Representation Invariants and Abstraction Functions

Announcements

- Exam 1 on Friday 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 next Tuesday

Outline

- Specifying an ADT
  - immutable
  - mutable
- Reasoning about ADTs
  - Representation invariants (rep invariants)
  - Representation exposure
  - Checking rep invariants (check rep)
- 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>
Poly, an immutable datatype:

Overview

```java
/**
 * A Poly is an immutable polynomial with integer coefficients. A Poly is:
 *  c₀ + c₁x + c₂x² + ...
 */

class Poly {
    // Specification (abstract) fields.
    More on this later.

    Overview: Always state whether mutable or immutable.
    Define abstract model for use in specification of operations. In ADTs state is abstract, not concrete (i.e.,
    this are NOT actual, implementation fields of Poly, just specification.)
}
```

Creators:

```java
// modifies: none
// effects: makes a new Poly = 0
public Poly() {
}

// modifies: none
// effect: makes a new Poly = cx^n
// throws: NegExponentException if n < 0
public Poly(int c, int n) {
}
```

Creators: This is example of overloading, two Poly
constructors with different signatures.

New object is part of effects not preexisting state. Hence,
modifies is none.

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)

Poly, an immutable datatype:

Observers

```java
// returns: degree of this polynomial
public int degree() {
}

// returns: the coefficient of the term of this polynomial, whose power is d
public int coeff(int d) {
}
```

Observers: Used to obtain information about this polynomial. Return values of other types. Never modify
the abstract state!

```
this: the current Poly object. Also called the receiver
Poly x = new Poly(5,4)
int c = x.coeff(3);
```

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)

Poly, an immutable datatype:

Producers

```java
// modifies: none
// returns: a new Poly with value this + q
public Poly add(Poly q) {
}

// modifies: none
// returns: a new Poly with value this*q
public Poly mul(Poly q) {
}
```

Producers: Operations on a type that create other objects
of the same type. Common in immutable types. No side
effects, i.e., cannot change abstract values of any
existing object

```java
Mutators:

// modifies: this
// effects: this_set = this_pre U { x }
public void add(int x) {
}

// modifies: this
// effects: this_set = this_pre - { x }
public void remove(int x) {
```

Mutators: operations that modify receiver this. Rarely
modify anything other than this. Must list this in
modifies: clause. Typically, mutators have no return value.

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)

Fall 17 CSCI 2600, A Milanova (slide due to Michael Ernst)
Exercise: String

- Overview?
  - Immutable!
- Creators?
  - `String()`, `String(char[] value)`, `String(String original)`, ...
- Observers?
  - `charAt`, `compareTo`, `contains`, `endsWith`, ...
- Producers?
  - `concat`, `format`, `substring`, ...
- Mutators?

Exercise: The Stack datatype

```java
public Stack()
public boolean empty()
public E peek()
public int search(Object o)
public E push(E item)
public E pop()
```

Example: Python Datatypes

- Tuples (e.g., `(1,"John")`) are **immutable**
- Lists (e.g., `[1,2]`) are **mutable**
- `len(t)`
- `t[j]`
- `t[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 (Java, C++)

- To implement the ADT
  - We must select the representation, the `rep`
  - Implement operations in terms of that `rep`
  - E.g., `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: this$\cap$ = this$\cup$ U \{x\}
public void add(int x);
// modifies: this
// effects: this$\cap$ = this$\cup$ -\{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

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

- States data structure well-formedness
  - E.g., IntSet objects, whose data array contains duplicates, e.g., $[1,1]$, are ill-formed.

- Must hold before and after every concrete operation!
- Correctness of implementation depends on it.

- 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 = new ArrayList<Integer>();
    public void add(int x) {
        data.add(x);
    }
    // Rep invariant may not hold after add!
}
```
### The Representation Invariant

**class IntSet**

```java
// Rep invariant:
// data has no nulls and no duplicates
private List<Integer> data = new ArrayList<Integer>;
public void add(int x) {
    if (!contains(x))
        data.add(x);
}

// If rep invariant holds before add, it
// holds after add too
```

**Rep invariant forbids ill-formed objects**

**class LineSegment**

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

// 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 forbids them
```

**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

### Another Rep Invariant Example

**class NameList**

```java
//Rep invariant: 0 <= index < names.length
int index;
String[] names; Where is the bug?
...
void addName(String name) {
    index++;
    if (index < names.length)
        names[index] = name;
}
```

**Rep Invariant Example**

**class Account**

```java
// Rep invariant:
// transactions != null
// no nulls in transactions
// balance >= 0
// balance = \sum_{i} transactions.get(i).amount
private int balance;
// history of transactions
private List<Transaction> transactions;
```

**Rep invariant states constraints** --- if not met, things can go into terrible mess
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.
}
```

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<>(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
  ```

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
// returns: an Iterator over the IntSet
public Iterator iterator();

Suppose the following implementation:
public Iterator iterator() {
    return new Iterator(data);
}

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
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

Aside: Invariants

Why focus so much on invariants?
  Loop invariants, rep invariants, immutability, which is a kind of invariant (immutable ADTs; modifies and effects clauses in the specification)
  Software is complex
    Lots of interactions between different “modules”. Interactions is what makes reasoning difficult
    Lots of “moving parts” (i.e., lots of changes)
Connecting Implementation to Specification

- **Representation invariant**: Object → boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful.
  - Defines the set of valid values.
- **Abstraction function**: Object → 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.

Abstraction Function:
rep → abstract value

- The abstraction function maps valid concrete data representation to the abstract value it represents. I.e., domain is all reps that satisfy the rep invariant.
- AF: Object → abstract value

The abstraction function lets us reason about behavior from the client perspective.

Another Abstraction Function Example

```java
class Poly {
    // Rep invariant: degree = coeffs.length-1
    // coeffs[degree] != 0
    private int[] coeffs;
    private int degree;
    // Abstraction function: coeffs [a_0, a_1, ..., a_degree]
    // represents polynomial
    // a_degree x^degree + ... + a_1 x + a_0
    // E.g., array [-2, 1, 3] → 3x^2 + x - 2
    // Empty array represents the 0 polynomial

    // Other methods...
}
```

Abstraction Function Example

```java
class IntSet {
    // Rep invariant:
    // data contains no nulls and no duplicates
    private List<Integer> data;
    // Abstraction function: data [a_1, a_2, ..., a_n]
    // represents the set { a_1, a_2, ..., a_n }
    // Empty list represents {}.
    ...
    public IntSet() {
        // Constructor...
    }
}
```

Another (Implementation of) IntSet

- What if we dropped the "no duplicates" constraint from the rep invariant?
- Can we still represent the concept of the IntSet? (Remember, an IntSet is a mutable set of integers with no duplicates.)

```java
class IntSet {
    // Rep invariant: data contains no nulls
    private List<Integer> data;
    ...
    public IntSet() {
        // Constructor...
    }
}
```
Yes. First, we have to change the abstraction function

```java
class IntSet {
    // Rep invariant: data contains no nulls
    private List<Integer> data;
    // Abstraction function: List data
    // represents the smallest set
    // \{ a_1, a_2, \ldots, a_n \} such that each a_i is
    // in data. Empty list represents \{ \}. 
    ...
    public IntSet() ...
}
```

Another IntSet

- We have to change implementation of operations as well
- What is the implication for `add(int x)` and `remove(int x)`? For `size()`?
- `add(int x)` no longer needs check `contains(x)`. Why?
- `remove(int x)` must remove all occurrences of `x` in `data`. Why?
- What about `size()`? What else?

Correctness

- Abstraction function allows us to reason about the implementation

Aside: the Rep Invariant

- Which implementation is better

```java
class IntSet {
    // Rep invariant: data has no nulls and no duplicates
    // methods establish & maintain invariant
    // and original abstraction function
    ...
    // methods maintain this weaker invariant
    // and new abstraction function
    ...
}
```
Aside: the Rep Invariant

- Often one role of the rep invariant is to simplify the abstraction function (by limiting valid concrete values which limits the domain of the abstraction function)

- Consequently, rep invariant simplifies implementation and reasoning!

Implementing an ADT: Summary so far

- Rep invariant
  - Defines the set of valid objects (concrete values)

- Abstraction function
  - Defines, for each valid object, which abstract value it represents

- Together they modularize the implementation
  - Can reason about operations in isolation
  - Neither is part of the ADT abstraction!!!

In practice

- Always write a rep invariant!

- Write an abstraction function when you need it
  - Write a precise and concise, but relatively informal abstraction function
  - A formal one is hard to write, and often not that useful
  - [As always with specs: we look for the balance between what is “formal enough to do reasoning” and what is “humanly readable and useful”]