions

Designing Data Structures
- From domain concept
  - E.g., the math concept of a polynomial, an integer set, the concept of a library item, etc.
- through ADT
  - Describes domain concept in terms specification fields and abstract operations
- to implementation
  - Implements ADT with representation fields and concrete operations

Specifying an ADT

<table>
<thead>
<tr>
<th>immutable</th>
<th>mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>class TypeName</td>
<td>class TypeName</td>
</tr>
<tr>
<td>1. overview</td>
<td>1. overview</td>
</tr>
<tr>
<td>2. specification fields</td>
<td>2. specification fields</td>
</tr>
<tr>
<td>3. creators</td>
<td>3. creators</td>
</tr>
<tr>
<td>4. observers</td>
<td>4. observers</td>
</tr>
<tr>
<td>5. producers</td>
<td>5. producers (rare!)</td>
</tr>
<tr>
<td>6. mutators</td>
<td>6. mutators</td>
</tr>
</tbody>
</table>

Example: Python Datatypes

- Tuples (e.g., (1,"John")) are immutable
- Lists (e.g., [1,2]) are mutable
- [j] len(t)
- [j:k] len(l)
- + [j:k] +
- * +
- l.append(e)
- l.reverse()
Why ADTs?
- Bridges gap between domain concept and implementation
- Formalizes domain concept. We can reason about correctness of the implementation
- Shields client from implementation. Implementation can vary without affecting client!

Reasoning About ADTs
- ADT is a specification, a set of operations
  - E.g., contains(int i), add(int i), etc., (the IntSet ADT)
  - add(Poly q), mul(Poly q), etc., (the Poly ADT)
- When specifying ADTs, there is no mention of data representation!
- When implementing the ADT, we must select a specific data representation

Implementation of an ADT is Provided by a Class
- To implement the ADT
  - We must select the representation, the rep
  - Implement operations in terms of that rep
  - E.g., the rep of our Poly can be
    a) int[] coeffs or
    b) List<Term> terms
- Choose representation such that
  - It is possible to implement all operations
  - Most frequently used operations are efficient

Connecting Implementation to Specification
- Representation invariant: Object \(\rightarrow\) boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object \(\rightarrow\) abstract value
  - What the data structure really means
    - E.g., array \([2, 3, -1]\) represents \(-x^2 + 3x + 2\)
  - How the data structure is to be interpreted

IntSet ADT
```java
/** Overview: An IntSet is a mutable set of integers. E.g., \(\{x_1, x_2, \ldots, x_n\}\). */
/** There are no duplicates. */
// effects: makes a new empty IntSet
public IntSet();
// modifies: this
// effects: thispost = thispre U \{x\}
public void add(int x);
// modifies: this
// effects: thispost = thispre \{-x\}
public void remove(int x);
// returns: (x in this)
public boolean contains(int x);
// returns: cardinality of this
public int size();
```

One Possible Implementation
```java
class IntSet {
    private List<Integer> data = new ArrayList<Integer>();
    public void add(int x) { data.add(x); }
    public void remove(int x) {
        data.remove(new Integer(x));
    }
    public boolean contains(int x) {
        return data.contains(x);
    }
    public int size() { return data.size(); }
}
```
The Representation Invariant

- `s = new IntSet(); s.add(1); s.add(1); s.remove(1);`

The representation invariant tells us what’s wrong with this code:

```java
class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data; ...
}
```

- States data structure well-formedness
  - E.g., IntSet objects, whose data array contains duplicates, e.g., [1,1], are not well-formed
  - Must hold before and after every concrete operation!
  - Correctness of implementation depends on it

```java
class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data = new ArrayList<Integer>();
    public void add(int x) {
        data.add(x);
    }
    // Rep invariant may not hold after add!
}
```

- Rep invariant excludes ill-formed concrete values
  - Conceptually, a line segment is defined by two distinct points. Thus, values with same start and end point (e.g., x1=x2=1, y1=y2=2), are meaningless. Rep invariant excludes them

```java
class LineSegment {
    // Rep invariant: !(x1 = x2 && y1 = y2)
    float x1, y1; // start point
    float x2, y2; // end point
}
```

```java
class RightTriangle {
    // Rep invariant: 0° < angle < 90° &&
    // base > 0 && base = hypot * cos(toRadians(angle))
    float base, hypot, angle;
    // Objects that don’t meet the above constraints are // not right triangles
}
```
Additionally...
- Rep invariant states: constraints imposed by specific data structures and algorithms
  - E.g., Tree has no cycles, array must be sorted
- Rep invariant states constraints between fields that are synchronized with each other
  - E.g., degree and coeffs fields in Poly (if we choose the array data representation)
- In general, rep invariant states correctness constraints --- if not met, things can go into terrible mess

Rep Invariant Example

```java
class Account {
    // Rep invariant:
    // transactions != null
    // no nulls in transactions
    // balance >= 0
    // balance = Σ_transactions.get(i).amount

    private int balance;
    private List<Transaction> transactions;
    ...
}
```

Another Rep Invariant Example

```java
class NameList {
    // Rep invariant: 0 ≤ index < names.length
    int index;
    String[] names;
    ...
    void addName(String name) {
        index++;
        if (index < names.length)
            names[index] = name;
    }
}
```

More Rep Invariant Examples

```java
class Poly {
    // Rep. invariant: degree = coeffs.length-1  
    // and more ...
    private int degree;
    private int[] coeffs;

    // operations add, sub, mul, eval, etc.

    private List<Term> terms;
    ...
}
```

Representation Exposure

- Suppose we add this operation to our IntSet ADT
  - returns: a List containing the elements
    ```java
    public List<Integer> getElements();
    ```
  - Suppose we decide on the following implementation
    ```java
    public List<Integer> getElements() {
        return data;
    }
    ```
  - What can go wrong with this implementation?

Representation Exposure

- Client can get control over rep and break the rep invariant! Consider
  ```java
  IntSet s = new IntSet();
  s.add(1);
  List<Integer> li = s.getElements();
  li.add(1);  // Breaks IntSet's rep invariant!
  ```
  - Representation exposure is external access to the rep. **AVOID!!!**
  - If you allow representation exposure, document why and how and feel bad about it.
Representation Exposure

- Make a copy on the way out:
  ```java
  public List<Integer> getElements() {
    return new ArrayList<Integer>(data);
  }
  ```
- Mutating a copy does not affect IntSet's rep
  ```java
  IntSet s = new IntSet();
  s.add(1);
  List<Integer> li = s.getElements();
  li.add(1); // mutates new copy, not IntSet's rep
  ```

- Make a copy on the way in too:
  ```java
  public IntSet(ArrayList<Integer> elts) {
    data = new ArrayList<Integer>(elts);
  }
  ```
- Why?

Representation Exposure

- How about this:
  ```java
  class Movie {
    private String title;
    ...
    public String getTitle() {
      return title;
    }
  }
  ```
- Technically, there is representation exposure
- Representation exposure is dangerous when the rep is mutable
  - If the rep is immutable, it's ok

Immutability, again

- Suppose we add an iterator
  ```java
  // returns: an Iterator over the IntSet
  public Iterator iterator();
  ```
- Suppose the following implementation:
  ```java
  public Iterator iterator() {
    return new Iterator(data);
  }
  ```

Immutability, again

```java
class Iterator {
  private List<Integer> theData;
  private int next;
  public Iterator(List<Integer> data) {
    theData = data;
    next = 0;
  }
  public boolean hasNext() {
    return (next < theData.size());
  }
  public int next() {
    return theData.get(next++);
  }
}
```

Checking Rep Invariant

- `checkRep()` or `repOK()`
- Always check if rep invariant holds when debugging
- Leave checks anyway, if they are inexpensive
- Checking rep invariant of `IntSet`
  ```java
  private void checkRep() {
    for (int i=0; i<data.size; i++)
      if (data.indexOf(data.elementAt(i)) != i)
        throw RuntimeException("duplicates!");
  }
  ```
Practice Defensive Programming

- Assume that you will make mistakes
- Write code to catch them
- On method entry
  - Check rep invariant (i.e., call checkRep())
  - Check preconditions (requires clause)
- On method exit
  - Check rep invariant (call checkRep())
  - Check postconditions
- Checking rep invariant helps find bugs
- Reasoning about rep invariant helps avoid bugs

Connecting Implementation to Specification

- Representation invariant: Object \( \rightarrow \) boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object \( \rightarrow \) abstract value
  - What the data representation really means
    - E.g., array \([2, 3, -1]\) represents \(-x^2 + 3x + 2\)
  - How the data structure is to be interpreted