cs2220: Engineering Software

Class 8:
Implementing Data Abstractions

Fall 2010
University of Virginia
David Evans

Menu

Implementing Data Abstractions

- Abstraction Function
- Representation Invariant

Recap: Abstract Data Types

Separate *what* you can do with data from *how* it is represented

Client interacts with data through provided operations according to their specifications

Implementation chooses how to represent data and implement its operations

Data Abstraction in Java

A **class** defines a new data type

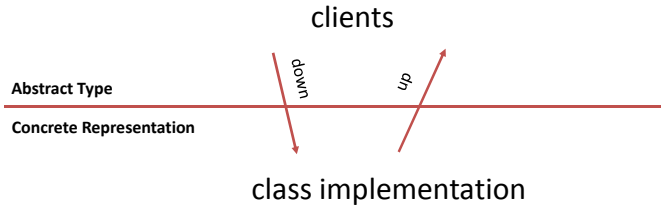
Use **private** instance variables to hide the choice of representation

private variables are only visible inside the class

```
public class StringStack {
  // Rep:
  private List<String> rep;
```

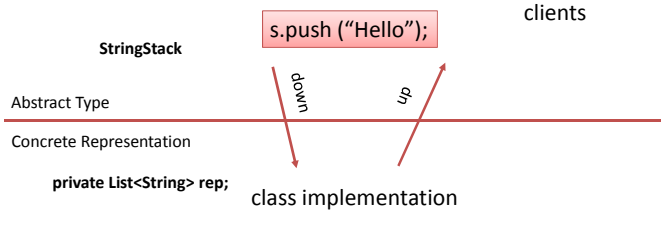
Up and Down

Clients manipulate an abstract data type by calling its operations (methods and constructors)



The representation of an abstract data type is visible only in the class implementation.

StringStack




```
public void push(String s) {
  rep.add(s);
}
```

Advantages/Disadvantages of Data Abstraction

- More code to write and maintain
- Run-time overhead (time to call method, lost opportunities because of abstraction)

+ Client doesn't need to know about representation
+ Can change rep without changing clients
+ Can reason about clients at abstract level

Choosing a Representation

Representation must **store the abstract state**

Think about how methods will be implemented

A good representation choice should:

Enable **easy implementations** of all methods

Allow **performance-critical methods** to be implemented efficiently

Use memory efficiently (if the data may be large)

Choosing the rep is the most important decision in implementing a data abstraction

StringStack Representation

Option 1: **private String [] rep;**

- Recall Java arrays are bounded
- Hard to implement **push**

Option 2: **private List<String> rep;**

- Easy to implement all methods
- Performance may be worse than for array

Implementing StringStack

```
public class StringStack {
  // Rep:
  private List<String> rep;

  /**
   * MODIFIES: this
   * EFFECTS: Pushes s on the top of this.
   *     For example, if this_pre = [ e_n-1, e_n-2, ..., e_1, e_0 ],
   *       this_post = [ s, e_n-1, e_n-2, ..., e_1, e_0 ]
   */
  public void push(String s) {
    rep.add(s);
  }
```

Is this implementation of **push** correct?

Could this implementation of **push** be correct?

It depends…

```
/**
 * MODIFIES: this
 * EFFECTS: If this is empty, throws EmptyStackException.  Otherwise,
 *   returns the element on top of this and removes that element from this.
 *   For example, if this_pre = [ e_n-1, e_n-2, ..., e_1, e_0 ],
 *       this_post = [ e_n-2, ..., e_1, e_0 ] and the result is e_n-1.
 */
public String pop() throws EmptyStackException {
  try {
    return rep.remove(0);
  } catch (IndexOutOfBoundsException e) {
    assert rep.size() == 0;
    throw new EmptyStackException();
  }
}
```

```
public String pop() throws EmptyStackException {
  try {
    return rep.remove(rep.size() - 1);
  } catch (IndexOutOfBoundsException e) {
    assert rep.size() == 0;
    throw new EmptyStackException();
  }
}
```

Is it correct?

- How can we possibly implement data abstractions correctly if correctness of one method depends on how other methods are implemented?
- How can we possibly test a data abstraction implementation if there are complex interdependencies between methods?

What must we know to know if **pop** is correct?

```
/**
 * MODIFIES: this
 * EFFECTS: If this is empty, throws EmptyStackException.  Otherwise,
 *   returns the element on top of this and removes that element from this.
 *   For example, if this_pre = [ e_n-1, e_n-2, ..., e_1, e_0 ],
 *       this_post = [ e_n-2, ..., e_1, e_0 ] and the result is e_n-1.
 */
public String pop() throws EmptyStackException {
  try {
    return rep.remove(0);
  } catch (IndexOutOfBoundsException e) {
     assert rep.size() == 0;
     throw new EmptyStackException();
  }
}
```

Abstraction Function

The **Abstraction Function** maps a concrete state to an abstract state:

$$\mathcal{AF}: \mathcal{C} \rightarrow \mathcal{A}$$

Function from concrete representation to the abstract notation introduced in Overview specification.

Abstraction Function for StringStack

```
/**
 * OVERVIEW: A StringStack represents a last-in-first-out stack where all
 *    elements are Strings.
 *    A typical stack is [ e_n-1, e_n-2, ..., e_1, e_0 ] where e_n-1 is the top
 *    of the stack.
 */
public class StringStack {
   // Rep:
   private List<String> rep;
```

Correctness of Push

```
/**
 * MODIFIES: this
 * EFFECTS: Pushes s on the top of this.
     For example, if this_pre = [ e_n-1, e_n-2, ..., e_1, e_0 ],
       this_post = [ s, e_n-1, e_n-2, ..., e_1, e_0 ]
*/
public void push(String s) {
    rep.add(s);
  }
```

Use abstraction function to show push satisfies its specification:

$$\mathcal{AF}(\text{rep\_post}) = [\mathcal{AF}_{\text{String}}(s)] + \mathcal{AF}(\text{rep\_pre})$$

```
/**
 * MODIFIES: this
 * EFFECTS: Pushes s on the top of this.
     For example, if this_pre = [ e_n-1, e_n-2, ..., e_1, e_0 ],
       this_post = [ s, e_n-1, e_n-2, ..., e_1, e_0 ]
*/
public void push(String s) {
    rep.add(s);
  }
```

How do we know **rep** is non-null?

Rep Invariant

Representation Invariant: properties all legitimate objects of the ADT must satisfy

$$I: \mathcal{C} \rightarrow \text{boolean}$$

Function from **concrete representation** to **boolean**.

Helps us reason about correctness of methods independently

Limits the range of inputs for the Abstraction Function

Reasoning with Rep Invariants

Prove all objects satisfy the invariant **before leaving** the implementation code

Assume all objects **passed in** satisfy the invariant

REQUIRES: Rep Invariant is true for this (and any other reachable ADT objects)
EFFECTS: Rep Invariant is true for all new and any modified ADT object on exit.

All non-private datatype operations have these specification implicitly!

Rep Invariant for StringStack

```
/**
 * OVERVIEW: A StringStack represents a last-in-first-out stack where all
 *     elements are Strings.
 *     A typical stack is [ e_n-1, e_n-2, ..., e_1, e_0 ] where e_n-1 is the top
 *     of the stack.
 */
public class StringStack {
  // Rep:
  private List<String> rep;
```

// Rep Invariant: I(c) = rep != null

Graph

```
public class Graph
  // OVERVIEW: A Graph is a mutable type that represents an undirected  graph.  It consists of
  //     nodes that are named by Strings,  and edges that connect a pair of nodes.
  //     A typical Graph is: < Nodes, Edges > where
  //       Nodes = { n_1, n_2, ..., n_m }
  //       Edges = { {a_1, b_1}, ..., {a_n, b_n} }  (the elements of Edges are unordered sets).

  public Graph ()
    // EFFECTS: Initializes this to a graph with no nodes or edges: < {}, {} >.

  public void addNode (String name) throws DuplicateException
    // MODIFIES: this
    // EFFECTS: If name is in Nodes, throws DuplicateException.
    //   Otherwise, adds a node named name to this:
    //    this_post = < Nodes_pre U { name }, Edges_pre >

  public void addEdge (String s, String t) throws NoNodeException, DuplicateException
    // MODIFIES: this
    // EFFECTS: If s and t are not names of nodes in  this, throws NoNodeException.  If there is
    //   already an edge between s and t, throws DuplicateEdgeException.   Otherwise, adds an
    //   edge between s and t to this:
    //       this_post = < Nodes_pre, Edges_pre U {node, node} >
```

s, t

Charge

Problem Set 3: Designing and Implementing Data Abstractions: Due Tuesday